Génération et manipulation d'espaces d'états distribués avec CADP : expériences sur Grid'5000

Hubert Garavel, Radu Mateescu, Wendelin Serwe

ComPAS'2013 : RenPar'21 / SympA'15/ CFSE'9
Grenoble, France, du 16 au 18 janvier 2013

Génération et manipulation d'espaces d'états distribués avec CADP : expériences sur Grid'5000

Hubert Garavel, Radu Mateescu et Wendelin Serwe

Inria et LIG — équipe CONVECS
655, avenue de l'Europe, Inovallée Montbonnot, 38334 St Ismier Cedex, France
{Hubert.Garavel,Radu.Mateescu,Wendelin.Serwe}@inria.fr

Résumé
La vérification distribuée utilise les ressources d'un ensemble de machines pour accélérer et surtout pour accéder à des quantités de mémoire au delà de ce qui est possible avec une seule machine. Dans cet article, nous présentons les outils de vérification distribuée fournis par la boîte à outils CADP, en mettant l'accent sur les améliorations récentes des outils pour l'exploration à la volée de systèmes de transitions étiquetées partitionnés. Nous faisons aussi état des résultats d'expériences avec ces outils sur Grid'5000 utilisant jusqu'à 512 processus répartis.

Mots-clés : vérification distribuée, systèmes asynchrones, vérification à la volée

1. Introduction

La vérification énumérative d'un système parallèle est une méthode d'analyse automatique permettant une détection rapide et économique des erreurs (interblocage, perte de cohérence des données, violation d'exclusion mutuelle, présence de famine, etc.). A partir d'une description formelle du système dans un langage de haut niveau, un compilateur spécialisé génère un modèle de type graphe d'états ou STE (Système de Transitions Etiquetées), sur lequel opèrent les procédures de vérification. Pour les systèmes de grande taille, comportant beaucoup de processus parallèles et/ou des types de données complexes, la taille des STE devient prohibitive, nécessitant des ressources de calcul excédant celles des machines séquentielles disponibles. Une solution naturelle à ce problème est d'utiliser les ressources de calcul (notamment la mémoire) des grappes ou des grilles de machines afin d'augmenter les capacités des outils de vérification de plusieurs ordres de grandeur. La vérification formelle est ainsi devenue, durant la dernière décennie, une application fertile du parallélisme, dont la mise en œuvre nécessite à la fois des représentations des STE adéquates au calcul réparti et des algorithmes parallèles et distribués pour générer et manipuler ces STE.
La boîte à outils CADP [9] (*Construction and Analysis of Distributed Processes*[1]) permet la spécification et la vérification formelle de systèmes parallèles asynchrones. Le flot d'analyse traditionnel d'un système parallèle consiste à générer le STE correspondant, puis à le minimiser modulo une relation d'équivalence appropriée, et enfin à le vérifier selon une certaine méthode (vérification de propriétés en logique temporelle, vérification par équivalences, inspection visuelle,

[1] http://cadp.inria.fr

etc.). Une autre variante consiste à effectuer la vérification à la volée, en explorant le STE au fur et à mesure des besoins, ce qui permet de détecter des erreurs sans construire entièrement le STE. Cette deuxième approche est plus adaptée aux premières phases de conception, où les erreurs sont fréquentes. La première approche est plus efficace dans les dernières phases de conception, lorsque le STE doit être parcouru entièrement pour garantir l'absence d'erreurs.
CADP fournit plusieurs outils de vérification distribuée, en particulier DISTRIBUTOR et BCG_MERGE [11, 10], qui exploitent les ressources de plusieurs machines pour construire un STE partitionné (consistant en plusieurs fragments répartis sur plusieurs machines), puis de le transformer en STE monolithique (stocké dans un seul fichier), sur lequel opèrent les outils de vérification séquentielle. Pour analyser efficacement les systèmes de grande taille, il est préférable de manipuler le plus longtemps possible (idéalement, durant tout le flot d'analyse) des STE partitionnés. Dans cet article, nous présentons les outils pour la génération et la manipulation de STE partitionnés de CADP, en mettant l'accent sur leurs dernières améliorations. Nous décrivons également les résultats expérimentaux concernant l'utilisation de ces outils pour traiter des STE de grande taille sur Grid'5000 [7] et nous analysons le passage à l'échelle et les gains de performance obtenus en augmentant jusqu'à 512 le nombre de processus répartis.
Le reste de cet article est organisé comme suit. La section 2 décrit la bibliothèque de communication utilisée par CADP et le format pour le stockage de STE partitionnés. La section 3 présente les outils pour la génération et la manipulation de STE partitionnés. La section 4 détaille les résultats de nos expériences sur Grid'5000. La section 5 compare notre approche avec quelques travaux voisins. La section 6 conclut l'article et précise quelques pistes de recherche.

2. Format PBG pour les STE partitionnés

Une application de vérification distribuée typique consiste en N processus *travailleurs* s'exécutant sur des machines différentes et en un processus *maître* qui coordonne l'exécution des travailleurs et gère l'interaction avec l'utilisateur. Pour la communication par envoi de messages entre les travailleurs et le maître, CADP fournit la bibliothèque `caesar_network_1`, qui est basée sur les primitives standard des systèmes d'exploitation (notamment les sockets TCP/IP) et les protocoles standard d'accès à distance (notamment `ssh`).
La configuration de l'ensemble de machines à utiliser est décrite dans un fichier en format GCF (*Grid Configuration File*) qui spécifie les machines à utiliser, la façon de s'y connecter (nom de l'utilisateur, protocole à utiliser, etc.) et les répertoires de travail à utiliser par les travailleurs.
Pour le stockage des STE ordinaires, la boîte à outils CADP utilise le format BCG (*Binary Coded Graphs*), qui permet de stocker les états, transitions et étiquettes d'un STE de manière compacte (en utilisant un codage binaire et des techniques de compression dédiées). CADP fournit des outils et bibliothèques de manipulation de fichiers BCG, ainsi qu'une interface de programmation pour la création et l'accès aux fichiers BCG. Or, dans le cadre de la vérification distribuée avec plusieurs travailleurs sur des machines distants, un seul fichier n'est plus suffisant.
Pour traiter cette situation, CADP fournit le format PBG (*Partitioned BCG Graph*), qui implante le concept théorique de STE partitionné [11] et permet un accès uniforme à un tel STE dont les fragments peuvent être répartis sur plusieurs machines. Un fichier PBG regroupe un ensemble de fichiers BCG, appelés *fragments*, qui forment une partition du STE. De plus, un fichier PBG contient des informations sur la configuration du réseau utilisée lors de sa création. Le format PBG est un résultat de l'équipe associée SENVA[2] (commune entre Inria et le CWI).

[2] `http://vasy.inria.fr/senva`

protocole	nombre de processus	taille d'un état (en octets)	STE originel		STE minimisé	
			états	transitions	états	transitions
Burns&Lynch	4	43	769 244	1 367 318	3 023	11 244
Anderson	4	32	1 986 060	3 738 240	320	848
MCS	4	56	4 293 881	7 719 512	320	848
Peterson_tree	4	102	7 205 545	12 692 584	2 361	8 352
Szymanski	4	60	9 243 653	18 859 330	3 090	10 356
Knuth	4	48	16 642 361	32 614 282	6 721	27 281
CLH	4	48	18 317 849	31 849 616	320	848
Lamport	4	62	78 535 973	154 003 176	29 719	99 850
Peterson	4	49	214 175 671	389 640 061	6 460	21 347
Dijkstra	4	57	289 120 985	542 886 005	41 513	163 538

TAB. 1 – Exemples utilisés pour les expériences

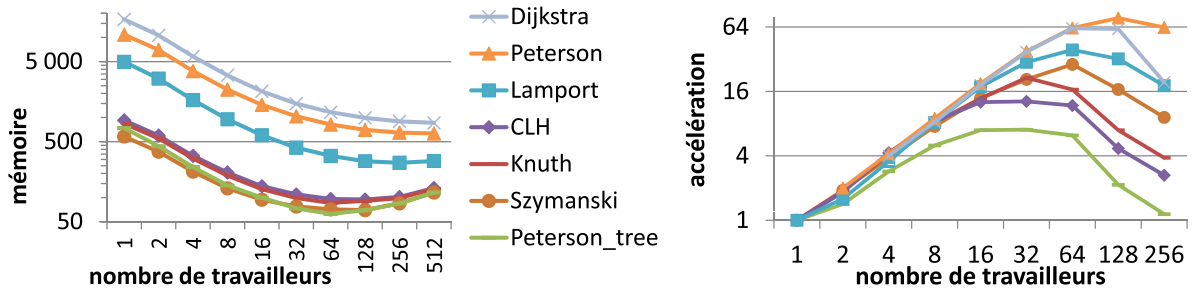
3. Outils pour la création et la manipulation des STE partitionnés

L'outil DISTRIBUTOR [11, 10] construit un STE partitionné, en utilisant un ensemble de travailleurs répartis sur un ensemble de machines décrit dans un fichier GCF. Chaque travailleur explore un fragment du STE et le stocke dans un fichier BCG ; une fonction de hachage statique détermine quel état est exploré par quel travailleur. Chaque fragment stocke les transitions entrantes (c'est-à-dire les transitions dont l'état cible est exploré par le travailleur), facilitant la gestion des états découverts indépendamment par plusieurs travailleurs.
Les outils PBG_CP, PBG_MV et PBG_RM effectuent la copie, le renommage et la suppression d'un STE partitionné, en gardant la cohérence entre les différents fichiers répartis sur plusieurs machines.
PBG_MERGE (appelé précédemment BCG_MERGE) construit un STE ordinaire à partir d'un STE partitionné, en renumérotant les états afin d'obtenir un fichier BCG plus compact.
L'outil prototype PBG_INVERT inverse les transitions d'un STE partitionné en générant un autre STE partitionné où chaque fragment stocke les transitions sortantes.
PBG_OPEN permet la vérification à la volée d'un STE partitionné. L'outil fournit une implantation de l'interface de programmation OPEN/CÆSAR [8], ce qui permet l'analyse du STE partitionné avec les outils de vérification à la volée de CADP, comme par exemple la réduction par ordres partiels (suppression d'entrelacements redondants) ou la vérification de formules de logique temporelle. PBG_OPEN est une application distribuée comportant un *maître* et un ensemble d'*agents*, chaque agent gérant un fragment du STE partitionné. Lorsque l'application de vérification demande l'énumération des successeurs d'un état au maître, celui-ci transmet la requête aux agents et fournit les réponses des agents à l'application. Afin de réduire le nombre de messages dans le cas d'une application parcourant plusieurs fois les successeurs d'un même état, PBG_OPEN peut utiliser un cache pour les listes des transitions successeurs.

4. Expériences

Nous avons expérimenté ces outils sur une sélection d'exemples de STE issus d'une étude de cas sur la vérification et l'évaluation de performance de protocoles d'exclusion mutuelle [13]. La table 1 récapitule quelques caractéristiques de ces exemples.
Nous avons utilisé plusieurs grappes de la plate-forme expérimentale Grid'5000 [7][3], notam-

[3] http://www.grid5000.fr



(a) mémoire maximale d'un processus (en Mo) (b) accélération (grappe genepi)

FIG. 1 – Génération distribuée d'un STE (axes avec échelle logarithmique)

ment les grappes à Grenoble et au Luxembourg, en utilisant le réseau de communication par défaut (Gigabit Ethernet). Dans la mesure du possible, nous avons utilisé un nœud séparé pour le maître et dédié un cœur à chaque travailleur. Nous avons utilisé l'outil `memtime`[4] pour mesurer le temps d'exécution et la consommation mémoire. Concernant la consommation mémoire, nous avons mesuré la consommation maximale pour chaque processus (maître et travailleurs) et nous indiquerons sur les courbes la valeur maximale de ces mesures. Concernant le temps d'exécution, nos mesures incluent les opérations nécessaires pour l'initialisation (la mise en place des répertoires de travail, les copies de fichiers, etc.) ; chaque valeur indiquée dans les courbes est obtenue par une moyenne après exclusion des valeurs extrêmes à hauteur de 20%.

4.1. Génération de STE partitionné

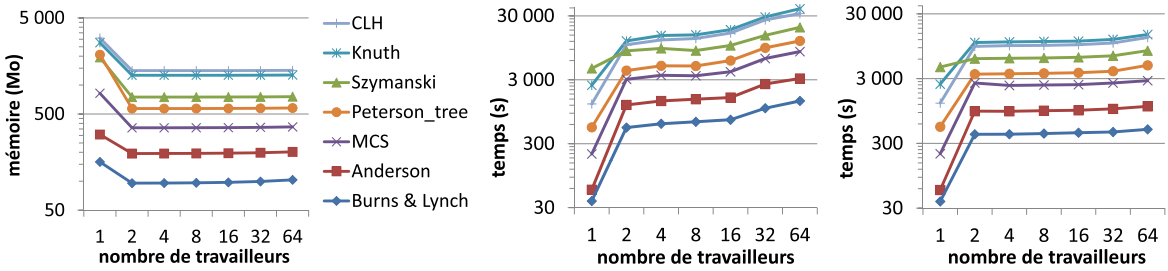
Nous avons généré plusieurs STE au moyen de DISTRIBUTOR. Les résultats de nos expériences avec jusqu'à 512 travailleurs sont illustrés dans la figure 1.

La figure 1(a) montre qu'augmenter le nombre de travailleurs diminue la quantité maximale de mémoire utilisée par un seul travailleur. Ainsi, l'utilisation d'un nombre important de travailleurs peut permettre de générer de grands STE même sur une grappe dont chacun des nœuds ne dispose pas d'une quantité de mémoire très importante (par exemple la grappe genepi avec 8 Go de RAM par nœud). Néanmoins, pour chaque exemple, on constate l'existence d'un nombre de travailleurs optimal : pour un nombre plus important de travailleurs, la mémoire maximale augmente à cause du surcoût de la bibliothèque de communication. On observe que plus le STE est petit, plus le nombre optimal de travailleurs est petit.

La figure 1(b) montre l'accélération observée en exécutant DISTRIBUTOR sur la grappe genepi seulement. La référence est l'exécution séquentielle sur un nœud de genepi, sauf pour les exemples Dijkstra et Peterson qui demandent plus de mémoire ; pour ces derniers, nous utilisons comme référence l'exécution avec le plus petit nombre de travailleurs possible. On observe que l'accélération est presque linéaire tant que le nombre de travailleurs est en dessous d'un seuil dépendant de l'exemple (et différent du nombre optimal de travailleurs mentionné ci-dessus concernant la consommation de mémoire) ; pour un nombre plus grand de travailleurs, l'accélération décroît fortement. Ceci peut s'expliquer par l'augmentation du coût de l'initialisation qui dépend du nombre de travailleurs : par exemple, la génération d'un STE avec un seul état et aucune transition prend 1s en séquentiel, moins de 2s pour deux travailleurs, mais plus de 90s pour 256 travailleurs.

Pour certains exemples, on observe une accélération super-linéaire : pour Dijkstra et Peter-

[4] Cet outil peut être téléchargé à l'adresse `http://www.update.uu.se/~johanb/memtime/`



(a) mémoire maximale (b) sans inversion (c) avec inversion

FIG. 2 – Réduction à la volée ; consommation de mémoire maximale de toute la chaîne d'outils : DISTRIBUTOR, PBG_INVERT et PBG_OPEN/REDUCTOR (axes avec échelle logarithmique)

son, elle s'explique par l'utilisation comme référence d'une exécution de DISTRIBUTOR (au lieu d'une exécution séquentielle) ; pour CLH, Knuth et Lamport, elle pourrait s'expliquer par les réallocations de la table de hachage pour les états si le nombre d'états dépasse certains seuils.

4.2. Réduction à la volée

Pour évaluer la performance de PBG_OPEN nous avons choisi une application d'analyse à la volée nécessitant beaucoup de ressources, à savoir REDUCTOR, qui réduit le STE selon la confluence des transitions internes (τ-*confluence*) en utilisant un algorithme séquentiel. Dans chacun de ces tests, REDUCTOR doit parcourir à la volée le STE complet, y compris les très grandes composantes fortement connexes de transitions internes contenus dans les STEs (ceci est indiqué par les très petites tailles des STE minimisés pour la bisimulation de branchement données dans les deux dernières colonnes de la table 1). La figure 2 résume ces expériences utilisant jusqu'à 64 travailleurs sur la grappe granduc sans utiliser les caches de PBG_OPEN.

La figure 2(a) montre que la consommation mémoire maximale d'un travailleur est réduite par rapport à une exécution séquentielle. Ceci s'explique par le fait que PBG_OPEN utilise 8 octets pour représenter un état du STE, ce qui est bien moins que les tailles des états données en table 1. Les courbes sont presque constantes pour n'importe quel nombre de travailleurs strictement plus grand que 1 car c'est le plus de mémoire est consommé par le maître exécutant l'algorithme séquentiel de REDUCTOR. Comme pour la génération distribuée, on observe également une augmentation de la consommation mémoire maximale si le nombre de travailleurs est trop important par rapport à la taille du STE.

En contrepartie de cette diminution de la consommation mémoire maximale, on observe une augmentation significative du temps d'exécution[5], due au latences de communication lors du parcours des transitions sortantes d'un état. Ceci nous a amené à étudier des techniques pour réduire le temps d'exécution du calcul des transitions sortantes d'un état.

4.3. Inversion des transitions

Pour rendre plus rapide le parcours des transitions sortantes d'un état, nous avons développé l'outil PBG_INVERT, qui transforme un STE partitionné pour ne pas stocker les transitions entrantes dans chaque fragment mais plutôt les transitions sortantes. Ainsi, lorsqu'une application de vérification doit parcourir les transitions sortantes d'un état, il lui suffit de demander à un seul agent (celui en charge du fragment auquel appartient l'état) au lieu de tous les agents. Cela diminue le coût de la vérification (entre 11 et 30 %), mais ajoute un coût fixe pour l'inver-

[5] Ces temps d'exécution longs ont été la raison de nous limiter aux exemples de taille pas trop importante.

exemple	LNT_OPEN		BCG_OPEN		PBG_OPEN		PBG_INVERT	
	mém.	temps	mém.	temps	mém.	temps	mém.	temps
Burns&Lynch	159	43	91	24	96	541	96	406
Anderson	307	62	202	51	194	1 222	194	921
MCS	822	215	390	157	360	3 069	360	2 309
Peterson_tree	2 076	549	626	365	570	3 954	570	3 488
Szymanski	1 940	4 530	842	1 290	750	8 537	750	6 047
Knuth	2 764	2 470	1 428	1 642	1 264	12 174	1 264	10 824
CLH	3 087	1 267	1 585	1 060	1 415	10 615	1 415	9 444
Lamport	oom	oom	6 930	9 227	6 015	59 932	6 015	52 627

TAB. 2 – Comparaison de différentes techniques de réduction ; 2 travailleurs pour les outils distribués ; temps d'exécution en secondes, consommation mémoire maximale en Mo ; «oom» signifie un débordement de mémoire (*out of memory*)

sion des transitions dans le STE partitionné. Notons que ceci n'a pas d'impact sur la quantité de mémoire utilisée, car PBG_INVERT consomme moins de mémoire que DISTRIBUTOR. La figure 2(c) montre les résultats de ces expériences : comparé au temps d'exécution sans inversion des transitions, le temps d'exécution est plus court, même pour 2 travailleurs. De plus, le temps d'exécution augmente beaucoup plus faiblement avec le nombre de travailleurs comparé au cas sans inversion.

4.4. Comparaison des différentes combinaisons d'outils pour la réduction à la volée

Avec la boîte à outils CADP, plusieurs combinaisons d'outils sont possibles pour produire le STE réduit à partir d'une description en LNT (le langage utilisé dans CADP pour décrire les programmes parallèles à vérifier). La table 2 compare les performances des combinaisons d'outils suivantes, qui comportent une exécution de REDUCTOR :

– exécution séquentielle sur la description en LNT (en utilisant LNT_OPEN) ;
– génération distribuée, suivie par la fusion dans un STE unique et une exécution séquentielle sur le BCG obtenu (en utilisant BCG_OPEN) ;
– génération distribuée et exécution sur le STE partitionné (en utilisant PBG_OPEN) ;
– génération distribuée, suivie par l'inversion du stockage des transitions (avec PBG_INVERT) et l'exécution sur le STE partitionné obtenu.

Ces résultats confirment les affirmations de l'introduction : la vérification à la volée d'un STE partitionné consomme le moins de mémoire et peut rendre ainsi la vérification faisable.

4.5. Mise en cache des listes de successeurs

Le temps d'exécution peut être réduit en utilisant un cache pour les transitions sortantes des états : ainsi, si la liste des transitions sortantes d'un état se trouve dans le cache, il n'est plus nécessaire d'interroger les agents gérant les fragments du STE partitionné. La figure 3 montre l'effet de l'augmentation du cache de PBG_OPEN jusqu'à un million d'éléments pour le cas de 4 travailleurs (les tendances pour d'autres nombres de travailleurs sont similaires). Ces courbes représentent les moyennes entre 2 et 12 exécutions avec un écart type maximal de 27 Mo (respectivement 56 s), qui reste toujours inférieur à 3% de la moyenne.

L'utilisation d'un cache permet de réduire le temps d'exécution, en augmentant légèrement la consommation mémoire. On observe les plus fortes réductions pour les exemples dont les composantes de transitions internes (que REDUCTOR parcourt plusieurs fois) peuvent tenir dans le cache. Pour un petit cache et de grandes composantes de transitions internes, on observe une

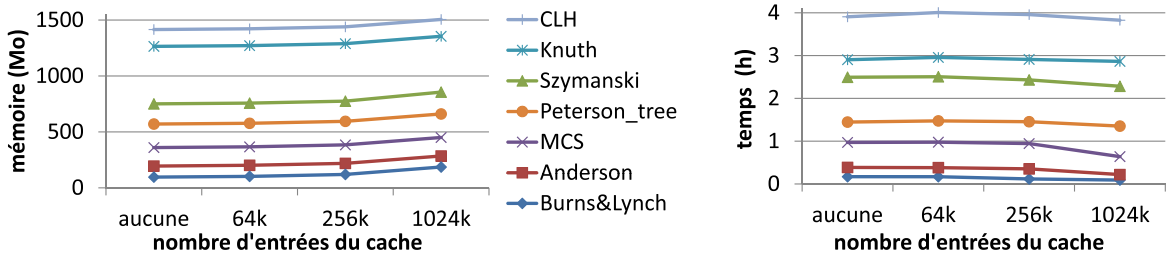


FIG. 3 – Utilisation d'un cache pour 4 travailleurs (PBG_OPEN/REDUCTOR seulement)

(légère) augmentation du temps d'exécution due aux opérations du cache (insertions qui remplacent des éléments déjà présents), qui contient rarement les transitions demandées.

5. Travaux voisins

D'autres équipes ont aussi utilisé des grappes et des grilles pour la vérification distribuée. Par exemple, l'outil DIVINE [3, 14] permet la vérification de propriétés de logique temporelle linéaire, mais dans un contexte basé sur les états, contrairement à CADP, qui est basé sur les actions et offre un large spectre de techniques de vérification (incluant la vérification de propriétés de logique temporelle, mais aussi la vérification d'équivalences).
L'outil LTSmin [6, 5] permet la génération et la minimisation distribuée d'un STE. Cependant, LTSmin ne permet pas la vérification à la volée d'un STE partitionné et, à notre connaissance, n'a pas encore été utilisé avec plusieurs centaines de travailleurs.
Un autre exemple est l'outil PREACH [4] qui permet l'analyse des états accessibles de modèles Murφ. PREACH utilise le code séquentiel de Murφ et implante les aspects de répartition et communication dans le langage fonctionnel Erlang, ce qui facilite l'expérimentation avec différents mécanismes de communication. Notons que le mécanisme de crédit pour le contrôle des flux, qui est crucial pour le passage à l'échelle de PREACH (étant donné les boîtes de messages non-bornées d'Erlang), n'est pas nécessaire pour la bibliothèque `caesar_network_1` de CADP, car ces aspects sont directement gérés par le protocole TCP/IP.

6. Conclusion

Nous avons présenté les outils récents de génération et de manipulation de STE partitionnés fournis par la boîte à outils CADP. Nos expériences avec la plate-forme Grid'5000, aussi bien que les travaux d'E. Madelaine avec PACAGrid [12, 2, 1], montrent que ces outils permettent d'exploiter les ressources des grappes et grilles de calcul, notamment la quantité mémoire, pour augmenter les capacités des outils de vérification de deux ou trois ordres de grandeur. En outre, l'outil prototype PBG_INVERT, qui inverse le stockage des transitions dans un STE partitionné, permet de réduire le surcoût de communication lors de la vérification à la volée.
Nous prévoyons de poursuivre nos travaux dans plusieurs directions. Tout d'abord, la bibliothèque de `caesar_network_1` de CADP devrait être optimisée pour supporter des transferts parallèles (en non séquentiels) lors de l'initialisation afin de réduire le surcoût observé pour un grand nombre de travailleurs. Aussi, la fonction de hachage statique pourrait être spécialisée afin d'améliorer la répartition des états sur les travailleurs en présence de machines hétérogènes. Enfin, on pourrait expérimenter la combinaison de PBG_OPEN avec un outil de vérification distribué, comme la version distribuée du model checker EVALUATOR 4.0.

Remerciements
Les expériences décrites dans cet article ont utilisé la plate-forme expérimentale Grid'5000 développée par Inria avec le support du CNRS, de RENATER, de plusieurs universités et d'autres sources de financement. Nous remercions aussi I. Bellicot, N. Descoubes, J. Fereyre, Y. Genevois et R. Hérilier pour leurs contributions au test et la correction d'erreurs des outils de vérification distribuée de CADP.

Bibliographie

1. Ameur-Boulifa (R.), Halalai (R.), Henrio (L.) et Madelaine (E.). – *Verifying Safety of Fault-Tolerant Distributed Components*. – Rapport de recherche RR-7717, INRIA, septembre 2011.
2. Ameur-Boulifa (R.), Henrio (L.) et Madelaine (E.). – Behavioural models for group communications. *In : Proc. of the Int. Workshop on Component and Service Interoperability, WICS'10*.
3. Barnat (J.), Brim (L.), Češka (M.) et Ročkai (P.). – DIVINE : Parallel distributed model checker. *In : Proc. of Parallel and Distributed Methods in Verification PDMC 2012*. pp. 4–7. – IEEE.
4. Bingham (B.), Bingham (J.), de Paula (F. M.), Erickson (J.), Singh (G.) et Reitblatt (M.). – Industrial Strength Distributed Explicit State Model Checking. *In : Proc. of the Int. Workshop on Parallel and Distributed Methods in Verification PDMC 2010*. pp. 28–36. – IEEE.
5. Blom (S.) et Orzan (S.). – Distributed state space minimization. *Software Tools for Technology Transfer*, vol. 7, n3, 2005, pp. 280–291.
6. Blom (S.), van de Pol (J.) et Weber (M.). – LTSmin : Distributed and symbolic reachability. *In : Proc. of the Int. Conf. on Computer Aided Verification CAV 2010*. pp. 354–359. – Springer.
7. Cappello (F.), Caron (E.), Daydé (M.), Desprez (F.), Jeannot (E.), Jegou (Y.), Lanteri (S.), Leduc (J.), Melab (N.), Mornet (G.), Namyst (R.), Primet (P.) et Richard (O.). – Grid'5000 : A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform. *In : Proc. of the Int. Workshop on Grid Computing GRID'2005*. pp. 99–106. – IEEE/ACM.
8. Garavel (H.). – OPEN/CÆSAR : An open software architecture for verification, simulation, and testing. *In : Proc. of the Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98*. pp. 68–84. – Berlin, mars 1998.
9. Garavel (H.), Lang (F.), Mateescu (R.) et Serwe (W.). – CADP 2011 : A toolbox for the construction and analysis of distributed processes. *Software Tools for Technology Transfer*, 2012.
10. Garavel (H.), Mateescu (R.), Bergamini (D.), Curic (A.), Descoubes (N.), Joubert (C.), Smarandache-Sturm (I.) et Stragier (G.). – DISTRIBUTOR and BCG_MERGE : Tools for distributed explicit state space generation. *In : Proc. of the Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2006*. pp. 445–449. – Springer-Verlag.
11. Garavel (H.), Mateescu (R.) et Smarandache (I.). – Parallel state space construction for model-checking. *In : Proc. of the SPIN Workshop on Model Checking of Software SPIN'2001*. pp. 217–234. – Berlin, mai 2001.
12. Henrio (L.) et Madelaine (E.). – *Experiments with distributed Model-Checking of group-based applications*. – Rapport technique, INRIA Sophia-Antipolis. Presented at the Sophia-Antipolis Formal Analysis Group 2010 Workshop SAFA2010, octobre 2010.
13. Mateescu (R.) et Serwe (W.). – Model Checking and Performance Evaluation with CADP Illustrated on Shared-Memory Mutual Exclusion Protocols. *Science of Computer Programming*, 2012.
14. Verstoep (K.), Bal (H. E.), Barnat (J.) et Brim (L.). – Efficient Large-Scale Model Checking. *In : Proc. of the Int. Symp. on Parallel and Distributed Processing IPDPS 2009*, pp. 1–12.