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— **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 data-flow dependencies between 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

Large 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 [5] provides strong authentication and secure communications. 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(§IV).
- *Task forgery*: the program is executed on a remote resource (also called *worker* in the sequel) and its expected output results 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. This may occur when the remote resource is the victim of a trojan horse or if the client software is modified on the remote resource, as experienced with SETI@Home [1], [9].

On peer-to-peer computing platforms, failures can be managed only at the software level. Failed tasks are recomputed till correctness of the full execution. Since tasks in a peer-to-peer parallel application are mobile and replicable, the macro data flow that represents the tasks and their logical dependencies is known at least implicitly or explicitly in environments such as Athapascan [13].

Using the knowledge of the macro data-flow, we propose an unified framework to tackle both node failures and tasks forgery in a peer-to-peer parallel application. Assuming the existence of at least one trusted machine (also called *oracle*) and extending previous works in PORCH and MPICH-V (§II), we use the macro data-flow to compute a distributed portable checkpoint (§III). This checkpointing of the data-flow provides both the asynchronous recovery in case of node failures (§IV) and the results certification in case of task forgery (§V). The certification algorithm we propose in is probabilistic: 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 propose in section V-A an algorithm that implements the probabilistic forgery detection test introduced in [16]. This test is based on duplication of randomly chosen tasks on trusted machines (oracles); communications and computations on oracles are assumed as totally reliable. To ensure a probability of non-detection of forgery lesser than ϵ, this test duplicates only $N_{\epsilon,q} = \frac{\ln \epsilon}{\ln(1-q)}$ tasks which is quickly negligible to n.

Our certification algorithm improves previous works on fault tolerance for peer-to-peer computations: it supports tasks dependencies on heterogeneous nodes including symmetrical multiprocessors and provides results certification. We have implemented it on top of Athapascan [13]; experimental measures (§VI) exhibit a small overhead 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, focusing on checkpoint/recovery approaches.

To support addition and resilience of resources for applications with independent jobs, the fault tolerant mechanism [7] developed in Condor consists in checkpointing each sequential process independently; To deal with message passing applications, and tasks with dependencies, MPICH-V[4] for MPI applications and Egida[11] for PVM dump independently each process at a given coordinated checkpoint; a consistent global state is built by logging all communications. The check-

point/recovery algorithm consists in replacing a failure node by a new node. This requires a memory space large enough to store all communication events between two checkpoints.

Those approaches are based on a memory core of each process and do not currently support neither restart on heterogeneous nodes nor checkpoint of concurrent multithreaded processes. To provide a portable checkpoint of a sequential program on heterogeneous resources, Porch[15] consists in logging the procedure call stack that describes a sequential consistent state. We propose to extend this portable log mechanism from a representation of the data-flow. Data-flow is used in [10] to provide fault-tolerance by replication; however, for global computations with many tasks, replication results in an inefficient use of CPU resources. Also, we propose to use the data-flow to compute a distributed and portable checkpoint.

Yet, the results of the application remain to be checked; since certification is performed by software, there is no absolute guarantee that 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 in adding informations to the execution to accept/refuse the result(s) of the jobs.

The *"simple checkers"*[2] approach consists in veryfing computed results thanks to a post-condition. This approach is simple and elegant when a post-condition is known that may be efficiently verified. Though, it is often impossible to automatically extract such post-condition on any program. Furthermore, if a computation is performed on numerous peers, the detection of a faked final result does not supply any information on the peer(s) responsible for the forgery. To tackle this problem, *"duplication"* approach[14], [6] is based on several executions of each task; non-trusted resources (*workers* are retracted to compute tasks while reliable ones may also check results (*oracles*). Duplicating all jobs would generate an important additional cost. To limit it, C. Germain and N. Playez propose a sequential test of Wald [17] a *batch* of n independent tasks[6] is checked by duplication of few randomly chosen tasks.

We extend this approach for the case of tasks with dependencies. For both distributed recovery and certification, we use a checkpoint of a portable representation of the tasks and all their dependencies, such as the data-flow representation introduced in the next section.

III. Distributed Checkpoint based on Data-Flow

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). 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. Note that $G_{\mathcal{S}}$ can be computed from G in linear time $O(|G|)$.

Furthermore, functional nature of data-flow enables both parallelism and fault tolerance [10]. In order to consider the current state of G as a portable checkpoint, we assume the following hypothesis (H1, H2 and H3):

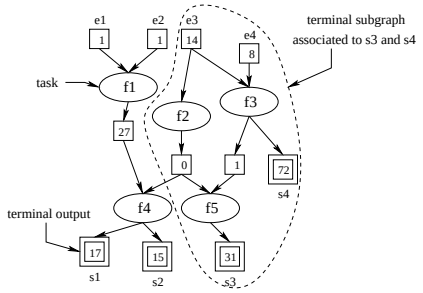

Fig. 1. Instance of a data-flow graph associated to the execution of five tasks $\{f_1, ..., f_5\}$. The input parameters of the program are $\{e_1, ..., e_4\}$ whereas the outputs (i.e the results of the computation) are $\{s_1, ..., s_4\}$.

(H_1) *Any synchronization between tasks is explicitly described in the data-flow graph.*
(H_2) *A task is carried out until the end of its execution without synchronization. Consequently, once ready, a task can be executed non-preemptively; it does not wait the results of any of its child task it has created.*
(H_2) *Tasks are deterministic; any execution of a task with same input delivers the same result.*

Those hypotheses are verified by most peer-to-peer applications; they ensure that any correct execution of any task will deliver the same output results.

Proposition 1. *Assuming hypotheses ($H_{1,2,3}$), the macro data-flow graph G describes a consistent global state.*

Our checkpoint mechanism is based on this proposition. 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) which implement atomic transactions.To ensure scalability, (SC) is hierarchic or distributed. Each node N in the grid is related to a proper stable memory SC_N. Then, the global state related to G can be computed in a distributed way locally on each processor. Figure 2 presents the principle of the checkpointing method. On any node N, each task is independently checkpointed and its track is saved on the stable memory SC_N related to N.

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. Also, successfully completed tasks are garbaged, providing guaranteed bounds for memory space[3].

IV. Fault Tolerance from Data-Flow

From the previous checkpoint of the macro dataflow, we propose in this section a recovery mechanism to resist to node failures and disconnections: failing node are supposed in fail-silent mode[12]. We consider the failures/disconnections as node volatility[4]: a volatile node is no more reacheable and, in the case of a later reconnection, its future results will be ignored. Fault-tolerance is managed by a module which is isolated in a secure environment, such as the oracles introduced in §I). This

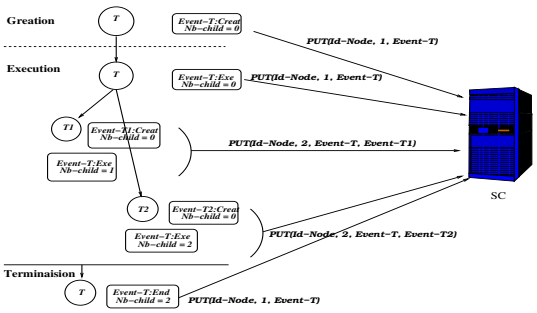

Fig. 2. Checkpoint method for a data-flow graph.

module is responsible for: launching the program; reacting to the addition or the resilience of nodes; recovering nodes that are detected failed.

When a node N is detected failed, its related stable memory SC_N is marked to be eventually uncompleted. SC_N contains a set of events related to a subgraph (see §III) of the dataflow graph G. The recovery from this stable memory consists in the rebuilding of this subgraph: all objects in G have a unique logical identifier which is defined at their creation and registered in the checkpoint server SC; this identifier remains the same until the execution of the full application is completed.

Recovering N then consists in recovering all tasks in SC_N that have not yet been completed while respecting data dependencies. The succesive atomic states of a task t are described in figure 3.A; they are checkpointed in SC_N. After the starting of t and before its termination, the current state of t is directly related to the number of task sucessfully created by t and registered on SC_N. The recovery of N is then derived from this automaton. Each task t in SC_N is restarted from its last successfully registered state (Figure 3.B).

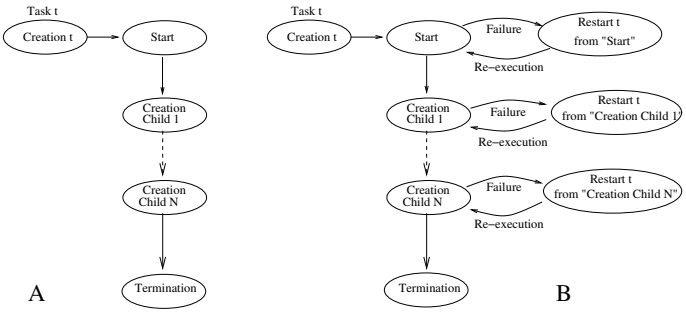

Fig. 3. A: State automaton of a task t. B: Recovery of a task t.

Under hypotheses ($H_{1,2,3}$), this recovery algorithm verifies that: modification events in G are registered once and only once; every task correctly ends its execution once and only once. Hence, the application is completed after a finite number of re-executions.

In the next section, we extend this recovery algorithm to check the computed results of the application from the checkpointing of G.

V. Result Checking from Data-Flow Checkpointing

The execution provides a set of terminal outputs $\mathcal{S} = \{s_1, ..., s_m\}$ to certify, and the associated terminal subgraph $G_{\mathcal{S}}$ can be computed thanks to the previous Data-Flow analysis. $\mathcal{S}$ can contain all or part of the terminal outputs of the application. 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) $\beta \leq \epsilon$. ϵ is an arbitrary threshold fixed by the user.

A. Probabilistic Monte-Carlo algorithm for forgery detection

In this section, we provide a probabilistic certificate for detection of forged tasks in $G_{\mathcal{S}}$. This test is inspired from the Miller-Rabin Monte-Carlo test of composition (see [8] 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*.

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 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 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 $N_{\epsilon,q}$ is a quantity independent from the number n of tasks.

Proposition 2. *Let consider an execution with n tasks and a probability of tasks forgery lesser than q. Then $\forall \epsilon > 0$, $\exists n_0 / \forall n > n_0$: 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, $\beta = 1 - \frac{\mathcal{P}(REJECT \cap H_1)}{\mathcal{P}(H_1)} = \frac{(1-q)^k - (1-q)^n}{1-(1-q)^n}$.Then $\beta \leq \epsilon \iff k \geq \frac{\ln[(1-q)^n(1-\epsilon)+\epsilon]}{\ln(1-q)} = f_{\epsilon,q}(n)$. 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 4 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. □

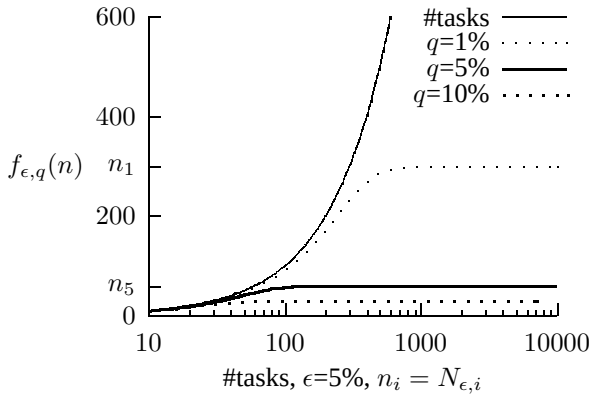

Fig. 4. Evolution of the minimum number of tasks to check relatively to the total number of tasks n to have $\beta \leq \epsilon$.

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.

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. Thus, the additional cost required for the certification is quickly negligible. For instance, in a program composed of at least 300 tasks with a probability of tasks forgery $q = 1\%$, only $N_{\epsilon,q} \simeq 298$ tasks have to be checked if the arbitrary certification rate is bounded to $\epsilon = 5\%$.

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 5. Thanks to the input parameters of a task t (extracted

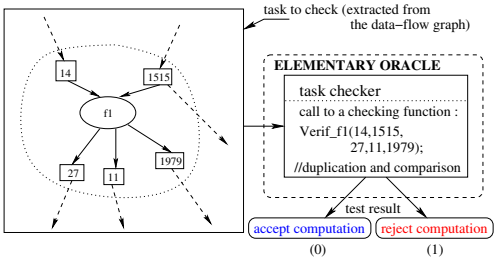

Fig. 5. Running of an elementary oracle

from the checkpoint of the data-flow), a re-execution of t can be perform and its results have to match the previous output results already stored in the data-flow checkpoint; otherwise, the task has been faked. In the sequel, $\mathcal{O}_e(t)$ indicates this operation.

B. Certification algorithm with forgery correction

In a certification with an arbitrary fixed threshold $\epsilon > 0$, $G_{\mathcal{S}}$ 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 %emph correct the forgeries. This algorithm is detailed in Algorithm 1.

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 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);
$G_C = G \backslash G_F$;
LaunchExecution(G_F) *//G_F must be re-executed*;
//Checking the tasks of G_C can be pursued;
//while G_F is being executed;
Check(G_F,G_C);
else
TasksChecked+=1;
until *TasksChecked* == $\min\{N_{\epsilon,q}, n(G)\}$;

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), 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 additional 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 re-executed, the unavoidable cost of this duplication has to be added to C. The memory cost of the certification is $O(n)$, 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.

VI. Experimental results

We have implemented this fault tolerant distributed mechanism on top of Athapascan [13]. Athapascan is a macro dataflow parallel language (C++ library) dedicated to distributed architectures including SMP nodes. The program describes only computations to be performed and their dependencies.

Figure 6 exhibits experimental results on the *knary* benchmark [3] 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

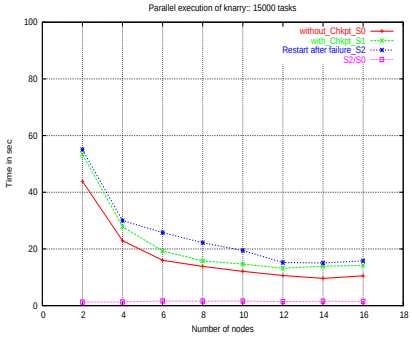

Fig. 6. Experimental performances with the application benchmark *knary* for a parallel execution on 16 nodes.

execution time with checkpoints; S_2 presents the complete run time with 1-fault execution. We remark that :

$$\frac{S_1}{S_0} \simeq 1.009; \quad \frac{S_1 - S_0}{\#tasks} \simeq 1ms; \quad \frac{S_2}{S_0} \simeq 1.01$$

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 figure 7. In this experiment no error was introduced during computation. In this context, we compare the certification time required by complete duplication (all tasks are reexucuted) and by partial duplication of $N_{\epsilon,q}$ tasks (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 favors the partial duplication approach in terms of certification time, even if the certification is then probabilistic. Note that, if $n = 8000$ $(resp.32000)$, then only 20% $(resp.10\ \%)$ tasks have to be checked.

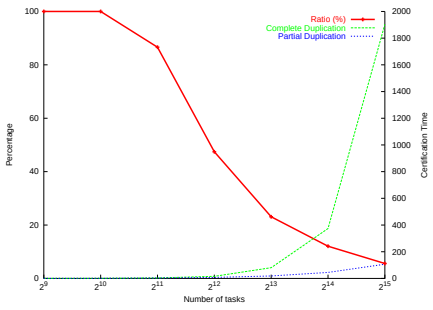

Fig. 7. Comparison between certification by complete and partial duplication. In the later case, the parameters are $\epsilon = 0,1$ and $q = 0,01$

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.

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 re-executing all garbaged tasks in case of forgery detection. On the other hand, the efficiency of the probabilistic result checking algorithm directly 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 re-executed. Also, a perspective is a certification algorithm submitted to both time and memory space constraints.

References

[1] "The SETI@Home project," 1999. [Online]. Available: setiathome.ssl.berkeley.edu

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

[3] 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

[4] 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.

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

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

[7] 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.

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

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

[10] A. Nguyen-Tuong, A. S. Grimshaw, and M. Hyett, "Exploiting data-flow for fault-tolerance in a wide-area parallel system," in *Proceedings 15 th Symposium on Reliable Distributed Systesm*, 1996, pp. 2–11.

[11] 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.

[12] 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.

[13] J.-L. Roch, T. Gautier, and R. Revire, "Athapascan: Api for asynchronous parallel programming," www-id.imag.fr/software/ath1, Projet APACHE, INRIA, Tech. Rep. RT-0276, Feb. 2003.

[14] L. F. G. Sarmenta, " Sabotage-Tolerance Mechanisms for Volunteer Computing Systems," in *ACM/IEEE Int. Symp. on Cluster Computing and the Grid (CCGrid'01)*, Brisbane, Australia, Mai 2001.

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

[16] 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.

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