Using Data-Flow Analysis for Resilience and Result Checking in Peer-To-Peer Computations

Samir Jafar, 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

{Samir.Jafar,Sebastien.Varrette,Jean-Louis.Roch}@imag.fr

Abstract— **In this paper, to achieve correct execution of peer-to-peer applications on non-reliable resources, we present a portable and distributed algorithm that provides fault tolerance and result checking. Two kinds of faults are considered: node failure or disconnection and result forgery. This algorithm is based on the knowledge of the macro dataflow dependencies beween the application tasks. It provides correct execution with respect to a probabilistic certificate. We have implemented it on top of Athapascan programming interface and experimental results are presented.**

Keywords—**Fault Tolerant, Checkpoint Recovery, Certificate of execution, Result Checking, Parallel Processing**

I. INTRODUCTION

LA rge scale distributed platforms, such as the GRID and Peer-to-Peer computing systems, gather thousands of nodes for computing parallel applications. At this scale, component failures, disconnections or results modifications are part of operation, and applications have to deal directly with repeated failures during program runs.

In this paper, we consider a large scale distributed platform where a secure system architecture such as Globus [10], [11] is deployed. Allocations 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. Moreover, to compete intrusion, the system provides strong authentication and secure communications. However, guaranteing the result of a parallel program execution is still an open problem. Indeed, even on such a secure environnement, two kinds of failures are distinguished.

- *node failures and disconnections*: To ensure resilience of the application, fault tolerance mechanisms are to be used. For the sake of scalability, a global distributed consistent state is needed; it is obtained from both checkpointing locally each sequential process and logging their dependencies (e.g. communications in MPICH-V[5] or Egida[20] presented in §II-A).
- *Task forgery*: the program is executed on a remote resource (also called *worker* in the sequel) and its expected output results may be modified on this resource 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 behaviour of a correct system for the outside. However there are more pernicious situations. Historically, the first infrastructure which highlighted task forgery issues was the SETI@Home project [26], [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; this undetected modification leaded to unusable results [18].

Since hardware approaches are not suited to the case of peer-to-peer computing platforms made of off-the-shelf standard components, managing failures is performed at the software level. Tasks that are detected failed are recomputed till correctness of the full execution. Since tasks in a peer-to-peer parallel application are mobile, the macro data flow that represents the tasks and their logical dependencies is known at least implicitly; the macro-dataflow is even explicit in some programming environments such as Jade [22]

or Athapascan [12].

In this paper, using the knowledge of the macro dataflow, we propose an unified framework to tackle both node failures and tasks forgery in a peer-to-peer parallel application. Indeed, assuming the existence of at least one trusted machine (also called *oracle*) and extending previous works in PORCH and MPICH-V (§II), the checkpointing of the macro dataflow provides a distributed portable global consistent state (§III). In section IV-B, we detail an asynchronous recovery algorithm based on this checkpointing.

Furthermore, taking benefit of analysis of data dependencies, we propose in section V-D a correction algorithm that provides a certification of the full application.

More precisely, we consider a peer-to-peer application G_0 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, if the number of tasks is large, the distribution of errors can be modelled as a Bernouilli distribution.

Using the checkpointing of the dataflow, we propose in section V-A an algorithm that implements the probabilistic forgery detection test introduced in [28]. This test is based on duplication of randomly chosen tasks on trusted machines (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 number of duplicated tasks required for certification is limited. Indeed, let ϵ be an arbitrary threshold, fixed by the user, that bounds the probability of considering correct an application while it is forged Then, the number of duplicated tasks required by our certification algorithm is $f_{\epsilon,q} = \frac{\ln[(1-q)^n(1-\epsilon)+\epsilon]}{\ln(1-q)}$ which is quickly negligible when n is large.

Our certification algorithm improves previous results on fault tolerance for peer-to-peer computations (cf §II). Concerning checkpoint/recovery, our mechanism brings portability; it supports heterogeneous nodes including symmetrical multi-processors. Concerning result checking, it extends previous approaches that are restricted to independent tasks and that define certification of a task with respect to the reliability of the resource where it is executed.

In order to evaluate the overhead of the proposed certification algorithm, we have implemented it on top of Athapascan [23]. Section VI comments the experimental measures, exhibiting that the overhead of the certification is small for a peer-to-peer parallel application with middle-grain tasks.

II. Related Work

In this section, we overview works on Fault-Tolerance and Result Checking in the software framework.

A. Fault-Tolerance by checkpointing

The study of fault tolerance mechanisms suited to peer-to-peer platforms is an active field of research [5], [14], [7], [16], [20], [9]. Namely two categories of applications are considered:

- **independent jobs**, like in the Condor [27] batch system: the application is a set of independent sequential jobs, each job being a process which may be dynamically created. The related fault tolerant mechanism [15] then consists in checkpointing each sequential process independently. This approach supports addition and resilience of resources, but is restricted to jobs without dependencies.
- **applications based on message passing**, most often developped with MPI or PVM.. The number of processors is fixed at the starting of the execution and the application consists in a fixed number of communicating processes. In MPICH-V [5], each MPI process is checkpointed independently at a given coordinated checkpoint [9]; to build a consistent global state, all communications performed by any process are logged [2]. Egida [20], provides a toolkit for PVM applications to support fault tolerance; this toolkit similarly enables the checkpointing of a global consistent state. In both MPICH-V and Egida, the checkpoint/recovery algorithm requires a memory space large enough to store all communication events between two checkpoints. Here, the recovery consists in replacing a failure node by a new node; but addition or resilience of resources is not supported.

Both cases are based on a memory core of each sequential process; also, they do not currently support neither restart on heterogeneous nodes nor checkpoint of concurrent multithreaded processes within a single process.

In order to deal with heterogeneous resources, a portable fault tolerant mechanism should be considered. In Porch [25], such a portable checkpoint is proposed for sequential architectures; it is based on the log of the procedure call stack (identity of the successive called procedures, values of their effective parameters and local variables).

Also, the satck describned a sequential consistent state. Thanks to the use of the Macro-Data Flow representation, we propose to extend this log mechanism to build a distributed and portable global consistent state.

However, the results of the application remain to be checked. Related works on result checking are expounded in the next section.

B. Result Checking

The execution of the application provides results that have to be checked. Basically, software certification consists in adding informations to the execution to accept/refuse the result(s) of the jobs. The objective is to minimize the certification cost, with a probability of certification error > 0 (as this is a software certification).

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

1. **"simple checkers"** [3] : 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. For instance, when resolving the Discrete Logarithm Problem (DLP, see [17] page 103-113), checking the correctness of a result can be done thanks to modular exponentiation and such computation is much faster than the DLP solving.
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 computation of the result has been performed in parallel on numerous peers. 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, as it will be described in §V-C, the knowledge of the dependencies graph provides a partial post-condition appliable to any program.
2. **duplication** [24], [13] : this approach is based on several executions of each task on many resources that are divided into workers and oracles. Workers are dedicated to perform task computation while **oracles** are used to check tasks results. Then, a reliable resource in a peer-to-peer platform may concurently play the role of a worker and an oracle while a non-trusted ressource can only be considered as a worker. Since the certification cost corresponds to the number of re-execution, duplication of all jobs generates an important additional cost. Assuming that the tasks are independent, C. Germain and N. Playez in [13] suggest to limit thhis overhead by using a sequential test of Wald [29] when testing a *batch* of n jobs.
Yet, this approach is limited to applications composed of independent tasks. We propose here an extension of this approach for the case of tasks with dependencies.

C. Importance of execution modeling

For both Fault-Tolerance and Software Certification, we need a representation of the application and more precisely of its execution. Such a representation is provided by dataflow description. This is the subject of the next section.

III. Distributed Checkpoint based on Data-Flow Representation

A. Data-Flow Mechanism

Our approach is based on the analysis of the dataflow. In that framework, the application is 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).

Modelling an execution by a dataflow graph is part of many parallel programming languages such as Jade [22] or Athapascan [12].

In the sequel, a leaf parameter in G is called a *terminal output*. Associated to a set of terminal outputs $\mathcal{S}$, the *terminal subgraph* is the subgraph $G_{\mathcal{S}}$ restricted to the ancestors of the vertices in $\mathcal{S}$. Figure 1 illustrates those notions.

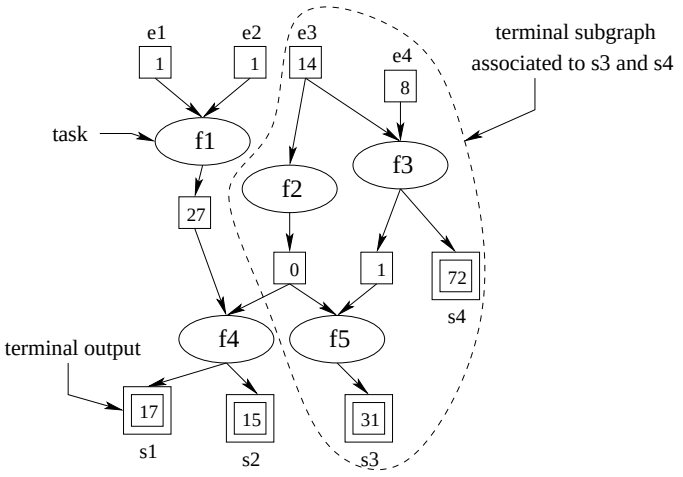


Fig. 1. Instance of a data-flow graph associated to the execution of five tasks $\{f_1, ..., f_5\}$. The input parameters of the program are $\{e_1, ..., e_4\}$ whereas the outputs (i.e the results of the computation) are $\{s_1, ..., s_4\}$.

The computation of the terminal subgraph $G_{\mathcal{S}}$ can be done in linear time of the size of the subgraph [6].

$G_{\mathcal{S}}$ is required for the certification of the terminal outputs contained in $\mathcal{S}$, thanks to tasks duplication. However, to ensure that the reexecution of a task deliver the same result, additionnal hypothesis are required. They are presented in the following section.

B. Deterministic task reexecution hypothesis

We exhibit hypothesis (H1, H2 and H3) on a macro dataflow parallel program that ensure deterministic results based on any individual tasks reexection. Those hypothesis are verified by most peer-to-peer applications:

- **H1 :** all synchronizations between tasks are explicitly described in the dataflow graph;
- **H2 :** a task is carried out until the end of its execution without synchronization. Consequently, once ready, a task can be executed non-preemptivly; it does not wait the results of any of its child task it has created;
- **H3 :** tasks are deterministic; any execution of a task with same input delivers the same result.

C. Distributed checkpoint

From previous hypothesis, a global consistent state of the application is defined:

Proposition 1. *assuming H1, H2 and H3, the macro dataflow graph describes a consistent global state.*

Our checkpoint mechanism is based on this proposition and relies on the macro dataflow graph. It consists in an asynchronous distributed systematic storage of each task (identifier and parameters) and of their data dependencies (identifier and related data value).

Atomic events are registered for each task declaration, start or completion. Those events are stored on a checkpoint server (SC) that provides a stable memory; it may be implemented by a transactional databases[8]. To ensure scalability, each node in the grid is related to a SC; but two nodes may be related to the same SC.

Moreover, this global state can be computed in a distributed way locally on each processor with no additional synchronization overhead. Figure 2 presents the principle of the checkpointing method; each task is independently checkpointed and its track is saved on a checkpoints server (SC) that can be centralized, herarchic or distributed.

On a theoretical point of view, this checkpoint algorithm avoids domino effects (e.g. the program is never restarted from initial state). Indeed, if the MTBF (Mean Time Between Failure) is larger than the maximal execution time τ of a task; then it is ensured that at least one task has been successfully completed and won't be reexecuted[1].

IV. Fault-Tolerance from Data-Flow Checkpointing

From the previous checkpoint of the macro dataflow, we propose in this section a recovery mechanism to resist to node failures and disconnections.

In distributed and parallel systems, the main source of failure/disconnection are the network and the nodes. In this section, we consider the failure/disconnection as node volatility[5]: the node is no more reacheable and the results computed by this node after the disconnection will not be considered in the case of a later reconnection. The failing node is supposed in a fail-silent mode[21].

[1] This property also enables to garbage successfully completed tasks, providing guaranteed bounds for memory space. Besides, since this checkpoint is performed at the macroscopic level of tasks, this compares advantageously to logging all inter-process communication events.

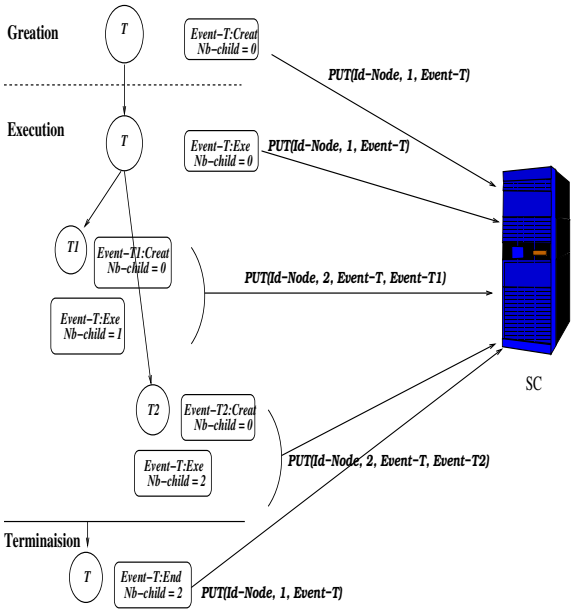


Fig. 2. Checkpoint method for a dataflow graph.

A. State automaton and recovery for a task

Since events registered in the checkpoint are atomic, in case of node failure, the checkpoint contains the last registered state for all tasks executed on this node. The various states of a task are described by the automaton in figure 3; indeed, after its start and before its termination, the current state of a task t is directly related to the number of children tasks it has created and which creations have been successfully registered.

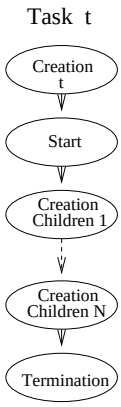


Fig. 3. Automaton for a task t.

The recovery mechanism is derived from this automaton. A task is restarted from its last successfully registered state before failure detection. Figure 4 exhibits the automaton related to fault tolerance.

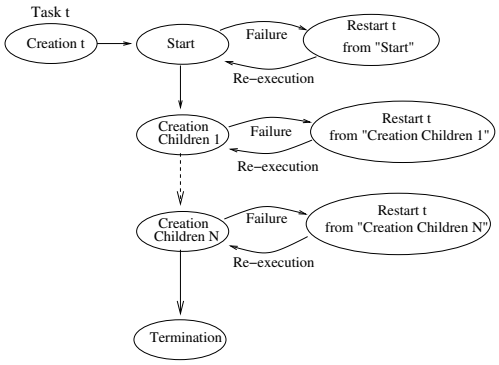


Fig. 4. Automaton that implements fault tolerance.

B. Recovery mechanism for a peer-to-peer computation

When a failure of a node is detected, it is handled by a module which is isolated in a secure enviroment (such as the oracles introduced in §I). This module is responsible for:
1. launching the program;
2. reacting to the addition or the resilience of nodes;
3. restarting the execution of failing nodes and eventually restarting all the execution.

The recovery algorithm is as follows. In case of a node failure detection, the stable memory related to the node is marked to be eventually uncompleted. This stable memory contains a set of events related to the dataflow graph, i.e. a subgraph (see §III-A). The recovery from this stable memory consists in the rebuilding of this subgraph : tasks that have not yet been completed are reexecuted, by respecting data dependencies. All objects that are part of the macro dataflow have a unique logical identifier which is defined at their creation and registered in the SC; this identifier remains the same until the execution of the full application is completed. The recovery algorithm is detailled in Algorithm 1.

Under hypotheses H1, H2 and H3 that ensure a deterministic program re-execution, this mechanism of Checkpoint/Recovery verifies the following properties at the time of a recovery :
1. an event is registered once and only once;
2. every task ends correctly its execution once and only once.

Algorithm 1: Recovery algorithm in case of failure

Data: $Dumpfile_i$: Checkpoint file of failing node i
Result: Launch re-execution of node i
//uncompleted tasks are restarted from their;
//last successfully registered state;
while $(!eof(Dumpfile_i))$ **do**
  //extract new valid event = a task,;
  //its arguments and its current valid state;
  (t,arg,state) = `getValidEvent(`$Dumpfile_i$`)`;
  //launch restart of the task from the last state;
  t.`Restart(`*arg, state*`)`;

Since the checkpoint avoids the domino effects, the recovery mechanism ensures that a complete execution of the application is perfomed in a finite number of re-executions.

The checkpoint mechanism makes it possible to access the state of the program and consequently to check its results. Using this checkpoint, we propose in the next section to limit the number of tasks duplication to provide a probabilistic level of certification arbitrary fixed by the user.

V. Result Checking from Data-Flow Checkpointing

After the execution of the application, a set of terminal outputs $\mathcal{S} = \{s_1, ..., s_m\}$ to certify is considered. $\mathcal{S}$ can contain all or part of the terminal outputs of the application. Those outputs are independent and their values result from the execution of tasks on one or several workers. From the analysis of the dataflow related to the application, the terminal subgraph $G_{\mathcal{S}}$ is computed (see §III-A). In the sequel, n refers to the number of tasks in $G_{\mathcal{S}}$.

The problem is then to decide whether or not $G_{\mathcal{S}}$ 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.

A. Monte-Carlo test of forgery

In this section, we provide a probabilistic certificate for detection of forged tasks in $G_{\mathcal{S}}$. This test is inpired from the Miller-Rabin Monte-Carlo test of composition (see [17] page 139) which 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, our test is a *Monte-Carlo test of forgery*.

A.1 Bound in the number of tasks to check

Let H_0 be the event "G_0 does not contain any forged tasks" and $H_1 = \overline{H}_0$ ("$G_{\mathcal{S}}$ contains at least a forged tasks"). Let G be a subset of k uniformly chosen tasks in $G_{\mathcal{S}}$. These tasks will be submitted to oracles. The *tester*, i.e the certification process, takes one of the following decisions:

- "ACCEPT" : no tested task was detected faked (i.e forged);
- "REJECT" : at least one of the tested tasks was detected faked.

The next proposition shows that if the number of tasks is large enough, then a partial duplication of only $N_{\epsilon,q}$ tasks, is sufficient to guarantee a given quality of certification (the risk of second kind is bounded by the arbitrary threshold ϵ). $N_{\epsilon,q}$ is a quantity independent from the number n of tasks.

Proposition 2. *Let n be the number of tasks of the program; let $P_{forgery}$ be the probability of tasks forgery;*

If $P_{forgery} \leq q$, then $\forall \epsilon > 0$, $\exists\, n_0 \,/ n > n_0 \Longrightarrow$ 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,

$$\begin{aligned} \beta &= 1 - \frac{\mathcal{P}(REJECT \cap H_1)}{\mathcal{P}(H_1)} \\ &= 1 - \frac{\mathcal{P}(REJECT)}{\mathcal{P}(H_1)} \\ &= \frac{(1-q)^k - (1-q)^n}{1-(1-q)^n} \end{aligned}$$

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 5 exhibits the evolution of $f_{\epsilon,q}(n)$ when n is increasing. We can see that it quickly tends to the value $N_{\epsilon,q}$, constant relatively to n. □

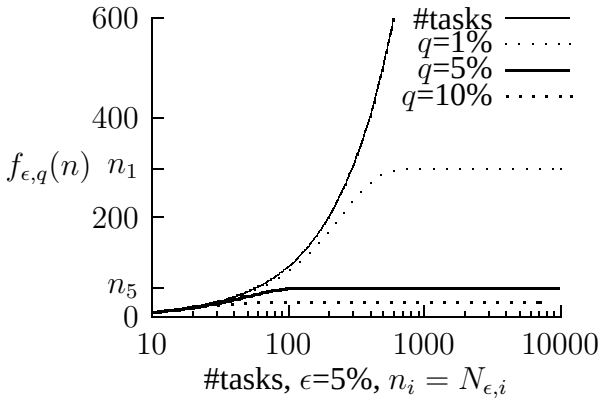


Fig. 5. Evolution of the minimum number of tasks to check relatively to the total number of tasks n to have $\beta \leq \epsilon$.

To illustrate the previous proposition, the following experiment has been set up.

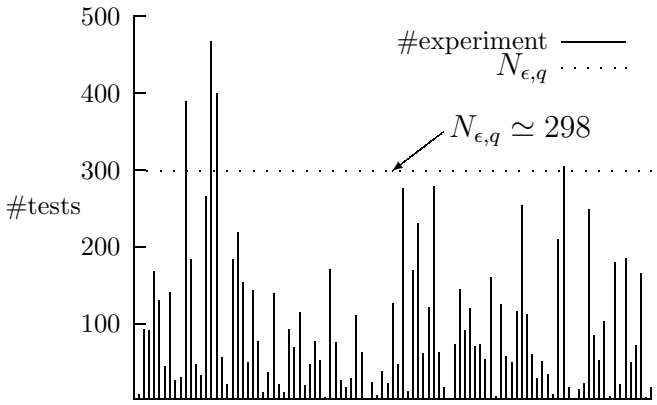
A program composed of $n = 10^4$ tasks is considered. The probability of tasks forgery is $q = 0.01$. The arbitrary certification rate is $\epsilon = 5\%$.

In each experiments, the number of tasks uniformly chosen to check before the first detection of forgery is computed. Figure 6 illustrates this algorithm repeated 100 times.

In practice for a peer-to-peer application, the total number n of tasks is large enough and thus $\min\{N_{\epsilon,q}, n\} = N_{\epsilon,q} = o(n)$ tasks have to be checked. For instance, in the context of the experiment described in fig. 6, $N_{\epsilon,q} \simeq 298$. Thus, the additionnal cost required for the certification is quickly negligible. This behaviour is illustrated by the experimentations done in §VI (figure 11).

This leads to a simple algorithm for error detection presented in the next section.

A.2 Detection Algorithm



Context : #tasks=10^6;q=1%;ϵ=5%

Fig. 6. Number of necessary tests before detection in 100 successive experiments.The value of $N_{\epsilon,q}$ is also represented in this context.

The proposition 2 directly leads to a Monte-Carlo test of forgery: either the test ends after $N_{\epsilon,q}$ successful checks and $G_{\mathcal{S}}$ is accepted; or else an error has been detected.

It is important to notice that in the experiment described in figure 6, this algorithm would have failed - i.e would have given the wrong answer - 4 times. Hence, in this context, $\beta = 4\%$ which is lesser than ϵ. This is a general behaviour.

This algorithm enables the detection of faked tasks by the execution of randomly chosen tasks. Yet, it requires that we have the possibility to test tasks independently to each other, and the way the tasks are checked has to be defined. Consequently, the inputs/outputs of the tasks have to be identified. That's where dataflow representation is required.

B. Safe tasks checker

In the previous algorithm, 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 7.

Thanks to the input parameters of the task (extracted from the checkpoint of the dataflow), a reexecution of the task can be perform. The results of this reexecution have to match the previous output results already stored in the dataflow checkpoint; otherwise, the task has been faked.

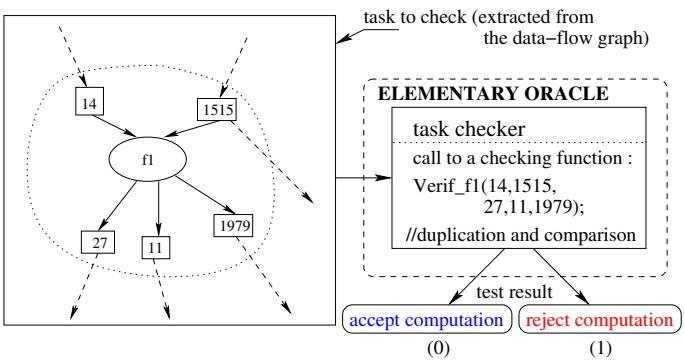


Fig. 7. Running of an elementary oracle

In the sequel, $\mathcal{O}_e(t)$ indicates a call to an elementary oracle to check the execution of the task t.

C. A Generic Partial Post-Condition

The dataflow introduced in §III-A, figure 1 is also used to provide 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.1. 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.1). This partial graph is a summary of the execution track called the **certification track**. It only describes the tasks to be executed and their dependencies. It has been generated on reliable resources (oracles) and verify the following properties:

Proposition 3. • *the certification track is a summary of the execution track;*
• *a partial execution is sufficient to generate it;*
• *any correct execution of the program (with the same inputs) on a remote unsecure worker supplies an execution track which summary has to correspond to the certification track.*

Therefore, 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, it makes it possible to control the general structure of the executed DAG. For instance, this post-condition would have managed to detect the patched clients for seti@HOME.

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.

D. A certification algorithm with error correction

§V-A.2 proposed an algorithm for forgery detection. Moreover, the execution track allows to identify the relevant tasks of the program and to access to their execution context (such as the values of inputs/outputs parameters).

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;
• 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 detailled in Algorithm 2.

Algorithm 2: Dynamic parallel certification algorithm with error correction

Data: $G_{\mathcal{S}}$: execution track to certify
Result: $\mathcal{O}(G_{\mathcal{S}})$
Check($\emptyset$,$G_{\mathcal{S}}$);

As the partial post-condition previously defined ensures the general structure of the program for any execution (tracks are supposed deterministic under hypothesis H1,H2,H3 defined in §III-B), 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, $C \leq \min\{N_{\epsilon,q}, n(G_{\mathcal{S}})\}$. Otherwise, in the case of error correction and if a certification is obtained after d detections then the additionnal cost for the full certification is $\leq (d+1)\min\{N_{\epsilon,q}, n(G_{\mathcal{S}})\}$. Of course, if the tasks of G_F are to be reexecuted, the unavoidable cost of this duplication has to be added to C.
The memory cost of the certification is $O(n(G_{\mathcal{S}}))$, and hence depends on the granularity of the graph. Moreover, there is a trade-off between the opera-

Procedure `Check`

Input: G_F : subgraph of forged tasks and their successors,
G_C : the rest of the graph

//Note that $G_C \cap G_F = \emptyset$;
$G = G_C \cup G_F$;
TasksChecked = 0;
repeat
  Pick up a new task t uniformly chosen among $n(G)$;
  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 *has to be executed again*;
      `LaunchExecution(`G_F`)`;
      //Checking the tasks of G_C *can be pursued*;
      //while G_F *is beeing executed*;
      `Check(`G_F,G_C`)`;
    else
      TasksChecked+=1;
until *TasksChecked* == $\min\{N_{\epsilon,q}, n(G)\}$;

tions 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.

VI. Experimental results

We have implemented this fault tolerant distributed mechanism on top of Athapascan [23], [12]. Athapascan is developped by the INRIA Apache project [19]; it is a macro-dataflow parallel language (C++ library) dedicated to distributed architectures including SMP nodes.

An Athapascan program describes only computations to be performed and their dependencies: parallelism is described at the grain of a procedure call (`Fork` instruction) that are the tasks; each task declares the way it accesses its effective shared parameters (access is typed `Shared_r` – resp. `Shared_w` – for a read – resp. write – access).

In order to register events related to each task and each shared object and to store them in the server of checkpoint related to the node, we have encapsulated Fork and Shared classes into two new classes, "FT::Fork" and "FT::Shared". Indeed, this should enable the use of our fault tolerance mechanism for any Athapascan program.

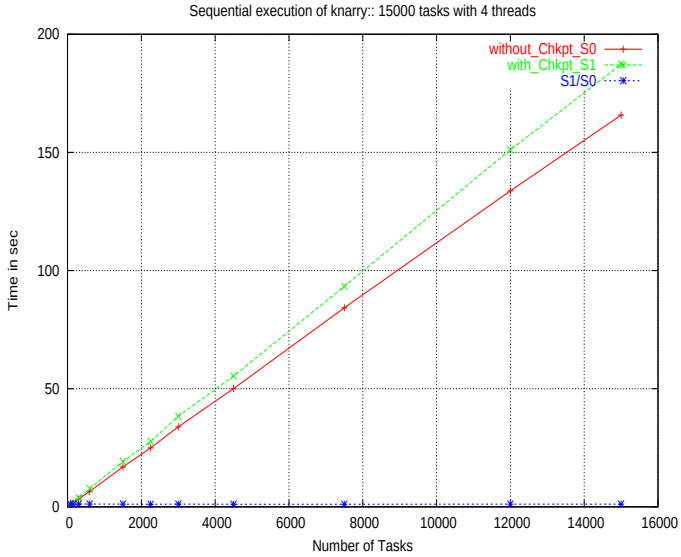


Fig. 8. Experimental performances with the application benchmark *knary* on a SMP with 4 processors.

Figure 8 exhibits experimental results on the *knary* benchmark [4] for recursive tree computations; this experimenal was made on a SMP node with 4 processors and the number of tasks varies between 75 and 1500. S_0 is the execution time without checkpoints; S_1 is the execution time with checkpoints. We remark that : $\frac{S_1}{S_0} \simeq 1$ $\frac{S_1 - S_0}{\#tasks} \simeq 1ms$. Thus, checkpoint overhead is constant, about 1 ms by task.

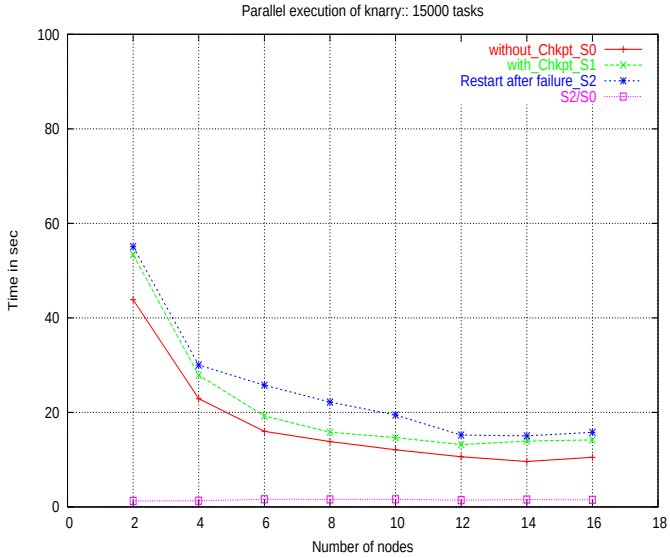


Fig. 9. Experimental performances with the application benchmark *knary* for a parallel execution on 16 nodes.

Figure 9 exhibits experimental results on the *knary* benchmark [4] for recursive tree computations (15000 tasks); the number of nodes varies between 2

and 16 nodes. The failure scenario is as follows: the execution starts on a cluster with 16 nodes (Pentium III, 733 MHZ, 256 MB, 15 GB, Ethernet 100 Mb/s); a failure occurs; recovery is performed from the files of checkpoints. S_0 is the execution time without checkpoints; S_1 is the execution time with checkpoints; S_2 presents the complete run time with 1-fault execution. We remark that :

$$\frac{S_1}{S_0} \simeq 1$$

$$\frac{S_1 - S_0}{\#tasks} \simeq 1ms$$

$$\frac{S_2}{S_0} \simeq 1$$

Thus, checkpoint overhead is constant too in this scenario. It can be seen that for tasks of 1ms, overhead is about 10% compared to a normal execution without fault tolerance; for longer unit tasks (0.1s) the overhead becomes lesser than 1%.

Concerning result checking, experimental results are exhibited in figures 10 and 11.

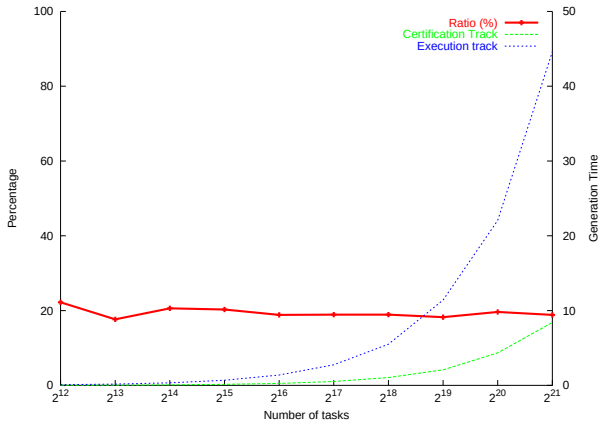


Fig. 10. Comparison between the times needed to generate both types of tracks relatively to the number of tasks in the tracks. The threshold used is 5%

Figure 10 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 11, 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 is compared to the certification time by partial duplication (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.

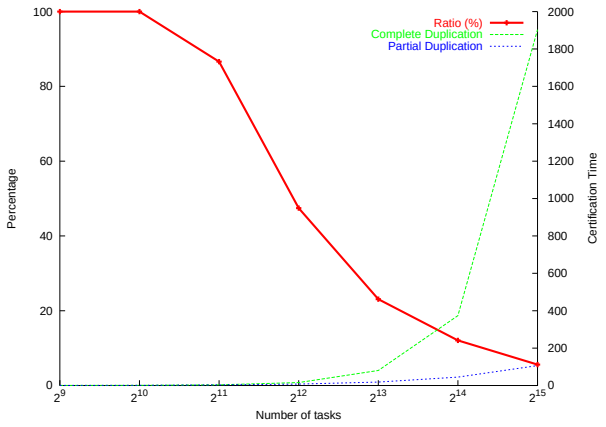


Fig. 11. Comparison between certification by complete and partial duplication. In the later case, the parameters are $\epsilon = 0,1$ and $q = 0,01$

VII. Conclusion and Perspectives

We have presented an execution algorithm for peer-to-peer applications that ensures robustness both to resource failure or disconnection and to result forgery. This algorithm is based on the checkpointing of the macro dataflow related to the application which describes both the tasks and their data dependencies. The checkpointing of the macro dataflow on reliable resources is asynchronous and distributed. It provides a portable mechanism to support resource failure and disconnection.

Furthermore, it enables reexecution of tasks. Then, detection of task forgery in the application is performed by duplication of randomly chosen tasks. Then a probabilistic certificate of the whole application is proposed based on the individual forgery probability of each task. The number of tasks to be duplicated is small: for instance, if probability of task forgery is 1%, 298 tasks only are duplicated in order to achieve a certification with probability 95%, whatever the number of tasks in the application is (fig. 6). If a task is detected forged, the checkpoint of the macro dataflow is used to invalidate and reexecute

only other dependent tasks till correction of the entire application.

Implementation of this algorithm on top of Athapascan system exhibits a small computational overhead that can be amortized for middle-grain tasks. Also, this algorithm is promising on a practical point of view and we are currently investigating its use for a medical application on a grid of resources in the framework of the french RAGTIME project.

When many tasks are dynamically created, there is an interesting tradeoff between the memory space required to store the checkpoint and the number of task duplications to be performed. On the one hand, the checkpoint/restart algorithm is distributed and enables to garbage tasks once they are completed; this property may be used to save memory space at the price of reexecuting all garbaged tasks in case of forgery detection. On the other hand, the efficiency of the probabilistic result checking algorithm direclty depends on the number of tasks to be certified: the more the tasks, the more efficient their checking. The certification algorithm we propose in this paper is motivated by minimizing the number of tasks to be reexecuted. Also, a perspective is a certification algorithm submitted to both time and memory space constraints.

REFERENCES

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

[2] L. Alvisi and K. Marzullo, "Message logging: Pessimistic, optimistic, causal, and optimal," *Software Engineering*, vol. 24, no. 2, pp. 149–159, 1998.

[3] M. Blum and H. Wasserman, "Software Reliability via Run-Time Result-Checking," *Journal of the ACM*, vol. 44, no. 6, pp. 826–849, Novembre 1997.

[4] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, 1995. [Online]. Available: citeseer.ist.psu.edu/blumofe95cilk.html

[5] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fédak, C. Germain, T. Hérault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Néri, and A. Selikhov, "Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes," in *SuperComputing 2002*, Baltimore, USA, 2002.

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

[7] M. E. and M. F.-J., "Fault-tolerant dynamic task scheduling based on dataflow graphs," in *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, Geneva, Switzerland, April 1997.

[8] A. K. Elmagarmid, *"Database transaction models for advanced applications"*. Morgan Kaufmann Publishers Inc., 1992.

[9] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message passing systems," School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, Tech. Rep. CMU-CS-96-181, Oct. 1996. [Online]. Available: citeseer.nj.nec.com/elnozahy96survey.html

[10] 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*, San Francisco, California, 3–5 Novembre 1998, pp. 83–92.

[11] ——, "Security for Grid Services," in *Twelfth International Symposium on High Performance Distributed Computing (HPDC-12)*, I. Press, Ed., Seattle, Washington, 22–24 Juin 2003.

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

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

[14] L. Juan, F. A. L., and S. Peter, "Fail-safe pvm: A portable package for distributed programming with transparent recovery, Tech. Rep. CMU-CS-93-124, Feb 93.

[15] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, "Checkpoint and migration of unix processes in the condor distributed processing system," Univ. Wisconsin, Madison, Tech. Rep. CS-TR-97-1346, 1997.

[16] G. Manimaran and C. S. R. Murthy, "A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 11, 1998.

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

[18] D. Molnar, "The SETI@Home Problem," November 2000.

[19] I. A. Project, "http://www.inrialpes.fr/apache.html," 2002.

[20] S. Rao, L. Alvisi, and H. M. Vin, "Egida: An extensible toolkit for low-overhead fault-tolerance," in *Symposium on Fault-Tolerant Computing*, 1999, pp. 48–55.

[21] J. Reisinger and A. Steininger, "The design of a fail-silent processing node for the predictable hard real-time system mars," *Distributed Systems Engineering Journal, pp. 104-111*, Jan. 1993.

[22] M. Rinard, "The design, implementation and evaluation of Jade : a portable, implicitly parallel programming language," *ACM Transactions on Programming Languages and Systems*, vol. 20, no. 3, pp. 483–545, Mai 1998.

[23] J.-L. Roch, T. Gautier, and R. Revire, "Athapascan: Api for asynchronous parallel programming," INRIA Rhône-Alpes, projet APACHE, Tech. Rep. RT-0276, Feb. 2003.

[24] 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.

[25] V. Strumpen, "Compiler technology for portable checkpoints," MIT Laboratory for Computer Science, Cambridge, Tech. Rep. MA-02139, 1998.

[26] W. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, and D. Anderson, "A new major seti project based on project serendip data and 100,000 personal computers," in *IAU Colloq. 161: Astronomical and Biochemical Origins and the Search for Life in the Universe*. Bologna, Jan. 1997, pp. 729–+.

[27] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the grid," in *Grid Computing: Making the Global Infrastructure a Reality*, F. Berman, G. Fox, and T. Hey, Eds. John Wiley & Sons Inc., December 2002.

[28] S. Varrette and J.-L. Roch, "Certification logicielle de calcul global avec dépendances sur grille," in *15èmes rencontres francophones du parallélisme (RenPar'15)*, M. Auguin, F. Baude, D. Lavenier, and M. Riveill, Eds., La-Colle-Sur-Loup, France, 15–17 Octobre 2003, pp. 169–176.

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