Certification logicielle de Calcul Global avec dépendances sur grille.

Sébastien Varrette, Jean-Louis Roch

RENPAR'15 / CFSE'3 / SympAAA'2003
La Colle sur Loup, France, 15 au 17 octobre 2003

Certification logicielle de Calcul Global avec dépendances sur grille

Sébastien Varrette, Jean-Louis Roch

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

Résumé
De plus en plus d'applications gourmandes en ressources de calcul et en mémoire exploitent des infrastructures distribuées composées de l'interconnexion de machines géographiquement dispersées. On parle de grilles de grappes. Il se pose alors le problème de la certification de l'exécution d'un programme parallèle sur une grille non sécurisée. Ce domaine fait l'objet de nombreux travaux, tant au niveau matériel que logiciel. Nous proposons ici une méthode logicielle originale basée sur le calcul dynamique du flot de données associé à une exécution partielle du programme sur une machine sécurisée. Ce flot de données est un résumé de l'exécution : toute exécution complète du programme sur une machine distante non sécurisée avec les mêmes entrées fournit un flot dont le résumé doit correspondre à celui obtenu par exécution partielle.

Mots-clés : Certification d'exécution, calcul pair-à-pair, parallélisme

---

1. Introduction

L'utilisation de ressources distantes dans l'exécution d'un Calcul Global [2] fait l'objet de nombreuses recherches, en particulier dans le domaine de la sécurité. Si des architectures telles que Globus [4, 5] garantissent un certain nombre de services (authentification, intégrité des communication etc...), elles ne fournissent pas en général de certificat d'exécution. On entend par là garantir l'intégrité (en fait, le résultat) de l'exécution d'un programme parallèle sur une grille de grappes non sécurisée.
Dans cet article nous considérons le cas d'un programme parallèle avec dépendances, constitué de tâches ayant en entrée des paramètres (éventuellement produites par d'autres tâches) et en sortie des résultats (éventuellement consommées par d'autres tâches). Un tel programme peut être modélisé par un graphe biparti qui peut être généré dynamiquement (cas de création récursive de tâches).
Une certification robuste de l'exécution d'un programme sur une architecture distante nécessite que cette architecture dispose d'un matériel sécurisé spécifique [8]; différents travaux proposent des solutions matérielles dans ce cadre [11]. Néanmoins, cette approche matérielle n'est pas adaptée aux calculs pairs à pairs pour lesquels le matériel utilisé est standard. Aussi, plusieurs travaux proposent des solutions logicielles pour fournir des certifications intermédiaires dans le cas de tâches indépendantes (§ 2).
Dans cet article, nous proposons un algorithme de certification intermédiaire : étant donnée la probabilité q de falsification d'une tâche donnée, le problème est de certifier le résultat du calcul global avec une probabilité d'erreur de certification inférieure à un seuil ϵ arbitraire.
Nous montrons d'abord (§3) que le seuil d'erreur ϵ peut être atteint par duplication partielle via le test unitaire d'un nombre $N_{\epsilon,q}$ de tâches indépendant du nombre total de tâches du programme.
A partir de ce résultat, nous proposons une approche logicielle originale basée sur l'analyse des dépendances du programme parallèle qui s'exécute. L'intérêt est double. D'une part, la connaissance du graphe de dépendance des tâches, préalable à l'exécution des tâches qui le constituent, fournit un résumé partiel de l'exécution du programme qui peut être calculé avec un surcoût limité et permet de vérifier la conformité (partielle) du résultat de chaque tâche sans duplication (§4). D'autre part, en cas

de détection de falsification d'une tâche t, la connaissance de ce graphe permet d'invalider les tâches successeurs de t tout en poursuivant la certification partielle des autres tâches. Ainsi, on obtient un algorithme dynamique de certification qui évite la réexécution totale du programme (§5).
Finalement (§6), nous proposons une architecture logicielle distribuée implémentant cette approche et évaluons le surcoût requis par cette approche en comparaison à une réexécution complète.

2. Contexte de l'approche logicielle et limitations

Les approches logicielles pour la certification d'exécution peuvent être classées en deux catégories:

- *"simple checkers"* [1], basées sur la vérification d'une post-condition sur le résultat de la tâche, dans le cas où le coût de vérification est plus faible que celui du calcul lui-même. Il est souvent impossible d'extraire une telle propriété sur un programme quelconque, mais nous utiliserons la connaissance du graphe de dépendances comme une post-condition partielle (§4).

- *duplication complète* [10, 7], basée sur plusieurs exécutions de chaque tâche (ou *job*) sur des ressources différentes (*workers*). Parmi les jobs, on distingue ceux correspondant au calcul global (i.e. tâche séquentielle au sein du programme parallèle) et les *oracles* chargés de la vérification de ces tâches par réexécution.

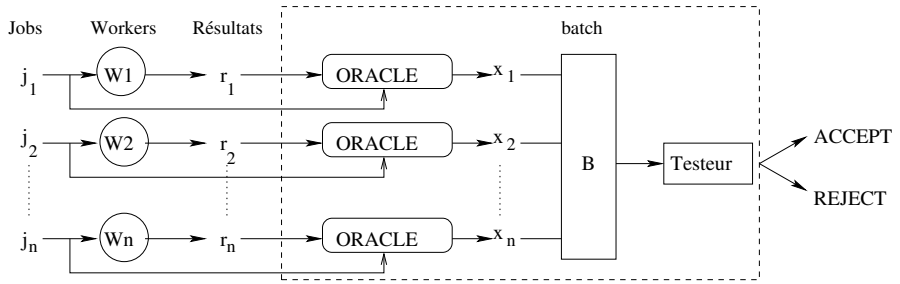

Figure 1: Test par duplication complète. Le résultat r_i d'un job j_i est soumis à un oracle de confiance qui se chargera de réexécuter le job. Cet oracle compare le résultat de la réexécution à r_i et renvoie un résultat binaire x_i (0=correct, 1=défaut). Si on considère un Calcul Global qui se décompose en n jobs indépendants j_i, le vecteur $[x_1, ..., x_n]$ est un *batch* soumis à un Testeur qui déterminera si la totalité du batch est acceptée ou rejetée.

La figure 1 présente le principe général du test par duplication complète. Ce modèle génère un surcoût important : le coût de la certification correspond au nombre d'appel aux oracles qui est égal à la taille du batch. C. Germain et N. Playez dans [7] proposent de limiter ce surcoût en utilisant au niveau du testeur un test séquentiel de Wald [13]. Un tel test garantit les objectifs fixés dans [7] à savoir assurer que le batch est globalement correct (tolérance aux fautes), ne pas éliminer à tort des résultats corrects et maintenir un coût de test faible. De plus, ce test est adaptatif (ie. recquiert plus ou moins d'oracles en fonction de l'allure du batch) et permet de faire évoluer dynamiquement le pool de ressources utilisées pour les oracles en fonction de leur comportement.
Pour un calcul global, le nombre de ressources sûres (ie. qui serviront d'oracles) reste très limité par rapport au nombre de ressources exploitées. De plus, on ne souhaite pas forcément pouvoir placer sa confiance dans des ressources distantes. Aussi, pour certifier le résultat d'un calcul global constitué de tâches indépendantes, [7] teste la fiabilité des workers par réexécution d'un nombre limité de jobs.
Dans toute la suite, on supposera que les oracles seront exécutés sur les ressources identifiées et fiables (en plus petite quantité que les ressources anonymes). On considèrera donc la réponse d'un oracle comme totalement fiable. De manière duale, nous proposons dans la section suivante de limiter le nombre d'appels à des oracles avec une probabilité d'erreur de certification arbitrairement bornée.

3. Certification par exécution partielle

Dans cette section, on se limite à la détection d'erreur. Soit G_0 un calcul global constitué de n tâches. Une tâche est dite *falsifiée* si le résultat qu'elle renvoie est erroné. On suppose qu'une tâche donnée est falsifiée avec une probabilité $q \in]0,1[$, de façon totalement indépendante des autres tâches.
Le problème est alors de décider si G_0 contient ou non des tâches falsifiées, avec une probabilité d'erreur de seconde espèce $\beta \leq \epsilon$, où ϵ est un seuil arbitrairement fixé par l'utilisateur.
Soit H_0 l'événement "Aucune tâche de G_0 n'est falsifiée" et $H_1 = \overline{H}_0$ ("Au moins une tâche de G_0 est falsifiée"). Soit G un sous-ensemble de k tâches uniformément choisies dans G_0 qui seront soumises aux oracles (sur le modèle de la figure 1). Le testeur prend l'une des deux décisions suivantes : "ACCEPT" (aucune tâche de G testée n'est falsifiée) ou "REJECT" (au moins une tâche de G testée est falsifiée).
Soit T_i le nombre de tâches falsifiées dans un ensemble $\mathcal{G}$ après i tests; T_i suit une loi binomiale $\mathcal{B}(i,q)$.
La proposition suivante montre que si le nombre de tâches n est suffisamment important, on peut se contenter de tester un nombre $N_{\epsilon,q}$ de tâches indépendant de n, pour garantir une qualité de certification donnée (l'erreur de seconde espèce est majorée par un seuil arbitraire).

Proposition 1 *Soit $\epsilon > 0$ un seuil fixé. Si $\frac{1}{n} \ll q$, alors il suffit de choisir uniformément $N_{\epsilon,q} = \frac{\ln(\epsilon)}{\ln(1-q)}$ tâches à vérifier pour garantir $\beta = \mathcal{P}(\text{ACCEPT}|H_1) \leq \epsilon$.*

Preuve : Soit k le nombre de tâches choisies uniformément pour le test.
On a : $\mathcal{P}(H_1) = 1 - \mathcal{P}(H_0) = 1 - \mathcal{P}(T_n = 0) = 1 - (1-q)^n$ et $\mathcal{P}(\text{ACCEPT}) = \mathcal{P}(T_k = 0) = (1-q)^k$.
Or, si le testeur renvoie 'REJECT', alors au moins une tâche de G_0 est falsifiée d'où

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

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

La figure 2 montre que lorsque n augmente, k tend rapidement vers la constante $N_{\epsilon,q}$.

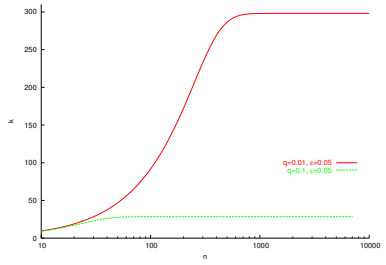

Figure 2: Evolution du nombre de tâches à vérifier (k) en fonction du nombre total de tâches (n) pour garantir $\beta \leq \epsilon$

Le tableau ci-dessous donne l'évolution du rapport $\frac{N_{\epsilon,q}}{n}$ (qui correspond au surcoût de la certification) en fonction de n.

n	$N_{\epsilon,q}/n$
10^4	30%
10^5	3%
10^6	0.3%

Les valeurs utilisées pour ce tableau sont $\epsilon = 0.05$ et $q = 10^{-}3$.
Comme la quantité $N_{\epsilon,q}$ est indépendante de n, le surcoût devient rapidement négligeable.

Les expérimentations du §6.2 (figure 8). illustrent ce comportement. Ce résultat est à la base de l'algorithme dynamique de certification (§5) qui évite la réexécution totale du programme. Mais avant, nous devons définir les concepts liés à l'analyse de dépendances et à notre approche : c'est l'objet de la section suivante.

4. Analyse de dépendances et résumé d'exécution

On modélise le programme parallèle par un graphe G orienté sans cycle biparti. La première classe de sommet est associée aux tâches et la seconde aux paramètres (d'entrée et de sortie) de ces tâches. On appellera dans la suite *sortie terminale* tout sommet associé à un paramètre de sortie sans successeur.
Les approches précédentes cherchaient à se ramener à des jobs indépendants et à certifier leurs calculs par un test sur la fiabilité des workers ; nous allons plutôt considérer un ensemble $\mathcal{S} = \{s_1, ..., s_m\}$ de sorties terminales $(s_1, \ldots, s_m)$ à certifier. Il se peut très bien que $\mathcal{S}$ ne contienne qu'une partie des sorties

terminales du programme. Ces sorties sont indépendantes et leur calcul résulte de l'exécution de jobs sur un ou plusieurs workers.
On note $G_{\mathcal{S}} \subset G$ le sous graphe restreint à tous les noeuds prédécesseurs des sorties terminales de $\mathcal{S}$. $G_{\mathcal{S}}$ est appelé sous-graphe terminal de $\mathcal{S}$. Ces différents concepts sont illustrés dans la figure 3.
Le calcul du sous graphe terminal $G_{\mathcal{S}}$ est réalisé en temps linéaire de la taille du graphe à partir d'un algorithme de type Bellmann [3] et d'un tri topologique du graphe.
$G_{\mathcal{S}}$ fournit alors un ensemble de tâches à vérifier. Un *oracle élémentaire* est défini comme étant un vérificateur de tâche opérant dans un milieu sécurisé. Son fonctionnement est illustré dans la figure 4 : à partir des paramètres d'entrées d'une tâche, il vérifie si les sorties d'une réexécution de cette tâche sont les mêmes que celles de l'exécution à tester.

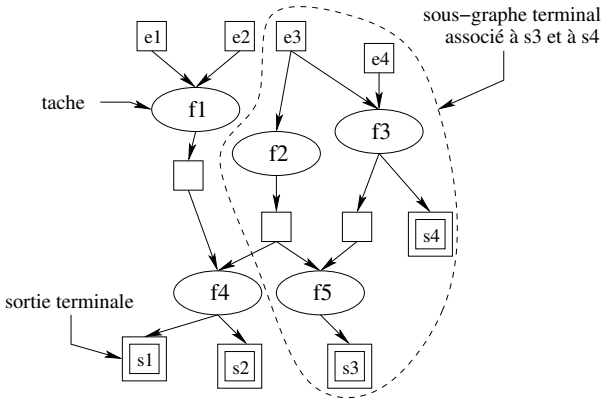

Figure 3: Un exemple de graphe de flot de données associé à l'exécution de 5 tâches $\{f_1, ..., f_5\}$. Les paramètres d'entrée du programme sont $\{e_1, .., e_4\}$ et ceux de sortie (résultat du calcul) $\{s_1, ..., s_4\}$.

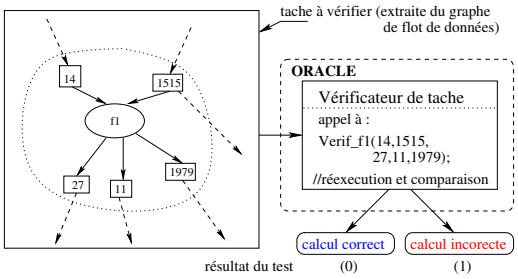

Figure 4: Fonctionnement d'un oracle élémentaire

La modélisation d'une exécution par un graphe de dépendances ou flot de données est implicite à de nombreux langages de programmation parallèle, par exemple Jade [9] ou Athapascan [6]; nous proposons d'adapter un tel langage pour générer le sous-graphe $G_{\mathcal{S}}$.L'exécution complète du programme fournit alors un graphe $G_{\mathcal{S}}$ (fig. 3) dans lequel la valeur effective des paramètres de chaque tâche est précisée; ce graphe est appelé dans la suite *trace d'exécution* du programme.
La connaissance de ce graphe sans considérer les valeurs effectives des paramètres (excepté pour les paramètres d'entrée du programme) fournit un résumé de la trace d'exécution appelé *trace de certification*, qui ne décrit que les tâches à exécuter et leurs dépendances.
Notre approche est basée sur la remarque suivante : une exécution partielle du programme suffit pour générer une trace de certification. De plus, en supposant les exécutions déterministes (cadre de l'étude), toute exécution complète du programme sur une machine distante non sécurisée (avec les mêmes entrées) fournit une trace d'exécution dont le résumé doit correspondre à la trace de certification. On obtient ainsi une post-condition partielle définie pour tout programme parallèle. Si elle ne permet pas de garantir l'exactitude du calcul effectué, elle fournit déjà un moyen de contrôler la structure du programme qui s'exécute. En outre, une fois cette post-condition vérifiée, il est possible d'exploiter cette trace d'exécution pour certifier les sorties de $\mathcal{S}$ et se prémunir des attaques qui ne modifieraient pas la structure du programme. C'est l'objet de la section suivante.

5. Algorithme de certification par exploitation des dépendances

La trace d'exécution $G_{\mathcal{S}}$ permet d'identifier chacune des $n(G_{\mathcal{S}})$ tâches du programme de façon unique et d'accéder à leur contexte d'exécution (valeur des paramètres d'entrée et de sortie). Ainsi, la recherche des arguments nécessaires aux oracles élémentaires (plus exactement aux fonctions de vérifications

qu'ils appellent : voir figure 4) est supportée efficacement. Dans la suite, $\mathcal{O}_e(t)$ désigne l'appel à un oracle élémentaire pour tester la tâche t.
Pour la certification avec un seuil $\epsilon > 0$ arbitraire, la trace d'exécution $G_{\mathcal{S}}$ est soumise à un oracle $\mathcal{O}$ de certification qui teste l'acceptation ou non de l'ensemble des tâches du graphe $G_{\mathcal{S}}$, avec une probabilité d'erreur de seconde espèce $\beta(G_{\mathcal{S}}) \leq \epsilon$.
L'algorithme de certification dynamique de $\mathcal{O}$ est le suivant :

- On choisit aléatoirement $k(G_{\mathcal{S}}, \epsilon) = \min(N_{\epsilon,q}, n(G_{\mathcal{S}}))$ tâches à tester parmi les $n(G_{\mathcal{S}})$ tâches.
- Pour chaque tâche t à vérifier, on appelle l'oracle $\mathcal{O}_e(t)$.
- En supposant que toutes les tâches testées sont correctes, on s'arrête au bout de $\min(k(G_{\mathcal{S}}, \epsilon), n(G_{\mathcal{S}}))$ tests et l'oracle peut alors renvoyer "ACCEPT".
- **Détection d'erreur.** Après la découverte d'une erreur, l'oracle renvoit "REJECT".
- **Correction des erreurs.** A la détection d'une erreur , le graphe $G_{\mathcal{S}}$ est partitionné en deux sous graphes indépendants : G_F d'une part (composé de la réunion des tâches t déjà détectées falsifiées et de tous leurs successeurs) et $G_C = G_{\mathcal{S}} \backslash G_F$ d'autre part.
  Les tâches de G_F doivent être réexécutées dans tous les cas et fournir un nouveau graphe G'_F. Soit alors $G' = G_C \cup G'_F$; comme la structure de $G_{\mathcal{S}}$ a déjà été certifiée (par la post-condition partielle du §4), la structure de G' est déduite de celle de $G_{\mathcal{S}}$. Il reste alors à soumettre G' à l'oracle qui réalise $k(G', \epsilon)$ tests de tâches. Il est à noter que les tests de G' portant sur les tâches de G_C peuvent être effectués en parallèle de la réexécution de G_F

Soit C le coût de certification en nombre d'opérations. Si aucune tâche testée n'est falsifiée, on a $C \leq k(G_{\mathcal{S}}, \epsilon)$. Sinon, en cas de correction d'erreurs, si on obtient une certification après d détections de tâches falsifiées, alors le surcoût de certification est majoré par $(d+1)k(G_{\mathcal{S}}, \epsilon)$ auquel s'ajoute le surcoût inévitable de la réexécution des tâches détectées falsifiées et de leurs successeurs.
Le coût mémoire de la certification est $O(n(G_{\mathcal{S}}))$ et dépend donc de la granularité du graphe à partir de laquelle on peut adapter le compromis entre nombre d'opérations et espace mémoire. En effet, plus la granularité est faible, plus le nombre de tâches est important et donc le temps de détection d'erreurs (majoré asymptotiquement par la constante $N_{\epsilon,q}$) est négligeable.
La section suivante va permettre d'illustrer les différents concepts définis dans les sections précédentes en les intégrant dans une architecture logicielle distribuée de certification.

6. Spécification d'une architecture de certification et expérimentations

6.1. Architecture de certification

Quatre étapes sont distinguées dans la certification de l'exécution d'un programme `prog.c` . (figure 5) :

1. *Génération des codes sources pour la certification* : on génère les trois codes suivants à partir d'un programme initial de calcul `prog.c` :

   - Code de génération d'une trace de certification : l'exécution de ce code fournit le graphe de flot de données partiel du programme initial. La génération d'un tel graphe ne nécessite qu'une exécution partielle du code source.
   - Code de génération d'une trace d'exécution : de façon similaire, l'exécution de ce code fournit le graphe de flots de données du programme initial (qui tiendra lieu de trace d'exécution) et exécutera le code source.
   - Code de vérification : on y trouve la liste des fonctions de vérification (cf § 6.2) qui seront utilisées par les oracles élémentaires de l'oracle de certification.

   La génération de ces codes peut être automatisée en se basant sur les modèles d'API macro-data-flow telles que Athapascan [6] ou Jade [9].

2. *Génération d'un certificat du programme* : le certificat est constitué d'une représentation de la trace de certification (soit la trace en elle-même, soit le code de génération de cette trace) d'une part, et le code de vérification d'autre part.

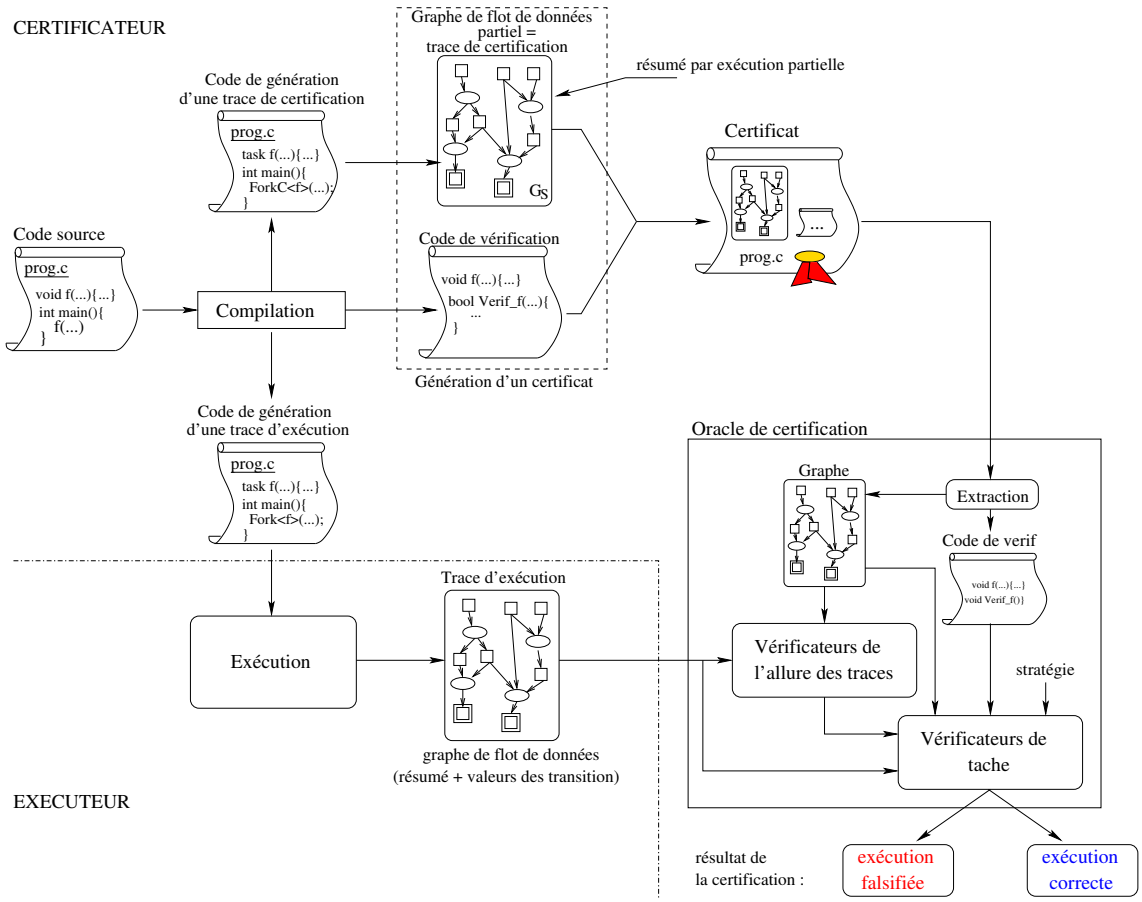

Figure 5: Architecture de la certification de l'exécution d'un programme prog.c

3. *Soumission du code de génération de la trace d'exécution à l'exécuteur*. Celui-ci lance alors le code qui en plus d'exécuter le programme initial fournit en sortie la trace d'exécution associée. Cette trace est soumise à l'oracle de certification.

4. *Traitement de la trace d'exécution par l'oracle de certification*. Celui-ci extrait d'abord du certificat la trace de certification et le code de vérification. Il procède ensuite à la vérification de l'allure des deux traces (post condition partielle définie dans le §4) avant de passer à la vérification des sorties terminales spécifiées par le certificateur à l'aide de l'algorithme exposé dans le §5. La vérification des traces est indépendante de celle des tâches : toutes peuvent être éxecutées en parallèle.

Un dernier point à traiter est la résistance à la résilience : il s'agit de tester la disponibilité des ressources distantes. La solution classiquement utilisée est la mise en place de challenges périodiques (ou éventuellement à pas adptatif); typiquement un challenge d'identification à clef publique de type SSL qui permet de garantir non seulement la présence d'une ressource mais également son authentification.
Néanmoins, une alternative est d'assimiler le challenge à un calcul particulier qui, coté worker, ne se distingue pas d'un vrai calcul pair-à-pair : ces tâches de challenge ne sont pas dupliquées lors de la certification. Sous ces hypothèses, la vérification des challenges permet d'évaluer la confiance que l'on peut placer dans la ressource distante.

6.2. Illustration et expérimentation

L'exemple simple de la figure 6 permet d'illustrer le comportement des fonctions de vérification déjà évoquées dans le §5. Lors de la vérification d'une tâche t appartenant à une trace d'exécution, la fonction de vérification associée crée le graphe d'exécution de t pour les paramètres d'entrée fournis.

Un appel à une création de tâche (`create_task`) ajoute un nouveau noeud au graphe. La valeur des paramètres manquants est obtenue à partir de la trace d'exécution (pour les tâches créées, par exemple `f` ou `sum`) ou par réexécution de fonctions séquentielles (cas d'un appel à une fonction séquentielle sans `create_task`, comme par exemple la fonction `g`).

```
void g(IN h, OUT res){...}
task sum(IN x, IN y, OUT res){
  res = x + y;
}
task f(IN n, IN seuil, OUT res){
  if (n <= seuil)
    g(n, res);
  else {
    OUT res1,res2;
    create_task   f(n-1,res1);
    create_task   f(n-1,res2);
    create_task   sum(res1,res2,res);
  }
}
int main(){  ... }
```

Figure 6: Source d'un programme qui calcule la valeur de 2^n par addition récursive et passage en calcul itératif sous un certain seuil.

Le graphe ainsi complété est alors comparé à celui de la trace d'exécution relatif à t.

Un premier prototype en C++ a été élaboré pour illustrer cet exemple sur l'architecture décrite dans la figure 5. Le formalisme XML est utilisé pour décrire les graphes. Ce premier prototype se limitait à détecter les erreurs et non à les corriger (voir l'algorithme exposé dans le §5).

Les premiers résultats expérimentaux obtenus sont illustrés dans les figures 7 et 8.

La figure 7 confirme la remarque du §4, à savoir qu'une exécution partielle suffit pour générer la trace de certification. On voit ainsi que la génération d'une trace de certification nécessite en moyenne 20% du temps de la génération de la trace d'exécution.

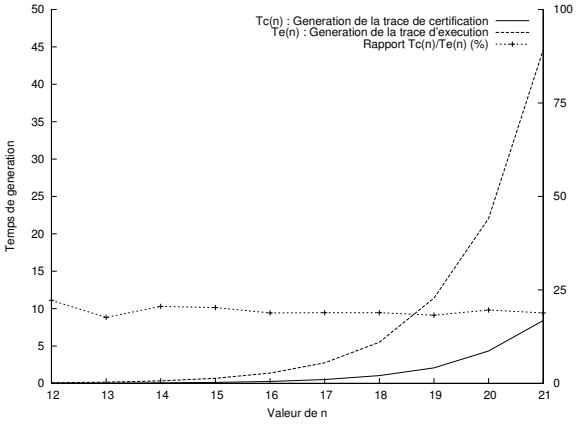

Figure 7: Evolution du temps de génération des deux types de trace en fonction de la taille du paramètre d'entrée n (le `seuil` est fixé à 5)

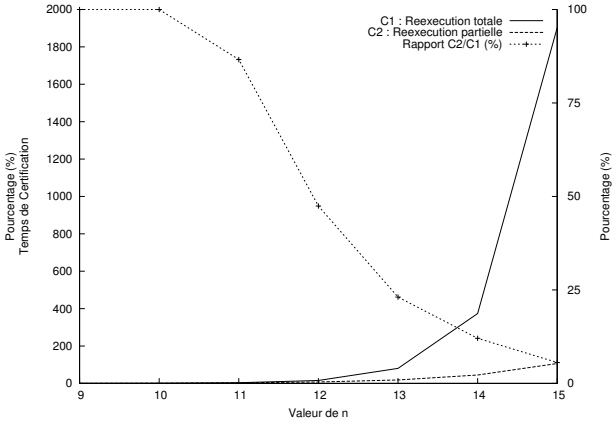

Figure 8: Comparaison entre les temps de certification par réexécution totale et par réexécution partielle de $N_{\epsilon,q}$ tâches (avec $\epsilon = 0.1$ et q=0.01) en fontion du paramètre d'entrée n

La figure 8 illustre expérimentalement le résultat théorique exposé dans le §4 en comparant le temps d'une certification par réexécution totale avec celui d'une certification par réexécution partielle de $N_{\epsilon,q}$ tâches. On voit que lorsque le nombre de tâches créés augmente (il est de l'ordre de 2^n où n est le paramètre d'entrée), la proportion de tâches à réexécuter pour garantir $\beta \leq \epsilon$ et donc le temps de certification diminue rapidement.

7. Conclusion et perspectives de travaux

Dans cet article, nous avons proposé un schéma de certification pour des programmes avec dépendances. La connaissance des dépendances via un graphe de flots de données permet deux niveaux de certification. D'une part, la structure du graphe fournit une post-condition partielle : la *trace de certification*. Cette post-condition est systématique et applicable à tout programme. De plus, une certification à un

niveau d'erreur paramétrable a été proposée. Asymptotiquement, et sous réserve que le nombre de tâches soit suffisamment grand, nous avons montré que ce niveau peut être atteint par la certification fiable de tâches choisies aléatoirement en quantité indépendante du nombre de tâches de l'exécution. Cette méthode, couplée avec la post-condition évoquée précédemment permet la détection de falsifications avec une probabilité d'erreurs paramétrable. Ce résultat est obtenu en supposant que tous les noeuds du graphe ont la même probabilité d'être falsifiés; une extension serait de considérer que les noeuds du graphe (voire machines de calcul) ont des probabilités de défaillance différentes.

D'autre part, en cas de détection, la trace d'exécution macroscopique que constitue le graphe de dépendances permet de corriger les erreurs en limitant la réexécution au sous graphe des tâches testées falsifiées et de leurs successeurs. De plus, la correction peut être menée en parallèle de la certification globale du graphe de tâches.

Enfin, une architecture de certification basée sur les concepts précédents est proposée. Les expérimentations menées sur un exemple didactique permettent d'évaluer le surcoût de génération des traces et témoignent de l'intérêt de cette certification par rapport à une réexécution complète.

Cette étude s'inscrit dans le cadre de l'ACI Grid-DOCG : dans ce contexte, nous étudions le developpement d'une architecture de certification optimisée et adaptée au passage à grande echelle sur une grille distribuée d'applications d'optimisation combinatoire. Les perspectives concernent l'expérimentation sur cette classe d'applications sur une grille où les noeuds ne sont pas tous équivalents. Un point particulier concerne alors l'estimation du taux de falsification d'une tâche, qui est a priori inconnu et dépendant de l'architecture d'exécution. Cette évaluation est importante pour déterminer le nombre de tests requis pour la certification partielle à taux d'erreur paramétrable proposée dans cet article. [12]

Bibliographie

1. M. Blum and H. Wasserman. Software Reliability via Run-Time Result-Checking. *Journal of the ACM*, 44(6):826–849, Novembre 1997.
2. F. Cappello. Calcul global pair à pair : extension des systèmes pair-à-pair au calcul. *Lettre de l'IDRIS*, 4(4):14–26, Février 2002.
3. T.H. Cormen, C.E. Leiserson, R.L. 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 IEEE 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. M.C. 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.
10. 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.
11. 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.
12. S. Varrette and J.-L. Roch. Certification logicielle de calcul global avec dépendances sur grille. In

M. Auguin, F. Baude, D. Lavenier, and M. Riveill, editors, *15èmes rencontres francophones du parallélisme (RenPar'15)*, pages 169–176, La-Colle-Sur-Loup, France, 15–17 Octobre 2003.
13. A. Wald. *Sequential Analysis*. Wiley Pub. in Math. Stat., 1966.