Table des matières

Table des figures

Liste des tableaux

Articles et communications

Articles

Allal, L. , Belalem, G., Dhaussy, P. and C. Teodorov. A parallel algorithm for the state space exploration. Scalable Computing : Practice and Experience, Vol. 17, No. 02, pp. 129-142, 2016.

Allal, L. , Belalem, G., Dhaussy, P. and C Teodorov. Sequential and parallel algorithms for the state space exploration. Cybernetics and Information Technologies, Vol. 16, No. 01, pp. 3-18, 2016.

Allal, L. , Belalem, G., Dhaussy, P. and C Teodorov. Distributed algorithm to fight the state explosion problem, in press.

Communications

Allal L. , Belalem G., Dhaussy P. (2016) Towards Distributed Solution to the State Explosion Problem. In : Satapathy S., Mandal J., Udgata S., Bhateja V. (eds) Information Systems Design and Intelligent Applications. Advances in Intelligent Systems and Computing, vol 433. Springer, New Delhi.

Allal L. , Belalem G., Dhaussy P., Teodorov C. (2016) Proposed Algorithms to the State Explosion Problem. In : Unal A., Nayak M., Mishra D., Singh D., Joshi A. (eds) Smart Trends in Information Technology and Computer Communications. SmartCom 2016. Communications in Computer and Information Science, vol 628. Springer, Singapore.

Allal L. , Belalem G., Dhaussy P, Teodorov C. (2017) Using parallel and distributed Reachability in model checking. 2^{nd} International Conference on Computer, Communication and Computational Sciences (RACCCS-2017). Aryabhatta College of Engineering & Research Center, Ajmer, India.

Chapitre 1

Introduction générale

Courage, espoir et confiance en soi sont les prémisses puissantes d'un fondement véhiculaire de la voie de la réussite.

Mofaddel Abderrahim

1.1 Contexte de travail

Toute personne ayant déjà manipulé des logiciels informatiques a déjà rencontré des bugs qui sont en général causés par une erreur non identifiée lors de la conception du logiciel.

Avec l'augmentation inévitable de la complexité des systèmes matériels et logiciels, la probabilité d'erreurs subtiles est élevée. De telles erreurs peuvent avoir des conséquences catastrophiques en termes d'argent, de temps ou même de vie humaine. En général, plus une erreur est corrigée tôt, moins elle coûte cher. Dans l'industrie, il existe donc une demande croissante de méthodologies qui peuvent accroître la confiance dans la conception et la construction du système. De telles méthodologies se traduiront par une amélioration de la qualité, ainsi que dans une réduction du coût de développement total d'un système. De plus, sur le plan théorique, il est nécessaire de fournir une base mathématique pour la conception de systèmes informatiques.

La part croissante que l'informatique occupe dans des domaines critiques comme les télécommunications, l'aviation, le nucléaire ou la médecine doit garantir de manière absolue l'absence de toute défaillances ou bugs dans certains logiciels. C'est ce que proposent les méthodes formelles.

L'approche traditionnelle de l'ingénierie pour la construction de systèmes complexes consiste à construire des modèles. Les modèles peuvent être étudiés et modifiés jusqu'à ce que la confiance soit obtenue dans leur exactitude. L'avantage est que les modèles sont plus simples, représentent les aspects particuliers du système, et leur coût de développement est négligeable par rapport au coût de construction du système lui-même. La vérification formelle signifie créer un modèle mathématique d'un système, en utilisant un langage pour spécifier les propriétés souhaitées du système d'une manière concise et sans ambiguïté, et en utilisant une méthode de preuve pour vérifier que les propriétés spécifiées sont satisfaites par le modèle. Lorsque la méthode est réalisée essentiellement par machine, nous parlons d'une vérification automatique. Une méthode bien définie pour la vérification est le *Model checking*.

La technique du *Model checking*, pour la vérification formelle consiste à prendre en considération un modèle mathématique du système et à vérifier la validité d'une formule de logique temporelle dans le modèle. Cette technique de vérification de modèle ne détermine pas seulement la validité d'une propriété exprimée en logique temporelle mais, dans le cas d'échecs, trouve également des séquences d'exécution qui la violent. Un ingénieur peut utiliser une telle séquence pour trouver et réparer une erreur dans le système. Une première difficulté liée à l'utilisation de ces techniques de vérification provient du problème bien identifié de l'explosion combinatoire du nombre de comportements des modèles, induite par la complexité interne du logiciel qui doit être vérifié. Cela est particulièrement vrai dans le cas des systèmes embarqués temps réel, qui interagissent avec des environnements impliquant un grand nombre d'entités. Beaucoup de travaux en méthodes formelles ont été menés afin de maîtriser l'explosion combinatoire [21, 24, 48]. Des techniques développées autour du *Model checking* consistent en général à diminuer l'espace d'états du

système global de telle manière à pouvoir effectuer une recherche d'accessibilité d'états. Toutes ces approches se basent sur l'étude de méthodes et structures de données différentes pour traiter l'explosion en espace mémoire. Dans cette thèse, nous nous sommes intéressés au temps d'exécution nécessaire pour effectuer une exploration d'un modèle à la recherche d'éventuels erreurs. Une exécution qui prend beaucoup de temps peut amener à une explosion en temps d'exécution. Dans cette recherche, nous avons étudié la partie exploration du modèle sans vérifier les propriétés sur le modèle.

1.2 Solutions proposées

L'objectif de cette recherche est de proposer un algorithme d'exploration permettant d'avoir un gain en temps, car lors de la conception d'un algorithme, on étudie sa complexité pour évaluer son efficacité. Pour cela, on se base sur la complexité temporelle. Après cette phase, nous pouvons comparer l'algorithme conçu avec un ou plusieurs algorithmes comparant leur complexités respectives.
nous avons proposé 3 algorithmes, 2 parallèles et un distribué. La première solution parallèle se base sur l'utilisation d'une interface parallèle permettant la gestion des *threads* en divisant le traitement en plusieurs tâches qui seront affectées aux *threads* . A la fin de l'éxécution, tous les résultats sont récupérés. Cette solution permet la soumission de tâches et leur exécution, et offre une gestion de pools de threads.
Dans la deuxième solution, nous avons proposé un algorithme qui s'inspire de l'algorithme d'exploration parallèle présenté dans [38]. Nous avons proposé une optimisation à cette solution, en apportant un gain en temps et en espace mémoire et cela, en effectuant moins d'allocation mémoire. Concernant l'approche distribuée, nous avons effectué une comparaison entre une exécution utilisant deux machines physiques et une exécution manipulant deux machines virtuelles dans un environnement de cloud.

1.3 Expérimentations et évaluation des résultats

Afin de réaliser les expériences et d'évaluer le résultat ainsi obtenu, nous avons utilisé 3 exemples d'exclusion mutuelle. Pour cela, nous avons spécifié leurs algorithmes respectifs sous *Java* . Nous avons effectué les expériences en comparant nos approches parallèles avec une approche parallèle présentée dans le chapitre *Etat de l'art* en se basant sur certains paramètres. Afin de traiter l'exploration distribuée, nous avons réalisé deux expériences, la première pour comparer entre une exécution sur deux machines "physiques" et "virtuelles" dans un cloud computing.
Dans la deuxième expérience, nous avons réalisé une étude comparative entre une exécution parallèle et une exécution distribuée.

1.4 Organisation de la thèse

Le **Chapitre 2** présente le principe de vérification par *Model checking*. L'objectif de cette technique est la vérification de propriétés d'un système. Ces propriétés sont généralement spécifiées par des formules logiques, permettent d'exprimer des propriétés sur les états/transitions passés, présents ou futurs qu'un système peut atteindre.

Le **Chapitre 3** présente un état de l'art sur quelques solutions proposées pour traiter le problème d'explosion combinatoire.

Le **Chapitre 4** décrit les contributions réalisées durant cette thèse.

Le **Chapitre 5** présente les expériences réalisées en effectuant des comparaisons entre quelques algorithmes sur le temps d'exécution. Des résultats expérimentaux encourageants sont présentés.

Chapitre 2

Model checking

Tout obstacle renforce la détermination. Celui qui s'est fixé un but n'en change pas.

Lénonard De Vinci

Sommaire

2.1 Introduction

Dans la conception de logiciels et de matériels de systèmes complexes, plus de temps et d'efforts sont consacrés à la vérification qu'à la construction. Les techniques sont recherchées pour réduire et faciliter les efforts de vérification tout en augmentant leur couverture. Les méthodes formelles offrent un grand potentiel pour obtenir une intégration précoce de la vérification dans le processus de conception, pour fournir des techniques de vérification plus efficaces et pour réduire le temps de vérification. Ceux sont des méthodes qui utilisent des techniques de conception des systèmes à base de modèles mathématiques rigoureusement spécifiés pour construire des systèmes logiciels et matériels. Les techniques de vérification de modèles sont utilisées pour vérifier qu'un modèle possède certaines propriétés. Les propriétés à valider peuvent être tout à fait élémentaires, par exemple, un système ne doit jamais être en mesure de parvenir à une situation d'interblocage, et sont principalement obtenues à partir de la spécification du système. Cette spécification prescrit ce que le système doit faire et ne pas faire, et constitue la base de toute activité de vérification. Un rejet est rencontré une fois que le système ne remplit pas l'une des propriétés de la spécification. Le système est considéré comme correct, à chaque fois qu'il satisfait toutes les propriétés obtenues à partir de sa spécification. Donc, l'exactitude est toujours relative à un cahier des charges, et ne constitue pas une propriété absolue d'un système. Les méthodes formelles sont des techniques fortement recommandées pour le développement des logiciels et en particulier pour les logiciels critiques. Elles ont conduit au développement de certaines techniques de vérification très prometteuses qui facilitent la détection précoce des défauts.

2.2 Vérification formelle

Notre dépendance aux applications informatiques (matériel et logiciel) a motivé les chercheurs à développer de nouvelles techniques pour accroître notre confiance sur leur exactitude. De telles applications vont de simples machines à café, aux centrales nucléaires, passant par des tours de contrôle de vol. Plusieurs applications sont incontestablement critiques et une faille peut causer des dommages élevés [27]. Juste pour mentionner une de ces grandes catastrophes, le 4 juin 1996, la fusée européenne Ariane a explosé quelques secondes après le lancement, ce qui a coûté des millions de dollars et de nombreuses années de travail perdu. Une enquête a révélé que l'erreur aurait pu être évitée grâce à des méthodes formelles pour détecter cette erreur.

Comment peut-on être sûr de l'exactitude des systèmes critiques, avec des milliers (et parfois des millions) de composants interagissant de manière complexe. Dans certains cas, il est possible de construire le système considéré, par exemple la puce réelle, ou la machine à café, puis "alimenter" le système avec des données sensibles pour vérifier qu'il se comporte réellement comme il est supposé. C'est le principe du test. Une telle approche est largement utilisée et extrêmement

utile dans la pratique, bien qu'il ne soit évidemment pas possible de l'utiliser dans des systèmes très critiques, si les données de test pourraient causer des dommages en cas d'erreur. Une autre solution consiste à simuler le comportement du système sur un ordinateur. La simulation ne fonctionne pas directement sur le système réel, mais sur un modèle. Un modèle est une représentation abstraite du système réel, généralement écrit en utilisant les mathématiques ou la logique. Parmi les méthodes de vérification formelle, il y a aussi la technique du *Model checking* qui consiste à explorer tous les états possibles du système. De cette manière, on peut montrer qu'un modèle d'un système donné satisfait ou pas une propriété donnée.
Parmi les techniques de vérification formelle, il y a les tests, la simulation et la vérification par *Model checking* [13].

2.2.1 Simulation

La simulation est basée sur un modèle qui décrit le comportement possible de la conception du système à portée de main. Ce modèle est exécutable dans un certain sens, de sorte qu'un outil logiciel (appelé simulateur) puisse déterminer le comportement du système sur la base de certains scénarios. De cette façon, l'utilisateur obtient un aperçu des réactions du système sur une certaine stimulation. Les scénarios peuvent être fournis par l'utilisateur, ou par un outil comme un générateur aléatoire qui fournit des scénarios aléatoires. La simulation est généralement utile pour une évaluation rapide de la qualité d'une conception. Il est impossible de simuler tous les scénarios représentatifs [33].

2.2.2 Test

Une technique largement appliquée pour valider l'exactitude d'une conception est au moyen de tests. En utilisant les tests, on suppose que l'implémentation du système est réelle. Avec cette technique, on vérifie si la réaction du système est conforme à sa spécification. Le principe du test est presque identique à celui de la simulation, la distinction importante étant que le test est effectué sur une implémentation du système d'exécution réel, tandis que la simulation est basée sur un modèle du système [63]. Le test est une technique de validation qui est la plus utilisée en pratique dans le domaine de l'ingénierie logicielle. Comme le test est basé sur l'observation d'un petit sous ensemble de toutes les instances possibles du comportement du système, il ne peut jamais être complet. Puisque les tests peuvent être appliqués sur une implémentation réelle, cette technique est nécessaire dans le cas où il est difficile de construire un modèle valide du système en raison de sa complexité [13].
Les tests et la simulation sont très répandus dans les applications industrielles et leurs utilisations ont été très utiles. Un inconvénient, cependant, est qu'il n'est pas possible, en général, de simuler

ou de tester tous les scénarios ou comportements possibles d'un système donné. C'est-à-dire que ces techniques ne sont généralement pas exhaustives, en raison du nombre élevé de cas possibles à prendre en compte (parfois infini en théorie) et les cas d'erreurs peuvent être parmi les cas qui ne sont pas testés ou simulés. Dans cette thèse, nous nous intéressons à la troisième technique qui est le *Model checking* [13].

2.3 Model checking

"Model checking is a technique for verifying finite state concurrent systems" Edmund M. Clarke et al. 1999 [22].

Le *Model checking* est une technique de vérification basée sur l'exploration exhaustive de toutes les configurations, ou états, du système à la recherche de comportements non conformes à la spécification [15, 20, 23]. Ces comportements sont spécifiés à l'aide de propriétés exprimées sur le modèle. Un *Model checker* représente l'outil de vérification des modèles, il peut être vu comme une boîte noire qui reçoit en entrée, un système, ainsi qu'une proprieté exprimée sur ce système et produit en retour, une réponse, indiquant si la propriété est vérifiée ou non [5]. La mise en œuvre des algorithmes, incluent généralement une construction de l'espace d'états du système, puis un parcours de cet espace à la recherche d'erreurs. Cet espace d'états est un graphe dirigé qui décrit toutes les évolutions possibles du système. Les nœuds de ce graphe sont les états du système et les arcs représentent les transitions entre ces états. Ainsi, si le système étudié est un programme, un nœud correspondra à un état de la mémoire (valeurs des variables), et les transitions seront les instructions du programme. Le *Model checking* est une technique de vérification qui explore tous les états possibles du système. La plupart des explorateurs, ne prennent en compte que les états du système, sans prendre en compte l'environnement extèrieur. Une solution décrite dans [60], permet de prendre en compte la spécification explicite de l'environnement dans lequel le modèle du système est plongé lors de son exploration. Un état décrit le comportement d'un système à un moment donné. Ce processus peut être comparé à un programme d'échecs qui vérifie les mouvements possibles, un *Model checker*, l'outil logiciel qui effectue la vérification du modèle, examine tous les scénarios possibles du système d'une manière systématique. De cette manière, on peut montrer qu'un modèle de système donné, satisfait ou pas une certaine propriété.

Les propriétés qui peuvent être vérifiées en utilisant un model checker sont de nature qualitative, exemple, *le résultat généré est-il correct ? Le système peut-il atteindre une situation d'interblocage ?* dans le cas de l'exclusion mutuelle. Le model checker examine tous les états du système, pour vérifier si le modèle satisfait la propriété désirée. Si une propriété n'est pas vérifiée, le *Model checker* fournit un contre-exemple indiquant à l'utilisateur le moyen d'atteindre l'état indésirable [13]. Parmi les propriétés à vérifier sur le modèle du système, on a :

- La sûreté : un mauvais comportement ne se produit jamais ;

– La vivacité : un comportement attendu finira par arriver.

La *Figure. 2.1* illustre le processus de vérification du modèle. Comme condition préalable, il faut un modèle d'un système et la spécification à vérifier. Considérant les contraintes de temps et de mémoire, la première est souvent une abstraction du système à vérifier. La spécification indique les propriétés que le système doit satisfaire. Cela est souvent spécifié à l'aide de la logique temporelle. Le *Model checker* vérifie si les propriétés indiquées dans la spécification sont satisfaites pour tous les chemins d'exécution possibles du système. Dans le cas où la spécification n'est pas satisfaite par le système, le *Model checker* renvoie une trace d'erreur.

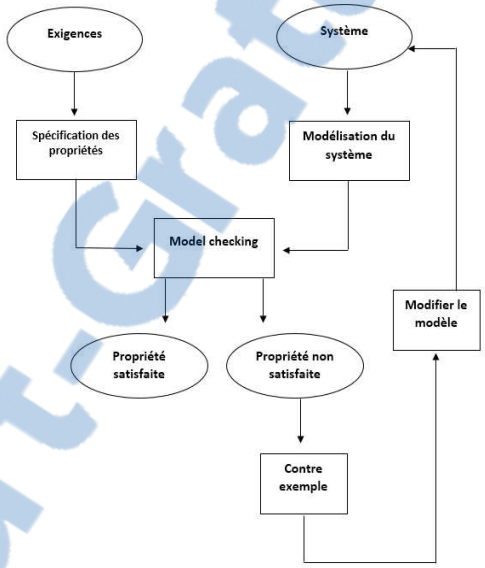

FIG. 2.1 – Structure du Model checking [13]

2.4 Domaines d'application du Model checking

Depuis qu'il a été proposé pour la première fois par Edmund M. Clarke et Allen Emerson dans [21] et Joseph Sifakis et Jean-Pierre Queille dans [55], la vérification des modèles a été utilisée avec succès dans la pratique pour vérifier des applications telles que les circuits séquentiels complexes et les protocoles de communication. Ainsi, pour ne citer qu'un exemple, 5 erreurs non détectées au préalable furent découvertes par cette approche dans un module de contrôleur du véhicule spatial Deep Space 1 de la NASA [41].

2.5 Logique temporelle

Les logiques temporelles [30 , 41] permettent d'exprimer des propriétés sur les états/transitions passés, présents ou futurs qu'un système peut/doit atteindre. A cet effet, elles offrent des opérateurs temporels spécifiques, des modalités, tels que finalement ou jamais, qui permettent de décrire des ordres entre les évènements/états sans pour autant introduire le temps explicitement. D'un point de vue plus théorique, on peut distinguer deux grandes familles de logiques temporelles. Ces deux familles, arborescentes et linéaires, sont distinguées par la nature de la relation d'ordre associée au temps que l'on considère. Cet ordre capture l'antériorité temporelle entre les états/événements et :

- Si l'ordre est total, le comportement du système est vu comme un ensemble de séquences d'exécution et la spécification se fait via les logiques temporelles linéaires. La logique LTL (Linear Time Logic) constitue un exemple de ces logiques.
- Si l'ordre est partiel, le comportement du système est vu comme un graphe et la spécification se fait via les logiques temporelles arborescentes. la logique CTL (Conditional/Computer Tree Logic) constitue un exemple de ces logiques.

Les pouvoirs d'expression des logiques arborescentes et linéaires sont incomparables. Les propriétés de sûreté et de vivacité sont expressibles tant en LTL qu'en CTL.

2.6 Avantages et inconvénients

Parmi les avantages du *Model checking* , nous citons la liste partielle suivantes [13 , 20 , 35] :

- C'est une technique qui supporte une vérification partielle, c'est-à-dire que les propriétés peuvent être vérifiées individuellement, permettant ainsi de se concentrer sur les propriétés essentielles dans un premier temps.
- La vérification par *Model checking* est rapide comparativement à d'autres techniques de vérification formelle, comme par exemple l'utilisation d'un vérificateur de preuves dont l'analyse peut nécessiter des mois de travail.
- Cette technique de vérification est applicable à grand nombre d'applications, notamment, pour la vérification des circuits électroniques, dans le domaine de l'aéronautique et dans les systèmes embarqués.
- Un contre-exemple est retourné quand la propriété n'est pas vérifiée. Si une des propriétés n'est pas satisfaite, le modèle checker va produire une trace d'exécution d'un contre-exemple qui montre pourquoi la spécification n'est pas vérifiée.
- Le processus de vérification est automatique. L'utilisateur du *Model checker* n'a pas besoin de construire une preuve de corrections. En principe, l'utilisateur doit entrer une description d'un circuit ou d'un programme et la spécification qui doit être vérifiée.

Parmi les inconvénients du *Model checking* [13 , 20] :

- Le problème majeur de la technique de vérification par model checking est l'explosion exponentielle du temps de calcul ou de la taille mémoire. La vérification d'un système critique peut amener à un temps indéterminé négligeant dans ce cas précis la propriété d'efficacité d'un algorithme. Dans le cas le l'espace mémoire, le nombre d'états nécessaires pour modéliser le système peut dépasser la quantité de mémoire disponible, ce qui conduit à un arrêt de l'exploration.
- Les résultats peuvent ne pas être corrects : comme tout outil, un vérificateur de modèle peut comporter des défauts logiciels.
- Il vérifie seulement la propriété déclarée c'est-à-dire qu'il n'y a aucune garantie de complétude. La validité des propriétés non vérifiées ne peut pas être jugée. Ce problème est associé à toute méthode de validation.

2.7 Processus d'exécution du Model checking

Le *Model checking* est une technique de vérification qui explore tous les états possibles du système. Un algorithme de model checking prend en entrée une abstraction du comportement du système réactif (un système de transitions) et une formule d'une certaine logique temporelle, et répond si l'abstraction satisfait ou non la formule.

Le processus du *Model checking* s'effectue en 3 phases [54] :

- **Phase de modélisation** : cette phase consiste à :
  - Modéliser le système dans le langage de spécification du *Model checker* ;
  - Formaliser les propriétés à vérifier en utilisant la logique temporelle.
- **Phase d'exécution** : exécuter le *Model checker* pour vérifier la validité de la propriété sur le modèle.
- **Phase d'analyse** : analyser les résultats de l'exécution :
  - Si une propriété est satisfaite, passer à la vérification des autres propriétés ;
  - Sinon, générer un contre exemple ;
  - L'exécution peut ne pas se terminer à cause d'un manque d'espace mémoire, dans ce cas, réduire la taille du modèle à vérifier puis relancer l'exécution.

Le processus de vérification est expliqué plus en détails ci dessous :

- **Phase de modélisation.** La première étape pour la vérification d'un système (tel un programme par exemple) va être de le modéliser à l'aide d'un langage utilisable par un *Model checker* (exemple : promela, réseaux de Pétri, automates). Afin d'effectuer la vérification, le *Model checker* a besoin du modèle du système à vérifier ainsi que d'une, ou d'un ensemble de propriétés spécifiées en logique temporelle.

  Les modèles des systèmes, décrivent le comportement des systèmes de manière précise et

sans ambiguïté. Ils sont principalement exprimés à l'aide d'automates à états finis, constitués d'un ensemble fini d'états et un ensemble de transitions. Les états représentent des informations sur les valeurs courantes des variables. Les transitions décrivent l'évolution du système d'un état à un autre.

Afin de réaliser une vérification correcte, les propriétés doivent être décrites de manière précise et sans ambiguïté. La logique temporelle permet la spécification de plusieurs propriétés telle que l'accessibilité (une certaine situation peut être atteinte), l'invariance (le système fait ce qu'il est supposé faire), la sûreté (quelque chose de mauvais n'arrive jamais) et la vivacité (quelque chose de bon finira par arriver).

- **Phase d'exécution.** Le *Model checker* va effectuer une exploration exhaustive de l'espace d'états. Il s'agit essentiellement d'une approche purement algorithmique, dans laquelle la validité de la propriété en cours d'examen est vérifiée dans tous les états du modèle du système. L'exploration débute toujours par l'état initial. Durant cette phase, tous les états explorés sont stockés en mémoire afin d'éviter de répéter l'exploration d'un état visité.
- **Phase d'analyse.** La dernière étape consiste à analyser si le modèle vérifie sa spécification. Trois résultats sont possibles : la propriété est valide sur le modèle ou pas, ou la vérification ne peut pas se terminer en indiquant un manque d'espace mémoire car le modèle à vérifier est large. Le *Model checker* prend en entrée un modèle et une spécification et renvoie vrai si la propriété est vérifiée et faux dans le cas contraire. Dans ce cas, le *Model checker* fournit un contre-exemple indiquant la façon dont le modèle pourrait atteindre l'état non désiré depuis l'état initial (trace d'erreurs). Cela implique une correction du modèle puis de relancer la traitement à nouveau. Si la propriété satisfait le modèle, le *Model checker* va à la suivante, s'il y'en a d'autres, en utilisant le même processus de vérification décrit ci-dessus. Si le modèle est trop grand, le temps d'exécution va augmenter de façon exponentielle ce qui amène à une complexité temporelle élevée.

2.8 Obstacle au Model checking : Explosion en temps ou en espace mémoire

L'avantage principal du *Model checking* est qu'il est automatique et rapide, l'inconvénient majeur du *Model checking* est apparu dans les années 80, et est connu sous le nom d'explosion combinatoire de l'espace d'états [25, 52, 56]. Seuls les systèmes avec un nombre petit d'états pouvaient être analysés à cette époque, alors que les systèmes réels en possèdent beaucoup plus voire une infinité d'états ce qui amène aussi à traiter la complexité temporelle en plus de la complexité spatiale. Par exemple lorsque les structures de données d'un programme sont infinies (buffers, files d'attente) et qu'il y a plusieurs instructions à exécuter. Aussi, dans les systèmes embarqués, la mise en œuvre de l'asynchronisme et de la concurrence augmente le problème

combinatoire *dû* aux entrelacements engendrés par les événements des processus concurrents. Autrement dit, le nombre d'états à explorer peut croître de façon exponentielle en fonction de la complexité du système, ce qui fait que le traitement des instructions est très ralenti, les temps d'accès aux données explosent et les taux de transfert de données s'effondrent.
Un certain nombre de méthodes ont été proposées pour éviter une explosion d'états, seulement, peu de recherches ont été menées dans le but de gérer la complexité temporelle de l'exploration qui correspond à l'objet principal de notre étude. Nous n'avons pas traité durant cette recherche l'espace mémoire nécéssaire pour l'exploration d'un modèle, nous avons traité l'aspect temporel. Dans ce qui suit, nous présentons les techniques propoposées pour la gestion de l'explosion spatiale. Elles se répartissent en quatre grandes catégories (voir *Figure. 2.2*)

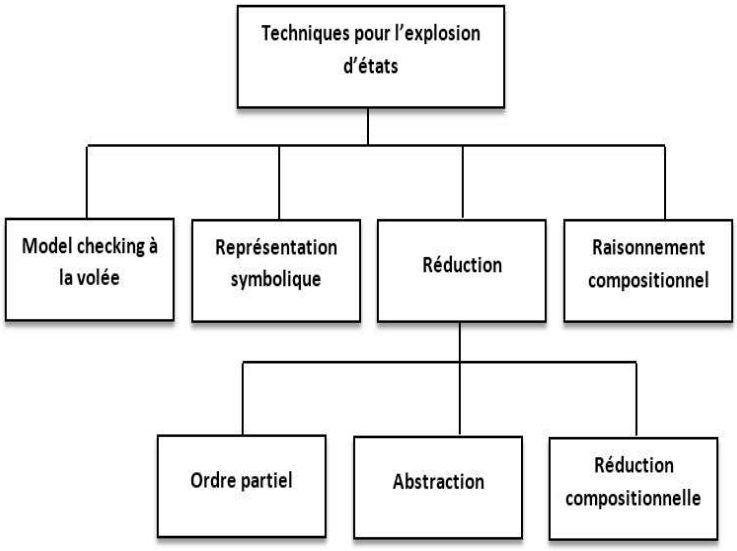

FIG. 2.2 – Approches pour éviter le problème d'explosion en espace mémoire [32]

2.8.1 Model checking à la volée

L'analyse de l'accessibilité d'un modèle est une technique de vérification qui effectue une exploration exhaustive de tous les états accessibles d'un système. Les techniques à la volée sont basées sur le fait que dans l'exécution de l'analyse d'accessibilité, il n'est pas nécessaire de stocker l'ensemble du graphe d'états du système global " *graphe d'états* ", pour éviter le problème d'explosion d'états. Au contraire, il suffit de simuler toutes les séquences de transitions possibles que le système est capable d'effectuer. Un parcours du graphe en profondeur peut être utilisé pour explorer le système «à la volée», c'est-à-dire sans stocker les transitions qui sont prises pendant le parcours. Cela réduit considérablement les besoins en espace mémoire. Pour une

recherche en profondeur du graphe, l'exigence de stockage minimal est celle du chemin courant exploré. Une telle recherche réduit les besoins en mémoire tout en garantissant une exploration exhaustive de l'espace d'états. Cependant, le temps nécessaire pour effectuer la vérification peut augmenter considérablement en raison de l'exploration des états déjà visités. Le parcours en profondeur du graphique peut être réalisé en stockant les états une fois qu'ils ont été visités. Cela réduit les exigences de temps au minimum, tout en exigeant le stockage de tous les états accessibles. Cependant, pour les graphes de grande taille, il peut être impossible de stocker tous les états. Différentes méthodes ont été proposées pour le stockage des états explorés en cache. En plus de stocker le chemin courant, la mise en cache de l'espace d'états crée un cache restreint d'états visités sélectionnés [37]. Initialement, tous les états visités sont stockés dans le cache, jusqu'à ce qu'il se remplisse. Lorsque cela se produit, les anciens états sont progressivement remplacés par de nouveaux [37]. L'efficacité de la mise en cache de l'espace d'états dépend de la taille du cache, mais aussi de la taille de l'espace d'états. Seulement, une explosion en temps est provoquée par des explorations multiples d'états non stockés de l'espace d'états. Un avantage de cette technique est que les propriétés sont vérifiés au moment de l'exploration des états. L'exploration est réalisée jusqu'à ce qu'une erreur soit détectée. Dans ce cas, un contre-exemple est généré pour aider le concepteur à corriger les erreurs. Souvent, les erreurs sont découvertes très tôt au cours de la recherche, évitant ainsi l'exploration entière de l'espace d'états. D'autre part, lorsque le système est correct, la recherche couvre l'ensemble de l'espace d'états. L'approche est donc particulièrement adaptée aux premiers stades de la conception, qui tendent à contenir de nombreuses erreurs [37]. Le temps de réponse est négligé dans cette technique, car plusieurs états visités peuvent être explorés plus d'une fois.

2.8.2 Représentation symbolique

L'approche symbolique apparue dans les travaux de Kenneth McMillan [47], se dintingue par les structures de données qu'elle utilise et les algorithmes qu'elle met en œuvre. La représentation symbolique est basée sur la représentation implicite d'un modèle à états finis d'un système [26 , 48]. Les méthodes symboliques évitent de modéliser le système par un système de transitions, au lieu de cela, ils représentent implicitement l'espace d'états du système en utilisant des fonctions et des formules booléennes. Cela se traduit dans l'implantation par l'utilisation de structures de données compactes comme les diagrammes de décision binaire (BDD) pour représenter l'ensemble des états [35 , 62 , 65]. Un BDD est une forme d'arbre binaire qui permet de représenter efficacement des fonctions booléennes, en permettant un partage fort entre les nœuds de l'arbre. Tout état d'un système est représenté par un vecteur de bits contenant des valeurs sur les variables du système à vérifier. Soit s un état d'un système, $s = x_0, x_1, x_2, ..., x_n$, o $\hat{u}$ x_i est le i^{me} bit dans le vecteur d'états. Les algorithmes symboliques de construction et d'exploration de l'espace d'états

sont fondamentalement différents des algorithmes explicites. La différence majeure réside dans le fait que les structures de données symboliques permettent de traiter simultanément non pas un état mais un ensemble d'états. L'algorithme d'une exploration utilisant le *Model checking* symbolique est le suivant (*Algorithm 1*) :

Algorithm 1 Algorithme d'exploration en largeur

1: toSee $\leftarrow S_0$
2: $Reached \leftarrow S_0$
3: **while** $\neg(toSee.isEmpty())$ **do**
4: toSee $\leftarrow$ toSee.genererSucesseurs()
5: Reached $\leftarrow$ Reached $\cup$ toSee
6: **end while**

A chaque étape, l'algorithme peut calculer l'ensemble des nouveaux états accessibles à partir de l'ensemble $toSee$ en une seule opération (ligne 4). De même, l'ajout des nouveaux états à l'espace d'états $Reached$ est implanté par la disjonction de deux diagrammes de décision (ligne 5). Les algorithmes symboliques procèdent implicitement par un parcours en largeur. L'ensemble $toSee$ contient au départ l'état initial S_0. A la première itération, il contient tous les successeurs de l'etat initial (les états se trouvant à la profondeur 1). A la seconde itération, il contient tous les successeurs des états à la profondeur 2, et ainsi de suite.

La représentation symbolique est réalisée de façon transparente à l'utilisateur par un outil (*Model checker*) supportant la vérification symbolique. Les propriétés sont vérifiées directement sur le diagramme BDD. Cette technique permet de stocker les états implicitement, cela veut dire que les états sont stockés sous une forme compressée, car le diagramme de décision binaire est réduit suivant des règles de réduction. Donc, pour vérifier si un état à été exploré, tous les états traités doivent être décompréssés, ce qui augmente considérablement le temps d'exploration d'un modèle donné.

2.8.3 Réduction

Les techniques de réduction se basent sur la construction d'une partie d'une abstraction de l'espace d'états d'un système, tout en conservant la capacité de vérifier les propriétés. Dans ce qui suit, nous décrivons les principales approches de la réduction de l'espace d'états.

Ordre partiel : Dans la plupart des méthodes de vérification de modèles, la concurrence est caractérisée par l'entrelacement des évenements, qui est un facteur majeur favorisant l'explosion d'états. La réduction par ordre partiel est basée sur l'observation que dans les systèmes concurrents, l'effet total d'un ensemble d'actions est souvent indépendant de l'ordre dans lequel ces actions se produisent. En conséquence, on peut éviter toute génération inutile

de tous les entrelacements possibles entre de telles actions. Dans les méthodes d'exploration classiques, pour chaque état atteint, toutes les transitions générées sont traitées. Les méthodes de réduction par ordre partiel quant à elles effectuent une recherche sélective de l'espace d'états du système [31]. Pour chaque état s exploré au cours de la recherche, un sous ensemble t de l'ensemble des transitions sortantes de s sont traités. Deux techniques principales ont été proposées dans la littérature pour identifier ces sous ensembles, ils sont basés sur le calcul de deux ensembles $persistent$ et $sleep$ [64]. Dans la première technique, toute transition qui est accessible à partir d'un état s en testant exclusivement des transitions qui n'appartiennent pas à l'ensemble $persistent\ t$ est indépendante des transitions de l'ensemble t. Il a été prouvé dans [61], que l'exécution de toutes les transitions restantes peut être retardée sans affecter les résultats de la vérification.

La deuxième technique consiste à réduire le nombre de transitions explorées, mais pas le nombre d'états. Chaque état s possède un ensemble $sleep\ u$ contenant les transitions atteignables à partir de s mais qui ne sont pas exécutées. Ces techniques sont détaillées dans [61 , 64].

Abstraction : Cette technique consiste à ignorer quelques états et transitions du système à vérifier. Le but de cette technique est de produire un modèle abstrait à partir du modèle concret (du système à vérifier). Le modèle obtenu est plus petit, donc le graphe d'états sera plus petit. Le modèle abstrait est appproximativement équivalent au modèle concret [35 , 46].

Réduction compositionnelle : Cette technique [46] consiste à analyser une partie du système. Ayant un système S à vérifier, le but est de construire un système S' à partir de S, afin d'obtenir un modèle minimisé du modèle concret du système S sans générer le graphe complet de S. Un système est vu comme une composition de plusieurs composants qui interagissent. Le graphe d'états n'est pas généré directement, mais suivant les étapes suivantes :

- Générer l'espace d'états pour chaque composant ;
- Réduire chaque espace d'états ;
- Générer le graphe d'états complet à partir des sous graphes d'états réduits calculés pour chaque composant.

2.8.4 Raisonnement compositionnel

Le raisonnement compositionnel (ou la vérification compositionnelle) exploite la décomposition d'un système complexe en composants plus simples [1 , 46]. Les propriétés des composants du système sont vérifiées en premier. Ces propriétés sont ensuite combinées pour déduire les propriétés du système global. Évidemment, l'approche ne souffre pas d'explosion d'états, car elle

n'exige pas la construction de l'espace d'états du système. Un problème qui se pose cependant est que souvent, les propriétés des sous systèmes ne sont satisfaites que lorsque des hypothèses spécifiques sont faites sur leur environnement [19 , 46].
Ces techniques ont été réalisées pour réduire l'explosion de l'espace d'états, car l'objectif est d'explorer le plus d'états possibles, seulement, en menant des recherches sur l'espace mémoire, une métrique très importante "le temps d'exécution" est négligée. Un algorithme correct doit aussi être efficace. Evaluer l'efficacité temporelle d'un algorithme revient à calculer la durée nécessaire à l'exécution de l'algorithme. Plus cette durée est petite, plus grande est l'efficacité temporelle de l'algorithme.

2.9 Quelques Model checkers

Un grand nombre d'outils de vérification de modèles ont été développés au fil des ans. Dans cette section, nous présentons quelques *Model checkers* .

2.9.1 SPIN

Spin est un outil général pour s'assurer qu'un modèle vérifie sa spécification de façon rigoureuse. Il a été développpé par Gerard Holzmann [35 , 38] dans le groupe Unix, du centre de recherche en sciences informatiques chez Bell Labs au New Jersey. Le logiciel a été disponible gratuitement depuis 1991. La vérification par le *Model checker* Spin, se concentre sur la l'analyse des interactions entre les processus (composants du systèmes qui communiquent entre eux) et non pas aux calculs internes des processus. Les modèles à vérifier sont décrits dans le langage Promela (Process Meta Language) [50], qui représente le langage de modélisation de Spin. Les propriétés à vérifier sur le modèle, sont décrites en logique temporelle linéaire LTL [22]. Spin effectue la vérification du modèle "à la volée". Il utilise un algorithme de recherche efficace en profondeur d'abord, qui est compatible avec tous les modes de vérification supportés par l'outil, c'est-à-dire des recherches exhaustives, des techniques de réduction partielle. Ces techniques, ainsi que la compression d'états, sont utilisées pour traiter les grands espaces d'états.

2.9.2 SMV

SMV [48] est un outil pour vérifier les systèmes à états finis par rapport aux spécifications de la logique temporelle CTL. Il prend en charge un langage de spécification flexible, et utilise un algorithme de *Model checking* symbolique, pour vérifier efficacement si les spécifications CTL sont satisfaites par le système.

2.9.3 NuSMV

NuSMV [18] est une réimplémentation du *Model checker* SMV (symbolic *Model checker*) [48]. SMV est le premier outil, basé sur le *Model checking* symbolique, qui a permis de vérifier de façon exhaustive des systèmes de taille très élevée. Il a été développé par K. L. McMillan [47]. Il utilise le langage SMV pour modéliser les systèmes à vérifier.

2.9.4 KRONOS

Kronos [66] est un ensemble d'outils développés à Grenoble, au sein du laboratoire VERIMAG, dans le but de vérifier les systèmes temps réel complexes. Dans le *Model checker* Kronos, les composants des systèmes temps réel sont modélisés par des automates temporisés, et les exigences (propriétés) sont exprimées en logique temporelle TCTL [9].
Kronos est un outil logiciel visant à aider les concepteurs de systèmes temps réel, pour développer des projets répondant aux exigences spécifiées. Un objectif majeur de Kronos, est de fournir un moteur de vérification pour être intégré dans des environnements de conception des systèmes temps réel.

2.9.5 UPPAAL

Uppaal [10] est un outil intégré pour la modélisation, la validation et la vérification des systèmes temps réel modélisés comme des réseaux d'automates temporisés. Il a l'avantage de posséder une version texte et graphique, cette dernière permettant même la simulation pas-à-pas de systèmes dont le nombre d'états est trop grand, pour appliquer les algorithmes classiques de vérification.
L'outil est développé en collaboration entre le département des technologies de l'information à l'Université d'Uppsala, en Suède et le Département des sciences informatiques à l'université d'Aalborg au Danemark [10].

2.9.6 DIVINE

Divine [14] est un vérificateur de systèmes parallèles et distribués. Le projet DIVINE vise à développer un *Model checker* général, rapide, fiable et facile à utiliser. Divine est un outil pour l'analyse de l'exploration des systèmes distribués. L'outil est en mesure d'exploiter efficacement la puissance de calcul globale de plusieurs postes de travail multiples, interconnectés à travers le réseau, pour faire face à des tâches de vérification très lourdes. De ce fait, Divine permet d'analyser des systèmes, dont la taille est bien au-delà de la taille des systèmes qui peuvent être traités avec des vérificateurs séquentiels. Il permet de vérifier des modèles *larges* comprenant plusieurs états.

2.9.7 OBP

OBP [29] est outil qui permet d'explorer l'exécution des états du système. Il a été développé à l'ENSTA Bretagne à Brest. Il prend en entrée, un langage dédié à exprimer formellement les exigences sur le comportement du système, appelé CDL et la spécification du système écrite en langage Fiacre.

2.10 Conclusion

Le *Model checking* est un modèle de vérification des systèmes. Nous avons vu dans ce chapitre les avantages et inconvénients de cette technique de vérification. Nous avons présenté quelques *Model checkers* . Il existe deux types d'algorithmes de vérification de modèles basés sur l'exploration de l'espace d'états : algorithmes explicites et algorithmes symboliques (implicites). Dans les approches explicites, de vérification des modèles, il existe deux types de techniques : les techniques qui construisent complètement l'espace d'états, puis commencent à explorer les états, et les techniques à la volée, qui construisent progressivement l'espace d'états. La vérification symbolique évite de construire le graphe d'états de façon explicite et exploite les fonctions booléennes et les BDD pour représenter implicitement l'espace d'états. Dans le chapitre suivant, nous décrivant quelques solutions au problème d'explosion en temps et en espace mémoire.

Chapitre 3

Explosion en temps et en espace mémoire : Etat de l'art

Dans la nature, tout a toujours une raison. Si tu comprends cette raison, tu n'as plus besoin de l'expérience.

Lénonard De Vinci

Sommaire

3.1 Introduction

Le *Model checking* est une méthode de vérification qui explore tous les états possibles d'un système [32]. De cette manière, nous pouvons montrer qu'un modèle d'un système satisfait une propriété donnée. Dans cette technique, le système qui doit être vérifié, est modélisé en un système de transitions afin de décrire tous les états possibles du système. En outre, la propriété à vérifier sur le modèle du système est décrite dans un langage formel tel que la logique temporelle. Ensuite, le modèle et la propriété qui est décrite dans le langage formel, sont appliqués comme entrée à un vérificateur de modèles (*Model checker*). Après cela, le *Model checker* commence à explorer tous les états atteignables du modèle pour vérifier si la propriété est vraie. Sur la base de l'exploration de l'espace d'états, les algorithmes de vérification des modèles se divisent en deux catégories : les techniques dans lesquelles tous les états sont explorés et stockés en mémoire, ces algorithmes sont dits énumératifs. Ainsi que des techniques, qui représentent l'espace d'états du système implicitement en utilisant des fonctions booléennes. Ces approches explorent un ensemble d'états en une seule étape au lieu d'énumérer un état à la fois. Ces algorithmes sont connus sous le nom d'algorithmes symboliques. Les techniques qui explorent l'espace d'états se divisent en deux catégories : des techniques qui construisent complètement l'espace d'états, puis commencent à explorer tous les états atteignables et des techniques à la volée, qui génèrent dynamiquement l'espace d'états pendant les opérations de vérification du modèle.

3.2 Définition du problème

La vérification des modèles repose sur la modélisation d'un système et la vérification de ce modèle en fonction des propriétés souhaitées. Les spécifications du système sont généralement exprimées dans une logique temporelle ou sous forme d'automates.

Le problème auquel nous nous sommes intéressés dans le cadre de cette thèse est celui de l'explosion en temps d'exécution, et la mise au point de techniques permettant de réduire la complexité temporelle. Plusieurs études ont été réalisées pour réduire le problème d'explosion en espace mémoire surtout pour des systèmes temps réel, où le nombre d'états est très grand, car l'espace mémoire disponible et nécessaire pour explorer les états modélisant le système est insuffisant pour stocker tous les états visités. Le fait de dépasser la capacité de la mémoire physique, conduit vers un overlay qui amène à stocker les données sur un périphérique de stockage. Même si le fait de prendre en compte la complexité temporelle d'un algorithme rend sa complexité spatiale moins bonne, cela peut amener vers des recherches plus affrofondies pour prendre en compte les deux critères de performance. Plusieurs méthodes ont été mises au point pour réduire l'explosion en espace mémoire, peu de recherches ont été menées pour effectuer une vérification de modèles en un temps déterminé.

3.3 Model checking et exploration

Le *Model checking* est une technique de vérification, basée sur l'exhaustivité et l'exploration de l'espace des états des systèmes, dans la recherche de comportements qui ne vérifient pas leurs spécifications. Un vérificateur de modèles peut être considéré comme une boîte noire, qui accepte comme entrée un système ainsi qu'une propriété exprimée sur ce système et renvoie une réponse, indiquant si la propriété est vérifiée ou non. Lors de la vérification des propriétés, tous les comportements possibles du système sont énumérés et les propriétés sont vérifiées. Les algorithmes mis en œuvre comprennent deux phases, une construction de l'espace d'états puis une exploration de cet espace d'états dans la recherche d'erreurs. L'espace d'états est représenté comme un graphe qui décrit toutes les évolutions possibles du système. Les nœuds du graphe représentent les états du système et les arcs représentent les transitions entre ces états [2, 3]. L'analyse de l'accessibilité consiste à explorer les modèles état par état, chaque état et tous ses successeurs sont stockés en mémoire. L'exploration se termine lorsque tous les états sont visités. Un algorithme d'exploration, à chaque étape de son exécution, peut soit visiter un nouveau nœud, soit un nœud exploré. La *Figure. 3.1* présente un organigramme de l'accessibilité d'un modèle en utilisant un parcours en largeur (BFS) [36]. L'algorithme d'exploration en largeur (*Algorithm*

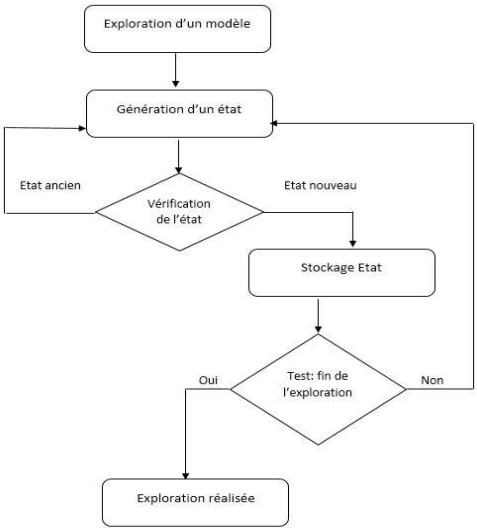

FIG. 3.1 – Etapes d'exploration d'un modèle

2) décrit les différentes étapes de traitement pour explorer tous les états d'un modèle.

s_0 représente l'état initial, S définit l'ensemble des états déjà explorés et M est une file contenant les états dont les successeurs n'ont pas encore été explorés. Tant que la file M n'est pas vide (Étape 3 de *l'Algorithme 2*), les états sont traités. La ligne 7 de l'algorithme indique

Algorithm 2 Algorithme d'exploration en largeur

```
1: S ← s0
2: M ← s0
3: while ¬(M.isEmpty()) do
4:     X ← M.dequeue()
5:     Successors ← X.genererSuccesseurs()
6:     for (State K : Successeurs) do
7:         if ¬ (S.Contains(K)) then
8:             S.Add(K)
9:             M.Add(K)
10:        end if
11:    end for
12: end while
```

que pour chaque état successeur, une vérification est effectuée sur l'ensemble S pour savoir si l'état est ancien ou nouveau. D'après cet algorithme, la phase d'exploration est composée de cinq étapes :

- Générer l'état initial et l'ajouter à l'ensemble des états explorés S ainsi qu'a l'ensemble M ;
- Vérifier si la file M est non vide, si oui, l'état qui se trouve en tête de file est retiré, sinon l'exploration se termine ;
- Pour chaque état de la file, générer ses successeurs et vérifier pour chaque état généré s'il a été observé ou pas ;
- Stocker les états successeurs dans les 2 ensembles M et S. Les états observés seront ignorés afin d'éviter d'avoir des duplications d'états. On dit qu'un état à été exploré s'il appartient à l'ensemble des états visités S ;
- Répéter ce processus à partir de l'étape 2 jusqu'à ce que la file M soit vide.

3.4 Etat de l'art

Nous allons présenter une solution basée sur une exploration parallèle, visant à obtenir une meilleure exploration en temps sur l'outil Spin. Les autres solutions ont été proposées pour le problème d'explosion de l'espace d'états. Ces approches sont exécutées sur des architectures distribuées, parallèles ou séquentielles. Chacune d'elles est basée sur des structures de données différentes.

3.4.1 Architecture séquentielle

Un algorithme correspond à une succession d'états et à des transitions entre ces états. L'idée est que les états correspondent à des descriptions instantanées de l'algorithme. Un algorithme d'exploration est défini principalement par trois paramètres : un critère d'arrêt de l'exploration, une fonction de sélection des états successeurs, à partir d'un état ou d'un ensemble d'états déjà explorés et stockés en mémoire, et une fonction qui permet d'effectuer le stockage de nouveaux états en mémoire. L'exécution de deux séquences est dite séquentielle lorsqu'elles sont exécutées l'une à la suite de l'autre. Dans une approche séquentielle, l'exploration est effectuée par un seul processus, le temps d'exploration peut être énorme si le nombre d'états est très élevé. Dans une exploration séquentielle, les états sont visités l'un à la suite de l'autre, le premier état visité est l'état initial, par la suite, les successeurs de cet état sont générés, puis stockés en mémoire. Ce processus s'arrête lorsque tous les états sont traités.

L'approche proposée dans [62] est basée sur une représentation symbolique des états. Ces derniers sont stockés dans une structure OBDD (Object Binary Decision Diagram) [16], qui est un arbre acyclique représentant des fonctions booléennes. C'est une forme réduite des diagrammes BDD [42] (voir chapitre 2 section 2.8.2). Pour chaque état exploré, un arbre OBDD lui est généré. Cela permet de stocker les états explorés. Les parties gauches de l'arbre représentent la valeur booléenne 0, et les parties droites la valeur 1. Les diagrammes OBDD peuvent être directement manipulés pour effectuer efficacement toutes les opérations booléennes de base [45]. Si un état s est exploré, il peut être représenté par une fonction booléenne f(s) = 1. La *Figure. 3.2* présente un arbre de décision binaire composé de 3 états, représentant la fonction $f(x_1, x_2, x_3) = \bar{x_1}.\bar{x_2}.\bar{x_3} + \bar{x_1}.x_2.\bar{x_3} + x_1.\bar{x_2}.\bar{x_3}$.

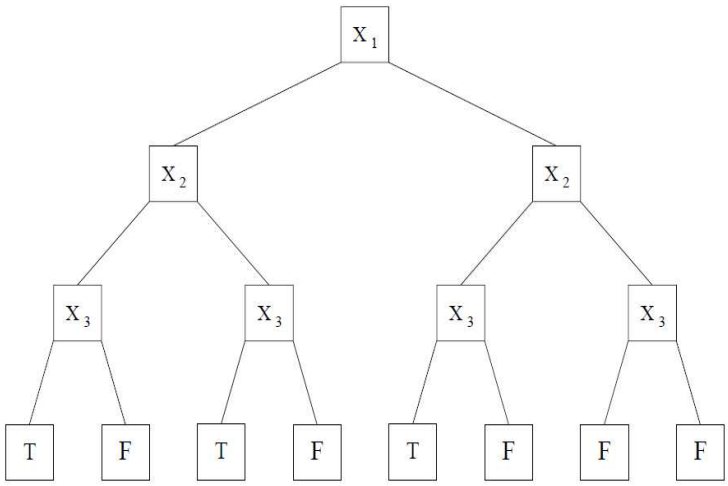

FIG. 3.2 – Arbre de décision binaire représentant la fonction $f(x_1, x_2, x_3)$

Le diagramme OBDD correpondant (*Figure. 3.3*) est modifié en tenant compte de quelques règles de telle sorte à stocker les états de façon implicite. Les règles à respecter sont :

- **Supprimer les terminaux dupliqués** : Il ne doit y avoir qu'un seul terminal avec un label donné. Tous les arcs entrant vers ce nœud terminal doivent être redirigés vers le nœud possédant le même label entrant.
- **Supprimer les non terminaux dupliqués** : Si deux nœuds non terminaux ont le même label et leurs arcs gauches et droits sont les mêmes, alors il faut les supprimer. Tous les arcs doivent être redirigés vers le nœud non terminal portant le même label.
- **Supprimer les tests redondants** : Si deux arcs sortants d'un même nœud non terminal t pointent vers le même nœud t', alors t peut être supprimé et tous les arcs entrant vers ce nœud sont redirigés vers le nœud t'.

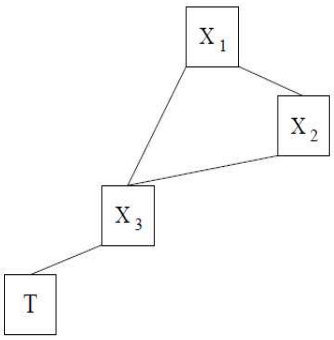

FIG. 3.3 – Diagramme OBDD représentant la fonction $f(x_1, x_2, x_3)$

Ces trois règles sont appliquées pour chaque arbre OBDD généré. Lorsque plusieurs arbres OBDD sont générés, il est nécessaire de les regrouper grâce à une fonction OR. Ayant deux arbres OBDD1 et OBDD2, l'arbre global résultant est généré grâce à la fonction OR(OBDD1,OBDD2). L'exploration de l'espace d'états a été réalisée en utilisant le *Model checker* Spin [38]. Le modèle sur lequel les expérimentations ont été réalisées représente n compteurs qui s'incrémentent de 0 à 9 puis retournent à la valeur 0 qui représente l'état initial. Deux structures de données ont été utilisées, les tables de hachage et les structures OBDD. La comparaison a été réalisée sur l'espace mémoire utilisé et sur le temps d'exécution. L'espace mémoire nécessaire pour effectuer l'exploration de l'espace d'états est plus important en utilisant les tables de hachage comparativement aux structures OBDD, seulement, le temps d'exécution est énorme lors de l'utilisation des structures OBDD. Cela s'explique par le fait que lors de la vérification d'un état (s'il a été exploré ou pas), le temps de recherche au niveau de cet arbre est très élevé. Pour cela, les auteurs ont utilisé une méthode de compression [3] afin de réduire le vecteur de bits, car la temps de recherche dépend de la taille du vecteur d'états. Cependant, l'utilisation de la table de hachage

reste une meilleure solution en performances par rapport aux structures OBDD. Même si l'utilisation des structures OBDD permet d'avoir un gain en mémoire, le fait d'utiliser les tables de hachage permet d'avoir un équilbrage en performance, entre temps de traitement et espace mémoire nécessaire pour l'exploration.

Cette structure de données a été utilisée dans [49] pour le stockage des états explorés. L'approche proposée se base sur le stockage des états sous une forme compressée. Le modèle exploré consiste en un réseau de Petri coloré. L'algorithme proposé est un algorithme séquentiel. Un état est représenté par un nœud, le passage d'un état à un autre est représenté par une transition. Le premier état généré (état initial) est stocké de façon explicite, les autres états sont stockés de façon compressée. L'information sauvegardée concerne la différence (D) entre l'ancien état et le nouvel état. Elle est représentée par l'expression suivante : $D = Etat\_nouveau - Etat\_ancien.$
Cette forme de compression d'états permet d'avoir un gain en espace mémoire et par conséquent plus d'états sont explorés. Lorsqu'un état est généré, il faut vérifier s'il n'a pas déjà été exploré, pour cela, il faut le comparer avec tous les états visités. Afin d'effectuer la compraison, ces derniers doivent être sous leur forme explicite. Donc, tous ces états sont décompressés. Pour cela, il faut pour chaque état compressé, remonter jusqu'à l'état initial et combiner à chaque étape les données stockées afin de regrouper toutes les informations qui composent l'état. Si l'état n'est pas décompressé, plusieurs états peuvent avoir la même différence de forme (le même D), donc ces états seront identiques (dupliqués) alors qu'ils n'ont pas les mêmes données.
La fonction de retour en arrière représente une surcharge, car il faut remonter jusqu'à l'état initial qui est stocké de façon compressée. Donc, le temps d'exécution de l'exploration augmente ce qui représente l'inconvénient de l'utilisation de cette méthode. En utilisant cette fonction, il faut à chaque fois retourner à l'état initial afin d'avoir toutes les données sur l'état stocké. Afin de limiter cette surcharge et par conséquent réduire le temps nécessaire pour exécuter cette fonction, les auteurs du travail [49] ont étendu leur algorithme en introduisant un nouveau paramètre δ. Tous les états qui se trouvent à une profondeur $i\delta$ (i varie de 1 à N, tel que N est un entier) de l'état initial sont stockés sous leur forme explicite. δ représente la plus petite distance entre deux états explicites. Ce paramètre est fixé lors des expérimentations réalisées. Le choix de cette profondeur est très important, car lorsque celle-ci est petite, le temps d'exécution diminue, seulement, l'espace mémoire nécessaire augmente. L'algorithme proposé est divisé en 3 parties :

- Algorithme *Search* : cet algorithme a pour but de vérifier si l'état généré est ancien ou bien nouveau. Dans le cas d'un ancien état, ses successeurs seront générés et traités, sinon l'état est stocké sous sa forme compressée en appelant la procédure *Insert* . Cet algorithme a comme paramètres d'entrées, l'état recherché, la distance entre cet état généré et le dernier état stocké de façon explicite S_{exp} ainsi que l'état S. Lorsqu'un état est généré, un index de l'état est généré grâce à une fonction de hachage h. Cela permet de rechercher l'état dans la table de hachage. Si l'index n'est pas trouvé, cela veut dire que l'état est nouveau et l'état

sera stocké dans la table de hachage à l'index généré. Si l'index est trouvé, cela veut dire qu'il y a des états stockés dans cet index. Ce dernier peut pointer vers une liste chaînée, dans ce cas, l'état généré est comparé avec chaque état de cette liste. Cette recherche est effectuée pour chaque état généré, ce qui nécessite plusieurs accès mémoire, donc, le temps de calcul va augmenter.

- Algorithme *Insert* : cet algorithme a pour but d'insérer un état au niveau de la table de hachage.
- Algorithme *Reconstruct* : cet algorithme est utilisé pour décompresser un état, il accepte comme paramètre, la différence D entre deux états. Il représente la fonction de retour en arrière vers un état explicite pour reconstituer un état implicite.

3.4.2 Architecture parallèle

Le parallélisme consiste à mettre en œuvre des architectures permettant de traiter des informations de manière simultanée. Dans ce qui suit, nous présentons 2 solutions au probléme d'explosion dans un environnement parallèle, la première pour gérer la complexité spatiale et la seconde prend en compte la complexité temporelle de l'algorithme d'exploration. La première solution a été étudiée dans [57]. Comme pour les 2 approches séquentielles précédentes (voir section 3.4.1) utilisant des structures de données visant à augmenter le plus possible le gain en espace mémoire. La solution décrite en détail dans [57] se base sur l'utilisation d'arbres AVL [2]. Le nom AVL provient de ceux qui ont créé cette structure. Ceux sont des arbres binaires de recherche. Les auteurs ont proposé un algorithme d'exploration appelé *general lock free* , exécuté dans une architecture parallèle à mémoire partagée. L'espace d'états est partagé par les processeurs et l'accès à cette dernière se fait en exclusion mutuelle. L'algorithme présenté par les auteurs est basé sur 2 structures de données, un *filtre de bloom* (structure de données probabiliste) [57], partagé entre les différents processeurs pour indiquer si un état a été visité ou pas, ainsi que des piles privées et publiques. Chaque processeur gère une pile privée contenant les états qui doivent être explorés et une pile partagée contenant les états partagés par les processeurs (voir *Figure. 3.4*). Le *filtre de bloom* a été utilisé, car il est rapide en temps d'exécution et compacte en mémoire. Lorsqu'un état est généré, un test est réalisé sur cet état afin de vérifier s'il est ancien ou nouveau. Cette information est donnée par le *filtre de bloom* . Dans le cas où l'état est nouveau, il est stocké dans un arbre binaire. Dans certains cas, le *filtre de bloom* peut rendre un résultat positif alors que l'état n'a pas encore été vu. Cela est appelé un faux positif. Cette situation où des états sont supposés être traités alors qu'ils ne le sont pas, est appelée collision d'états. Ces états sont stockés dans un arbre binaire réservé aux collisions. L'algorithme est réparti en trois phases, exploration, collision et terminaison.

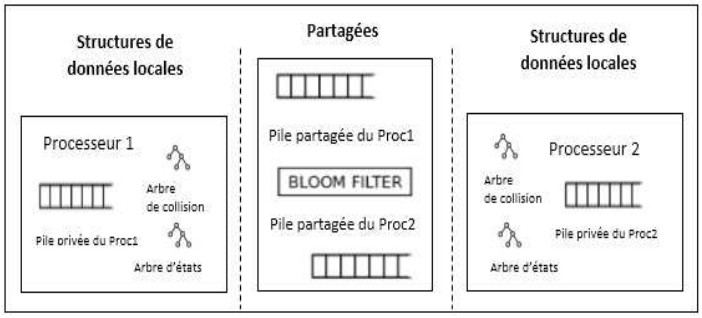

FIG. 3.4 – Structures de données utilisées

- **Phase d'exploration** : Cette phase est exécutée en boucle. A chaque fois qu'un nouvel état est exploré, il est stocké dans un arbre binaire de recherche. L'algorithme s'arrête lorsque les piles privées et publiques sont vides. A ce moment, si au moins une structure d'arbre AVL de collision est non vide, l'algorithme passe à la seconde phase (phase de collision) afin de vérifier les faux positifs.
- **Phase de collision** : Cette phase consiste à vérifier pour chaque état qui se trouve dans l'arbre de collision s'il a été visité ou pas afin d'éviter les faux positifs. Cela se fait en vérifiant l'état avec tous les états stockés dans les arbres AVL (arbre pour le stockage des nouveaux états) de chaque processeur.

  L'insertion d'un état au sein de l'arbre AVL est réalisée de la même manière que pour les arbres binaires. Chaque élément inséré possède une clé, la clé de l'état à insérer est calculée. Lorsqu'une feuille est atteinte, cette dernière deviendra un nœud et l'état à insérer sera dans la branche droite ou gauche du nœud. Il sera stocké dans la partie gauche si sa clé est inférieure à celle du nœud et à droite si elle est supérieure. Donc, pour vérifier pour chaque processeur dans cette phase si un état a déjà été visité, la vérification commence par la partie gauche. Si la clé de l'état recherché est inférieure à la clé de l'état stocké, la recherche s'arrête et on conclut que l'état n'a pas été exploré (car la recherche commence par le nœud de gauche qui a la clé la plus petite). Dans ce cas, l'état sera mis dans la pile privée du processeur et la phase d'exploration sera déclenchée. Sinon, la recherche passe au second état dont la clé est directement supérieure à la clé de l'état précédent (les clés sont triées). La recherche se termine avec un état dont la clé est égale à la clé d'un état stocké ou avec une clé non trouvée, dans ce cas, l'état sera mis dans la pile privée du processeur et la phase d'exploration sera déclenchée.
- **Phase de terminaison** : Cette phase est déclenchée lorsque les piles privées et publiques sont vides et lorsque les arbres binaires réservées aux collisions sont vides.

L'inconvénient de cette méthode est que plusieurs états qui ont déjà été traités vont être revérifiés à nouveau, à cause du *filtre de bloom* qui risque de retourner un faux positif. Les auteurs ont utilisé cette structure afin d'éviter de vérifier sur l'arbre AVL qu'un état a été exploré, seulement, cela amène vers un traitement supplémentaire à réaliser qui nécessite plus de calcul, lorsque le *filtre de bloom* indique que l'état est ancien.

Une autre solution parallèle a été présentée dans [58]. Cette approche a été proposée par les auteurs de la solution précédente. L'architecture utilisée est une architecture multiprocesseurs à mémoire partagée. Les états sont stockés dans des tables de hachage locales. Chaque processeur détient une pile privée et une pile partagée. La coordination des états entre les processeurs se fait à travers une table de localisation (LT), qui contient la liste des états ayant été visités. Le *filtre de bloom* a été remplacé par une table de localisation dont le rôle primaire est de retourner vrai si l'état a été visité ainsi que le numéro du processus exécuté, faux sinon. Le travail réalisé par chaque processus est le suivant : prendre un état, vérifier s'il n'a pas été visité et cela en consultant la table de localisation (LT) qui va rendre l'identifiant (id) du processus dans le cas où l'état a déjà été généré, sinon l'état est traité par le processus l'ayant généré.

La *Figure. 3.5* présente un exemple de table de localisation. Cette table représente un vecteur dont chaque cellule contient deux informations. La représentation est la suivante : (id, h(état)). Lorsqu'un processus vérifie si un état s a été visité ou pas, il calcule sa clé $h(s)$ grâce à une fonction de hachage h et grâce à la fonction map qui permet d'accéder à cette table, on accède à la cellule correspondante. Si le contenu est vide cela veut dire que l'état est nouveau, des informations sur cet état sont insérées dans cette table sous la forme (id,h(s)). Aussi, l'état sera stocké dans une table de hachage locale du processeur en question. Si la cellule n'est pas vide, comparer les clés des états. Si ces deux valeurs correspondent, cela veut dire que l'état a déjà été visité et à ce moment, le processus s'assure de cela en vérifiant si cet état se trouve dans la table de hachage du processeur qui a traité l'état. Si effectivement il a été traité, il passe à l'état suivant, sinon, cela veut dire qu'il y a collision et cet état sera empilé dans la pile partagée du dernier processeur consulté lors de la génération de clés pour insérer un nouvel état. La *Figure. 3.5* présente l'insertion de trois états (x, y, z) dans une table de localisation contenant 4 cellules. Les valeurs des clés varient entre 1 et 31. Le nombre de fonctions de hachage est fixé à 2.

L'inconvénient de cette méthode est l'application d'une vérification supplémentaire au niveau de la table de hachage d'un processeur pour s'assurer si un état a réellement été visité.

Dans les méthodes qui ont été présentées, les auteurs se sont focalisés sur l'aspect mémoire uniquement en négligeant le temps d'exécution qui est une métrique importante pour l'analyse d'un algorithme. Par exemple dans la solution proposée dans [49], les états compressés sont reconstitués dans leurs formes explicites pour vérifier si un état a été visité ou pas. Ce traitement prend beaucoup de temps. Dans une architecture distribuée, plusieurs machines vont contribuer à l'exploration d'une partie du modèle. Dans le cas où une machine est confrontée au problème

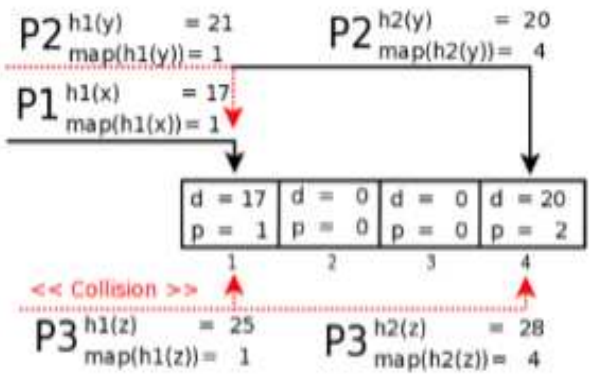

FIG. 3.5 – Exemple d'une table de localisation permettant l'insertion de trois états x, y, z

d'explosion en espace mémoire, une partie des états est traitée par d'autres machines. Le fait d'atteindre le problème d'explosion combinatoire et de répartir à chaque fois les états vers d'autres nœuds du réseau entrainera une explosion en temps d'exécution.

Les travaux de [39] ont présenté un algorithme parallèle rajouté au *Model checker* Spin [38], afin d'avoir un gain en temps d'exploration des états, en exploitant une machine multi-coeurs.

Spin est un outil de vérification et de simulation des systèmes concurrents. Pour être vérifié, un système concurrent est décrit dans le langage Promela (Process Meta Language), qui représente le language de spécification des modèles dans Spin. L'algorithme (*Algorithme 3*) proposé dans [39] est basé sur l'utilisation d'une file à trois dimensions Q[t][i][j] pour le stockage des états, dont les successeurs n'ont pas encore été observés. Le paramètre t, varie de 0 à 1, il permet de passer d'une exploration des états actuels vers une exploration des états futurs. Puisque l'algorithme est parallèle, plusieurs threads vont construire une partie de l'espace d'états. Le nombre de threads est statique, il est défini au début des expériences. À chaque étape d'exploration, tous les états de Q[t][i][j] sont traités et leurs successeurs sont stockés dans Q[1-t][k][i], correspondant aux configurations (états) qui seront observées à l'étape suivante. k est choisi aléatoirement, il représente l'indice du thread qui va explorer l'état successeur. Une table de hachage a été utilisée pour le stockage des états déjà observés. La structure de données Q est définie de façon statique et est construite en fonction du nombre de threads définis lors des expérimentations. Une tâche importante dans l'algorithme proposé dans [39], consiste à déterminer le moment auquel tous les états auront été explorés pour arrêter l'exploration.

Algorithm 3 Parallel Exploration Algorithm in $SPIN$

1: $done \leftarrow false$
2: $t \leftarrow 0$
3: Chercher (i : 1..N)
4: **while** (!done) **do**
5: **for** (chaque j dans 1..N) **do**
6: Prendre un état s de Q[t][i][j]
7: **for** (chaque état successeur c de s) **do**
8: **if** ¬ (S.Contains(c)) **then**
9: S.add(c)
10: $k \leftarrow$ choisi aléatoirement de 1..N
11: Ajouter l'état c à $Q[1-t][k][i]$
12: **end if**
13: **end for**
14: **end for**
15: Wait()
16: **if** (i==1) **then**
17: Attendre que tous les threads soient en état de veille
18: **if** (all $Q[1-t][i][j]$ == NULL) **then**
19: done $\leftarrow$ true
20: **else**
21: Réveiller tous les threads (notify_all_threads())
22: $t \leftarrow$ 1- t
23: **end if**
24: **end if**
25: **end while**

L'inconvénient de cette solution est l'utilisation d'une file à 3 dimensions pour le stockage des états dont les successeurs n'ont pas encore été visités. Ce qui nécessite d'avoir plus d'allocation mémoire comparativement à l'utilisation d'une file simple. Aussi, le temps d'accès mémoire à cette structure de données est plus long. Durant cette étude, nous avons proposé un algorithme parallèle et nous avons réalisé des comparaisons avec ce dernier algorithme.

3.4.3 Architecture distribuée

L'analyse de l'accessibilité d'un modèle permet d'effectuer une exploration globale de l'espace d'états lors de la vérification par *Model checking* . Le problème majeur de cette méthode est le problème d'explosion du nombre d'états en temps et en espace mémoire. Plusieurs méthodes ont

été proposées pour apporter une solution à ce problème. Certaines se basent sur la distribution du calcul sur plusieurs machines pour gérer l'explosion en espace mémoire. Dans la solution proposée dans [44], les auteurs considèrent une distribution de l'exploration d'états pour les CPN (réseaux de Petri colorés). La principale contribution des auteurs est d'explorer l'utilisation de la distribution de l'exploration de l'espace d'états des modèles CPN. L'une des particularités de la distribution est la transparence de traitement pour l'analyste, comme si il y avait une seule machine qui effectue l'exploration. Le temps nécessaire pour l'exploration de l'espace d'états d'un modèle CPN est déterminé par le temps de génération des états et de leurs successeurs, par le temps nécéssaire pour vérifier si un état a déjà été visité ou pas, ainsi que par le temps d'envoi/réception de messages dans le réseau. Le temps d'exploration peut être très élevé pour des modèles CPN avec plusieurs jetons et transitions.

Une distribution du calcul pour l'exploration de l'espace d'états peut garantir trois choses essentielles :

- Le calcul des états successeurs peut être effectué en parallèle pour différents nœuds.
- La vérification d'un état s'il a été exploré ou pas est réalisée en parallèle.
- L'espace mémoire disponible pour l'exploration de l'espace d'états augmentera grâce à une architecture composée de plusieurs machines.

Les auteurs ont proposé d'exécuter l'exploration de l'espace d'états dans une architecture distribuée, en distribuant le stockage et le calcul des états successeurs sur plusieurs machines. La distribution du calcul des états successeurs peut être très avantageuse car le calcul peut être coûteux surtout lorsque le nombre de jetons est grand au niveau d'une place ou le nombre de transitions est élevé. Les auteurs ont utilisé un processus coordinateur pour distribuer les états sur les nœuds du réseau. L'utilisation d'un coordinateur facilite la détection de la terminaison, car la distribution d'états s'arrête. Lorsqu'un processus traite un état successeur, il vérifie s'il est responsable de cet état, cela grâce à une fonction de hachage. S'il n'est pas responsable de l'état, il l'envoie au coordinateur qui l'enverra au nœud adéquat. Dans le cas contraire, il vérifie localement si l'état à traiter est nouveau. Chaque processus s'exécute sur une machine. Les machines ont les mêmes caractéristiques matérielles. La communication entre les processus et le coordinateur se fait à travers un réseau local. L'algorithme (*Algorithm 4*) [44] présente le code exécuté par le coordinateur pour la distribution des états.

Algorithm 4 Algorithme exécuté par le coordinateur

```
1: computed ← false
2: send(stateM_0, worker(h_ext(M_0)))
3: nextprobe ← h_ext(M_0)
4: send(probe, worker(nextprobe))
5: nextprobe ← nextprobe + 1
6: while ¬computed do
7:    for all i ∈ 1, ..., n do
8:       if canReceive(worker(i)) then
9:          Receive(message, worker(i))
10:         if message == state M then
11:            nextprobe = min(nextprobe, h_ext(M))
12:            send(state M, worker( h_ext(M)))
13:         else
14:            {probe returned}
15:            if nextprobe > n then
16:               computed ← true
17:            else
18:               send(probe, worker(nextprobe))
19:               nextprobe ← nextprobe + 1
20:            end if
21:         end if
22:      end if
23:   end for
24: end while
25: for all i ∈ 1, ..., n do
26:    send(stop, worker(i))
27: end for
```

Le processus coordinateur commence par envoyer le premier état (état initial) M_0 à un processus déterminé par une fonction de hachage $h_{ext} : M \rightarrow 1, 2, 3, ..., n$, utilisée pour la distribution des états entre les différentes machines. Cette distribution doit être uniforme sur les processus. Cette dernière doit aussi respecter la localité des états, par exemple, regrouper plusieurs états successeurs dans une même machine ce qui veut dire que leur traitement sera effectué par le même processus. Cela évite la surcharge de communication entre machines. Par la suite, le coordinateur envoie au processus un message $probe$. Le message $probe$ et la variable $nextprobe$ servent à détecter la terminaison de la phase d'exploration. La boucle $while$ du code permet de récupérer les messages reçus des différents processus. Un message reçu peut être un état envoyé d'un processus vers le coordinateur, pour le prévenir qu'il n'est pas le processus chargé de l'exécuter ou bien un message $probe$. Lorsque la valeur de la variable $nextprobe$ est supérieure au nombre de nœuds (processus), l'exploration s'arrête.

L'archictecture proposée dans [44] est composée d'un coordinateur et de plusieurs nœuds réalisant l'exploration d'un modèle. Seulement, nous pouvons remarquer que le nœud coordinateur représente un goulot d'étranglement, car tous le traitement passe par ce processus. Aussi, dans une architecture distribuée, plusieurs messages vont transiter à travers le réseau ce qui peut amener à un temps d'exécution important.

Une autre approche distribuée a été présentée dans [34], les auteurs ont proposé d'exploiter les ressources (calcul et espace mémoire) d'un ensemble de machines, afin d'avoir un gain en espace mémoire. La distribution d'états représente une tâche délicate car le partitonnement d'états entre les nœuds doit être équitable. Afin de réaliser l'exploration, les auteurs ont utilisé un *Model checker* symbolique nommé *Division* . Dans une représentation symbolique, tous les états du système à vérifier sont encodés par des fonctions booléennes. A l'étape d'exploration, des structures de données appelées BDD (binary decision diagram) sont utilisées pour représenter les fonctions booléennes. Dans le *Model checking* symbolique, à chaque itération, l'exploration est réalisée sur un ensemble d'états (les états ne sont pas explorés l'un à la suite de l'autre). L'architecture utilisée est composée de plusieurs nœuds qui peuvent être coordinateurs ou machines ($workers$) réalisant l'exploration. Chaque machine gère deux ensembles R et N. R contient les états explorés et N contient les états dont les successeurs n'ont pas encore été explorés. Le traitement réalisé par les workers est le suivant :

- Un seul $worker$ débute l'exploration de l'espace d'états, s'il y a explosion de l'espace d'états, les 2 ensembles R et N seront répartis sur k workers disponibles. Chaque $worker$ traitera une parti du modèle.
- Chaque $worker$ vérifie s'il doit traiter l'état généré ou pas. Cela est réalisé par une fonction de hachage. Dans le cas où un état doit être traité par un autre $worker$, l'état est envoyé au nœud adéquat. Une explosion de l'espace d'états peut être causée par un nombre importants d'états successeurs à générer.

Il existe deux types de coordinateurs, ceux qui gèrent la répartition d'états sur les machines *workers* en cas d'explosion d'états, et ceux qui s'occupent de rassembler les états dans le cas où les machines n'ont pas beaucoup d'états à explorer. Dans ce cas, une seule machine sera sélectionnée pour traiter les états restants.
La solution se base sur l'exploration des modèles dans une architecture distribuée. Le nombre de messages envoyés entre les machines est très important et par conséquent, le temps d'exécution augmente rapidement. La taille des graphes BDD dépend de l'ordre des variables qui composent le vecteur d'états. Si le bon ordre est réalisé, le graphe sera minimal, seulement, il existe des fonctions booléennes pour lesquelles la complexité du graphe BDD retourné est exponentielle même avec un ordre correct entre les variables [32]. Dans ce cas, le temps de calcul ainsi que l'espace mémoire nécessaire pour réaliser l'exploration est très important.

3.5 Conclusion

La technique du *Model checking* souffre d'une limitation appelée explosion en temps et en espace mémoire. Dans le premier cas, ce phénomène se produit si l'exploration se réalise en un temps indéterminé, à cause de l'accès à plusieurs structures de données en mémoire dans une architecture multiprocesseurs ou multi-coeurs, ou, à cause d'un échange important entre processus dans une architecture distribuée. Dans le deuxième cas, le problème se produit lorsque le système en cours de vérification comporte plusieurs composants, ce qui génère des transitions en parallèle. Dans ce chapitre, nous avons présenté une solution au problème posé et avons apporté quelques critiques sur les autres solutions présentées, pour gérer l'espace mémoire. Ces solutions sont exécutées sur des architectures différentes. Le choix des structures de données utilisées est très important, car cela influe sur le temps et l'espace mémoire utilisé. Par exemple, dans la solution utilisant les diagrammes OBDD, les auteurs ont eu de meilleurs résultats en performances en utilisant des tables de hachage. La compression d'états possède un avantage majeur parmetant un gain en espace mémoire, seulement, le temps d'exécution augmente rapidement en réalisant une décompression d'états. Donc chaque solution a ses avantages et ses limites. Dans le chapitre suivant, nous contribuons par deux algorithmes parallèles et un algorithme distribué pour le problème de l'explosion en temps d'exécution.

Chapitre 4

Contributions

La connaissance s'acquiert par l'expérience, tout le reste n'est que de l'information.

Albert Einstein

Sommaire

4.1 Introduction

La vérification par *Model checking* est une technique de vérification automatique, qui permet de vérifier qu'un système satisfait une spécification donnée. Il s'agit d'un outil couramment utilisé dans les situations, où il est essentiel de certifier le bon fonctionnement d'un système. Ainsi, les outils de vérification des modèles sont largement utilisés dans les industries de haute technologie, pour la vérification des circuits électroniques, ou même dans l'aéronautique, pour assurer la sécurité des systèmes embarqués [13]. Tout algorithme de vérification de modèles est basé sur deux étapes : (1) exploration des états atteignables du système et (2) vérification des spécifications dans cet espace d'états. Ces deux étapes peuvent parfois être effectuées simultanément, ce qui est appelé la vérification à la volée. L'exploration est un processus informatique qui détermine une séquence d'actions, permettant d'atteindre un objectif souhaité. Une bonne exploration signifie, la réalisation et le stockage d'un grand nombre d'états sans dépasser les ressources mémoire disponibles [1] et en un temps fini. L'espace d'états peut être décrit par un état initial et un ensemble de transitions. Une succession d'états produits par des actions forment un chemin dans l'espace d'états [2, 3]. Dans le cas d'exemples réels, le nombre d'états peut être énorme, par exemple, dans un compteur de n bits, le nombre d'états est exponentiel (2^n). L'analyse de l'accessibilité est limitée par le problème d'explosion de l'état [4, 5, 6]. Ce problème se produit lorsque l'espace d'états à explorer est grand et ne peut être exploré par les algorithmes, par manque d'espace mémoire parce que l'espace mémoire nécessaire pour effectuer l'exploration est plus élevé que l'espace mémoire contenu dans la machine. De nombreuses approches ont été proposées pour traiter plus d'états, seulement, ces algorithmes traitent surtout l'aspect stockage et néglige la métrique du temps de calcul.

4.2 Objectifs

Dans cette thèse, nous nous sommes intéressés à l'analyse du temps d'exécution nécessaire à l'exploration d'un modèle. Au cours de l'analyse d'accessibilité, il est essentiel de prendre en compte la notion du temps, car un système est un ensemble cohérent de composants en interaction, ce qui amène à explorer plusieurs états. Même en distribuant le graphe d'états sur plusieurs machines (dans un cloud par exemple), le problème ne sera pas résolu, en raison de la croissance exponentielle du nombre d'états surtout pour les systèmes temps réels. Cette distribution à un impact négatif sur le temps d'exécution, à cause du nombre de messages échangés, car le réseau devient un goulot d'étranglement surchargé par les envoi/réception des requêtes. Dans ce cas, nous avons donc besoin de traiter l'explosion temporelle. Pour cela, nous avons proposé 3 algorithmes, 2 exécutés dans un environnement parallèle et un dans un environnement distribué.

4.3 Exploration et graphe d'états

Afin d'illuster le processus d'exploration, nous allons prendre l'exemple suivant [43] : Alice a un chat, Bob a un chien. Alice et Bob partagent un même jardin. Le chien ainsi que le chat ne peuvent pas être en même temps dans le jardin. Comment Alice et Bob peuvent partager le jardin afin que leur animaux puissent y accéder à tour de rôle ? Cela représente un problème d'exclusion mutuelle. Afin d'effectuer une exploration de ce modèle, il faut identifier tous les états possibles, ceci est réalisé par l'association *conditions/actions* . Nous allons dans un premier temps identifier les évènements possibles entre Alice et Bob, ensuite, nous allons construire le graphe d'accessibilité de ce modèle. Afin qu'un seul animal puisse ètre dans le jardin à la fois. Alice et Bob vont utiliser des drapeaux pour tester si la section critique (le jardin) est libre. Lorsqu'un animal veut aller dans le jardin, l'un des deux cas suivants se produit :

Premier cas

- Alice/Bob lève son drapeau ;
- Le drapeau de Bob/Alice est baissé ;
- Le chat/chien entre dans le jardin ;
- Le chat/chien libère le jardin et Alice/Bob baisse son drapeau.

Deuxième cas

- Alice lève son drapeau ;
- Bob lève son drapeau ;
- Bob baisse son drapeau et se met en attente jusqu'à ce qu'Alice baisse son drapeau ;
- Le chat entre dans le jardin ;
- Le chat libère le jardin et Alice baisse son drapeau ;
- Bob lève son drapeau et le chien entre dans le jardin.

Suivant les deux cas cités ci-dessus, nous pouvons identifier les états possibles :

- Etat 1 : état initial.
- Etat 2 : état de test qui consiste à lever le drapeau pour vérifier si le jardin est libre.
- Etat 3 : état d'accès au jardin pour le chat et état d'attente du chien pour entrer en section critique.
- Etat 4 : état d'accès au jardin pour le chien.

A chaque évenement, nous avons le passage d'un état vers un autre. Ce passage est représenté par une transition. Un état est représenté par un vecteur de bits. Dans cet exemple, le vecteur d'état est composé de 4 bits (voir *Figure. 4.1*) contenant respectivement les informations suivantes :

- Drapeau d'Alice. Si le drapeau est levé, la valeur sera égale à 1.
- Drapeau de Bob. Au début de l'exploration, les 2 drapeaux sont initialisés à 0.
- Etat d'Alice
- Etat de Bob. Initialement l'état d'Alice et de Bob est fixé à 1. Par exemple, la configuration

<1 0 2 1> indique qu'Alice a levé son drapeau et est passée en état de test (état 2).

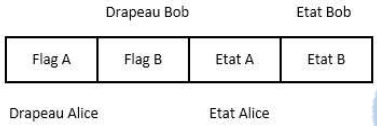

FIG. 4.1 – Structure du vecteur d'état de l'exemple

L'état d'Alice et de Bob peut varier de 1 à 3 et de 1 à 4 respectivement.

Le graphe d'accessibilité obtenu suivant les deux cas cités ci dessus est présenté dans la *Figure 4.2* .

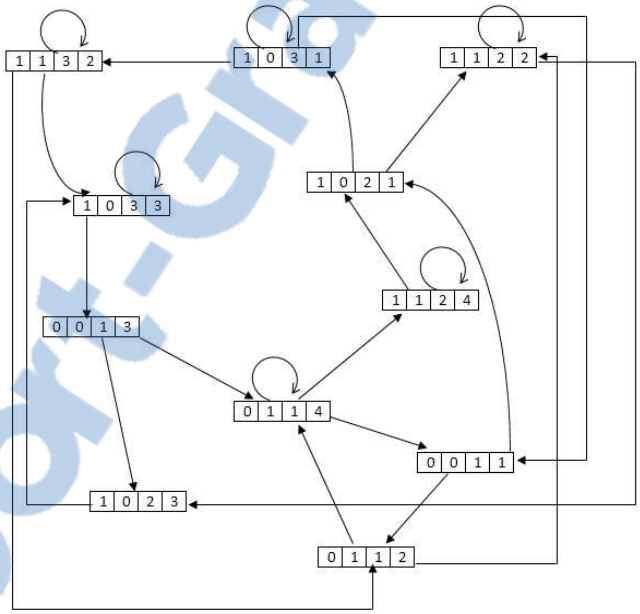

FIG. 4.2 – Graphe d'accessibilité du modèle Alice et Bob

4.4 Contributions

Un algorithme parallèle s'exécute sur un ordinateur parallèle pour résoudre un problème donné. Les calculs d'un programme séquentiel sont décomposés en tâches et chacune est affectée à un processus. Les instructions sont exécutées simultanément, ce qui peut entraîner un gain considérable en temps d'exécution. Une tâche importante dans une approche parallèle est l'affectation des travaux aux processus pour avoir un équilibrage de charge entre ces processus, afin

que tous les processeurs ou les noyaux aient la même charge ou presque à traiter. Dans ce qui suit, nous présentons deux algorithmes d'exploration en parallèles proposés pour avoir un gain en temps d'exécution.

4.4.1 Première contribution

Notre première solution [7] se base sur l'utilisation du framework Executor [28] qui consiste à découper la tâche en sous tâches et d'affecter chacunes d'elles à un processus (voir *Figure 4.3*). Ayant un modèle à explorer, l'algorithme se base sur la génération de l'état initial et de ses états successeurs. Par la suite, chaque état successeur sera effecté à un *thread* , le nombre de *threads* dépend du nombre de successeurs de l'état initial. Une sous tâche correspond à un état. Chaque *thread* explore une partie de l'espace d'états. La *Figure 4.4* montre les étapes d'exécution de l'approche parallèle proposée. Le traitement commence par la génération initiale de l'état, il est stocké dans l'ensemble S contenant tous les états visités et dans la file M qui contient les états dont les successeurs n'ont pas encore été observés. Par la suite, l'état initial est supprimé de la file et tous ses états suivants sont générés, puis pour chacun d'eux, un processus est lancé et cet état lui est attribué, ce qui signifie que le graphe d'accessibilité de cette configuration est créé par le *thread* . Chaque *thread* exécutera le code séquentiel produit par un processus unique. L'ensemble S est protégé par un verrou, car il est accessible par tous les *thread* s générés. À la fin de l'exploration, chaque processus renvoie l'ensemble des états observés, ce qui signifie que l'ensemble S est construit par tous les *threads* . L'algorithme 5 détaille le processus d'exploration

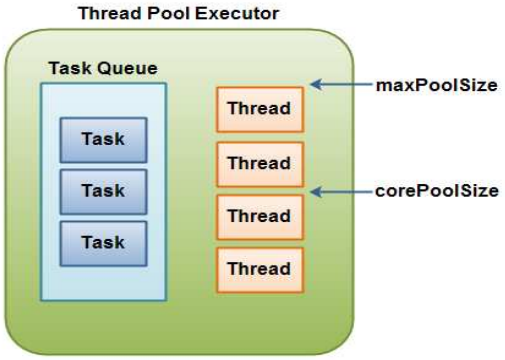

FIG. 4.3 – Principe du framework Executor

de l'algorithme proposé. Au début de l'exploration, l'état initial est généré, puis ses successeurs sont observés. Ensuite, pour chaque nouvelle configuration, un processus est créé (ligne 6 de l'algorithme 5). Ce dernier est actif jusqu'à ce que tous les états aient été traités. Chaque *thread* exécute une fonction pour l'exploration de l'espace d'états, donc les données utilisées sont locales, à l'exception de l'ensemble S dont l'accès est partagé. Prenons un exemple composé de deux

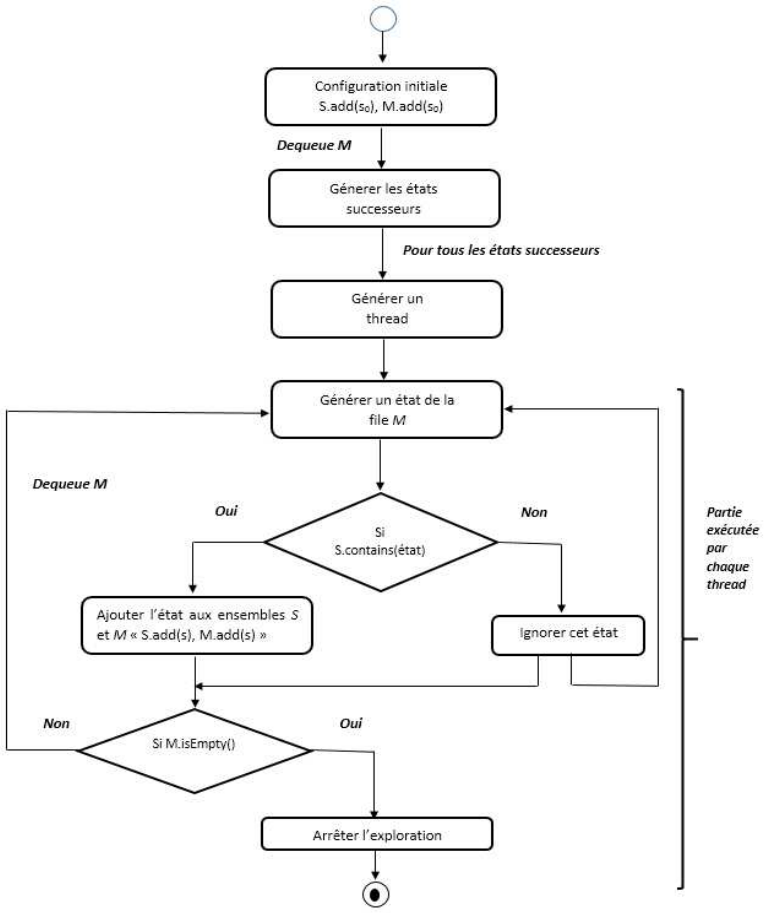

FIG. 4.4 – Etapes d'exécution de l'algorithme executor

compteurs qui sont incrémentés jusqu'à 3 et décrémentés à 0. La *Figure 4.5* présente un automate d'un compteur qui s'incrésente et se décrémente. L'état initial est généré en premier et stocké dans les ensembles S et M, par la suite ses successeurs <1 0> et <0 1> sont visités et stockés à leur tour dans ces deux ensembles. Après cela, deux *threads* sont lancés et chacun exécutera le code d'exploration à l'aide d'une file M contenant des états locaux dont les successeurs n'ont pas été observés. L'exploration s'arrête lorsque les deux files locales réservées aux *threads* générées sont vides. La *Figure 4.6* montre le graphe d'accessibilité généré par l'algorithme parallèle. À partir de cette figure, nous constatons que les deux processus génèrent l'état <1 1>, cet état est traité par l'un des deux processus qui explorera tous ses états successeurs.

Algorithm 5 Algorithme executor

```
1:  S ← S_0
2:  M ← S_0
3:  X ← M.dequeue()
4:  for allNextStates do
5:      Tasks.add(work)
6:      Executor.submit(Tasks)
7:  end for
8:  while ¬(M.isEmpty()) do
9:      X ← M.dequeue()
10:     Successors ← X.GetSuccessors()
11:     for (State K : Successors) do
12:         if ¬ (S.Contains(K)) then
13:             S.add(K)
14:             M.add(K)
15:         end if
16:     end for
17: end while
```

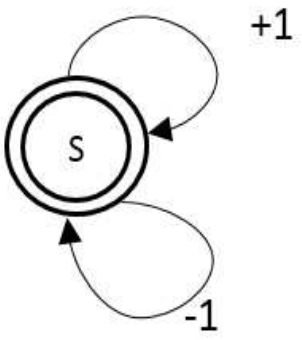

FIG. 4.5 – Automate d'un compteur qui s'incrémente puis se décrémente

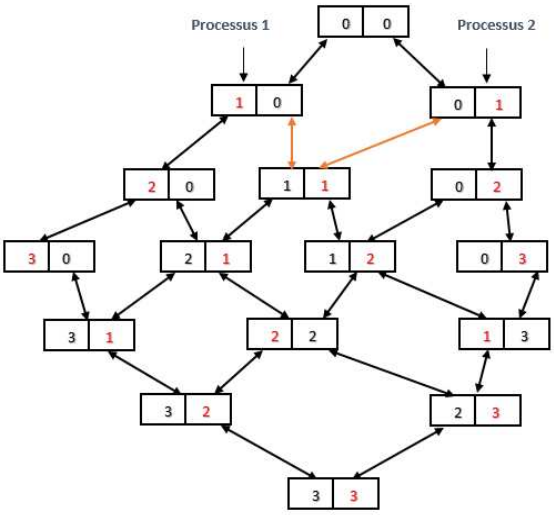

FIG. 4.6 – Exploration de deux compteurs en parallèle

4.4.2 Seconde contribution : Algorithme SPA

Une machine parallèle est essentiellement un ensemble de processeurs qui coopèrent et communiquent. Un algorithme parallèle s'exécute sur un ordinateur parallèle. Les instructions sont exécutées simultanément, ce qui peut entraîner un gain considérable en temps d'exécution. Une tâche importante dans une approche parallèle, est l'affectation du travail aux *threads* de tel sorte à respecter l'équilibrage de charge entre les *threads* .

L'algorithme parallèle synchronisé (SPA) [5] est basé sur l'utilisation d'un nombre fixe de *threads* et utilise 2 ensembles d'états : K et Q[i]. L'ensemble K, contient les états visités, cet ensemble est protégé par un verrou car il est partagé par tous les *threads* . Les états successeurs sont stockés dans l'ensemble Q[i], où i varie de 1 à N où N est le nombre de *threads* . Initialement, cet ensemble est vide. Q[i] est une file FIFO où chaque *thread* i traite les états stockés dans Q[i]. La taille de cette file est illimitée. Chaque entrée de cet ensemble représente un pointeur vers une file concurrente sans verrous contenant les états à traiter. A chaque fois qu'un état est généré, un *thread* est sélectionné aléatoirement et l'état est stocké dans sa liste. La fonction aléatoire a été utilisée, pour avoir un équilibrage de charge entre les *threads* . Nous avons utilisé la classe ThreadLocalRandom(), afin d'associer à chaque thread sa propre instance du générateur, améliorant ainsi les performances lors de son utilisation dans des traitements parallèles.

La *Figure 4.7* montre l'ensemble des opérations réalisées par les *threads* sur l'ensemble Q[i] contenant les états à traiter. À la première étape, le *thread* 1 génère l'état initial. Pour chaque état successeur, un *thread* est généré de manière aléatoire et l'état est stocké dans sa liste. Après

cette étape, chaque *thread* i où Q[i] n'est pas vide, retire un état de la file par la fonction de suppression d'états. Plusieurs *threads* peuvent exécuter cette fonction, car la file est concurrente. Les successeurs de cet état sont générés par la fonction de génération des successeurs. Après cela, un test est réalisé sur chaque état successeur, si l'état est nouveau, il est stocké de manière aléatoire dans l'ensemble Q[k] du *thread* k, en exécutant la fonction de stockage d'états, sinon il passe à l'état suivant. La vérification de la terminaison est déclenchée, pour vérifier si la fin de l'exploration a été atteinte, par la fonction de vérification de la terminaison. Si l'ensemble Q[i] pour i variant de 1 à N est vide, l'exploration se termine, sinon le processus est répété.

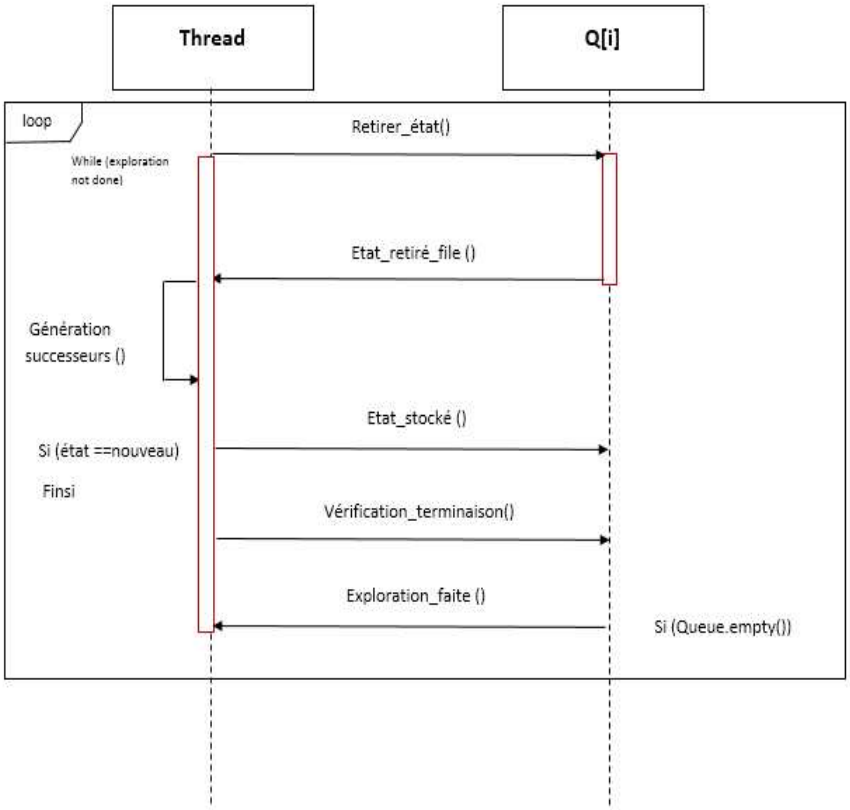

FIG. 4.7 – Opérations sur l'ensemble Q[i]

L'algorithme d'exploration parallèle est présenté ci dessous.
Au début de l'exploration, l'état initial est généré, puis ses successeurs sont observés. À ce moment, pour chaque nouvelle configuration, un *thread* est choisi de façon aléatoire (ligne 9 de l'Algorithme 6). Les données utilisées sont partagées entre les *threads*, la synchronisation est effectuée sur l'ensemble K contenant tous les états visités. Pour expliquer le processus d'exploration, nous avons pris un exemple de compteurs (voir *Figure 4.5*). La représentation d'un compteur qui est incrémenté et décrémenté, est réalisée à l'aide d'un automate déterministe composé d'un

seul état (initial et terminal). Un compteur est incrémenté jusqu'à une valeur val et peut être décrémenté jusqu'à la valeur de l'état initial. Si nous ajoutons un autre compteur, nous obtiendrons un vecteur d'état composé de 2 bits, donc, avec N compteurs, nous aurons un vecteur d'état composé de N bits. Par exemple, en fixant la valeur de val à 5 et ayant 6 compteurs, le nombre

Algorithm 6 Synchronized Parallel Algorithm (SPA)

```
1:  exploration_done ← false
2:  For each (w : 1..N)
3:  while ( !exploration_done) do
4:    for (each s in Q[w]) do
5:      delete s from Q[w]
6:      for (each successor s' of s) do
7:        if ((Synchronized) s' ∉ K) then
8:          (Synchronized) add s' to K
9:          w'= choose Random 1 .. N
10:         add s' to Q[w']
11:       end if
12:     end for
13:   end for
14:   if (w==1) then
15:     if (all Q[1..N] == NULL) then
16:       done ← true
17:     end if
18:   end if
19: end while
```

total d'états à explorer est égal à 46656 $((la\_valeur\_de\_val + 1)^{le\_nombre\_de\_compteurs})$. La *Figure 4.8* présente une partie du graphe d'accessibilité de 6 compteurs pouvant être incrémentés ou décrémentés à chaque étape. Pour chaque état exploré, un ensemble de successeurs est généré. La première étape consiste à générer l'état initial (0 0 0 0 0 0) par le *thread* 1. À partir de l'état initial, six états sont découverts, car à chaque transition, un compteur peut être incrémenté. Une transition d'un état à un autre se produit, lorsqu'un compteur est incrémenté ou décrémenté. L'état initial est stocké dans les deux ensembles K et Q[1], car l'état est nouveau, pour explorer ses prochaines configurations (états). Par la suite, tous ses successeurs sont visités et stockés à leur tour dans les deux ensembles, pour ce faire, nous devons choisir aléatoirement, un *thread* x pour chaque successeur généré. Cet état est stocké dans Q[x]. L'exploration s'arrête lorsque l'ensemble Q[i] est vide, i varie de 1 à N.

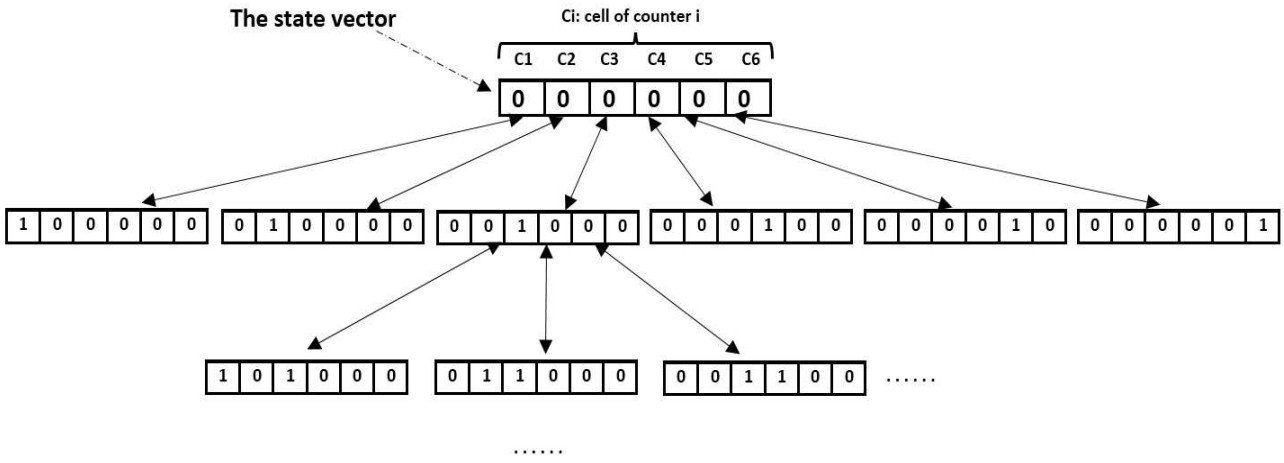

FIG. 4.8 – Une partie du graphe d'exploration de 6 compteurs qui s'incrémentent jusqu'à 5 et se décrémentent jusqu'à 0

4.4.3 Comparaison entre les deux approches parallèles

Les deux algorithmes parallèles présentés ci dessus visent à réaliser l'exploration d'un modèle en apportant un gain en temps d'exécution. Dans la première contribution, le nombre de *threads* générés va dépendre du nombre d'états successeurs de l'état initial. Si ce nombre est grand, le temps d'exécution augmentera à cause du nombre important de *threads* en mémoire causant un ralentissement de l'exécution, ce qui mènera vers une exploration partielle. Ceci se produit, car l'espace mémoire disponible est insuffisant pour réaliser l'exploration. Dans la deuxième solution, nous avons fixé le nombre de threads ce qui permet d'avoir une meilleure gestion de l'espace mémoire.

Afin de réaliser une comparaison avec une cette deuxième approche parallèle, nous avons proposé une approche distribuée, qui consiste à effectuer l'exploration dans un environnement composé de deux machines virtuelles dans un cloud computing [1].

Par définition, un algorithme est un ensemble d'instructions qui régit le déroulement d'un programme informatique. Un algorithme distribué se dit d'un algorithme qui est exécuté de manière simultanée sur un ensemble de ressources. Cette exécution, en simultanée sur plusieurs ressources distinctes, permet alors la réalisation d'un seul et même calcul. Le comportement de chaque processus est déterminé par un algorithme local et la communication entre les processus se fait par échange de messages uniquement. Dans cette section, nous présentons une exploration distribuée entre deux machines virtuelles dans un cloud computing. Le cloud computing représente l'exploitation de la puissance de calcul ou de stockage de serveurs informatiques distants par l'in-

[1] L'expérience a été réalisée à l'université d'Oran 1 Ahmed Ben Bella

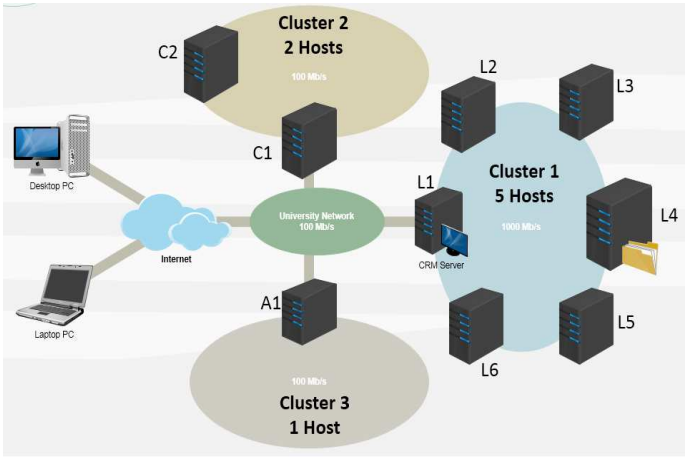

FIG. 4.9 – Topologie du cloud utilisé

termédiaire d'un réseau [11]. Nous avons utilisé deux machines virtuelles créées dans un même hôte. La communication se fait par sockets [59] à l'aide du langage de programmation Java. Chaque machine virtuelle exécutera le même algorithme d'exploration en distribué (voir Algorithm 7) et chaque machine se mettra en attente de configurations entrantes (l'algorithme de réception s'exécute en continu).

Le cloud est réparti sur 3 sites, l'accès aux ressources est transparent pour les utilisateurs. Le cloud est hétérogène par rapport aux ressources et à l'adressage ip (ipv 4, ipv 6). Le cloud est construit grâce au middleware VMware vCloud. La Topologie du cloud est présentée dans la *Figure 4.9* . Il est composé de 9 hôtes [4]. L1 est un serveur CRM pour la surveillance et la gestion des données. Les hôtes L2, L3, L6, C1, C2 et A1 sont utilisés pour le calcul. L2 contient 6 CPU avec une fréquence de 2 Ghz, 16 Go de mémoire physique et 550 Go d'espace de stockage. L3 contient 6 CPU fonctionnant à une fréquence de 2,1 Ghz, 48 Go de mémoire physique et 800 Go d'espace mémoire. L6, C1 et C2, contiennent respectivement 4 CPU avec 2,4, 2 et 2,4 Ghz respectivement et 8 Go de mémoire physique. A1 dispose de 6 processeurs d'une capacité de 2 Ghz. L'hôte L4 est utilisé pour le calcul, avec une capacité de stockage de 4096 Go. L5 est utilisé pour le calcul et le stockage avec 4 CPU opérant à une fréquence de 3.1 Ghz, 8 Go de mémoire physique et 2024 Go d'espace de stockage.

Les machines s'échangent des messages contenant des états. La communication se fait par sockets. Pour cela, un même numéro de port est utilisé entre les 2 machines. L'algorithme distribué est composé d'un code d'envoi et de réception d'états (l'algorithme de réception s'exécute en boucle), ainsi qu'une partie de code réalisant l'exploration des configurations (états) du modéle. L'exécution commence par l'établissement d'une connexion entre les 2 machines virtuelles. La *Figure 4.10* , illustre les étapes d'exécution de l'algorithme distribué. La machine VM1 (virtual machine 1), va jouer le rôle du serveur qui va initier la conversation. L'initiation consiste en la

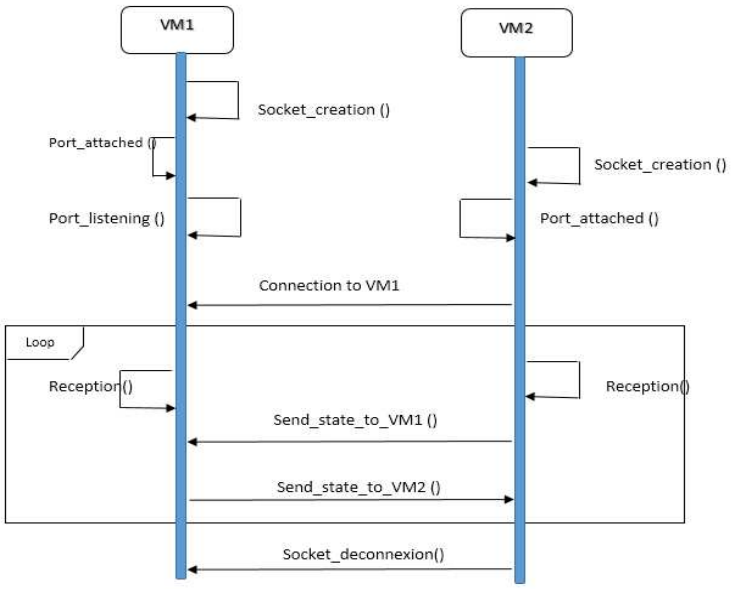

FIG. 4.10 – Communication distribuée utilisant 2 machines virtuelles

création du socket. Le serveur est lié à un numéro de port spécifique. Le serveur se met à l'écoute du client (VM2), qui établit une connexion. Aprés cela, un échange de messages est effectué entre les 2 entités. Les messages représentent les configurations échangées. La réception d'états se fait en continu jusqu'à ce qu'il n'y ait plus d'états à explorer. Chaque machine viruelle va explorer les états mis dans l'ensemble M, générer leurs successeurs et les stocker dans l'ensemble S. Au début de l'exploration, la machine VM1 va générer l'état initial et le stocker dans l'ensemble des états explorés S, ainsi que dans l'ensemble M pour que ses successeurs puissent être visités, puis, il va exécuter l'étape d'exploration qui est la suivante :

- Explorer les successeurs de l'état généré ;
- Vérifier pour chaque état si il a été exploré ou pas, en effectuant un test sur l'ensemble S ;
- Si l'état est nouveau, une machine sera choisi grâce à une fonction de hachage, pour explorer les successeurs de l'état, cet état sera stocké dans les deux ensembles de cette machine, sinon aucun traitement ne sera effectué sur cet état. Un état est composé d'un vecteur de bits. Chaque cellule de ce vecteur contient une valeur. La fonction utilisée consiste à additionner toutes les valeurs du vecteur puis de calculer le reste de la division de la somme obtenue par 2. Cette fonction est représentée sous la forme h(x)= $\sum$ valeurs_vecteur_états mod 2 . Si somme mod 2 est égale à 0, alors l'état est exploré par la machine virtuelle VM1 sinon l'état est traité par la machine virtuelle VM2 ;
- Extraire un état de l'ensemble M puis refaire le même traitement jusqu'à ce que l'ensemble M soit vide.

L'algorithme 3 est distribué ce qui fait que chaque machine exécute ce code et chacune à un ensemble S et un ensemble M. L'algorithme d'exploration est le suivant : La ligne 8 permet de

Algorithm 7 Algorithme d'exploration distribuée

```
1:  S ← S_0
2:  M ← S_0
3:  while ¬(M.isEmpty()) do
4:      X ← M.dequeue()
5:      Successors ← X.GetSuccessors()
6:      for (State K : Successors) do
7:          if ¬ (S.Contains(K)) then
8:              Num_process = hash(K);
9:              if (Process.currentProcess == Num_process) then
10:                 S_Num_process.add(K)
11:                 M_Num_process.add(K)
12:             else
13:                 Envoyer l'état à la machine virtuelle;
14:             end if
15:         end if
16:     end for
17: end while
18: Envoyer un message à la machine virtuelle indiquant la fin de l'exploration
```

définir la machine qui va stocker l'état afin d'explorer ses successeurs. Les instructions des lignes 10 et 11 permettent le stockage des états explorés dans les deux ensembles S et M. Tous les états de la file M sont explorés. L'exploration se termine lorsque M est vide. Les états échangés se font sur la base d'une fonction de hachage qui est calculée pour chaque état afin de savoir quelle machine va l'explorer. Lorsqu'une machine termine l'exploration des états, elle envoie une notification à la deuxième machine indiquant la fin de l'exploration. Cette dernière va stocker cette information et terminer l'exploration de ses états. Lorsque l'ensemble M sera vide, cette machine va déclencher la fin de l'exploration distribuée en fermant le socket.

L'algorithme 4 sert à la réception des messages entre les 2 VM. Le programme de réception tourne en continu à l'attente d'états. Pour l'émission et la réception, nous avons utilisé un *bufferedReader* ainsi qu'un *printStream* pour l'échange de données entre les machines. Le code d'envoi et de réception de données est défini par les *algorithmes 8 et 9* .

Algorithm 8 Algorithme de réception d'états

```
1: BufferedReader reader=new BufferedReader() ;
2: while (true) do
3:     State ⟵ reader.readLine() ;
4:     toSee.add(State) ;
5:     Known.add(State) ;
6: end while
```

Algorithm 9 Sending Algorithm

```
1: PrintStream print ⟵ new PrintStream() ;
2: print.println(state) ;
```

4.5 Conclusion

Le *Model checking* est une technique de vérification automatique de propriétés temporelles sur les systèmes réactifs. Il prend comme entrée, un système de transitions et une formule écrite en logique temporelle, et répond si le modèle satisfait sa spécification. Cette technique souffre du problème d'explosion en temps et en espace mémoire. Dans cet article, nous avons proposé deux approches parallèles et une approche distribuée pour l'exploration des modèles dont le but d'avoir un gain en temps d'exécution et par conséquent atteindre de meilleures performances. Afin de concrétiser nos propositions, nous avons implémenté nos 3 solutions à l'aide du langage de programmation Java. La description de cette implémentation fera l'objet du chapitre "expérimentations".

Chapitre 5

Expérimentations

Un problème sans solution est un problème mal posé.

Albert Einstein

Sommaire

5.1 Introduction

Après avoir abordé dans le chapitre précédent les différents points essentiels, l'étude conceptuelle et les objectifs attendus de nos travaux de recherche. Nous nous intéressons dans ce chapitre, à la description de l'environnement de travail qui nous a permis d'aboutir à des résultats intéressants. Afin de valider nos approches et ainsi comparer nos travaux parallèles avec une approche déjà proposée, nous avons réalisé l'étude expérimentale sur quelques modèles d'exclusion mutuelle, en variant quelques paramètres.

5.2 Environnement de développement

Java [12] est un langage de programmation informatique. *Java* a été créé par une équipe dirigée par *James Gosling* pour Sun Microsystems. C'est un langage de haut niveau, car il peut être lu et écrit facilement par les humains. Le développement des solutions a été réalisé sous Eclipse. Nous avons implémenté nos 2 contributions parallèles sur une machine ayant un processeur Intel i7, 8 cœurs avec une vitesse de 2 MHz, dotée d'une mémoire physique de 16 GB sous windows seven. La solution distribuée a été implémentée sur deux machines virtuelles, chaque VM est dotée de 21 GB de mémoire physique et de 256 GB d'espace de stockage.

5.3 Etude expérimentale

Nous avons présenté durant les travaux de recherche de cette thèse 3 approches pour le problème d'explosion en temps d'exécution. Afin de concrétiser nos approches, nous avons réalisé quelques expériences permettant de comparer entre les approches parallèles proposées et une approche [39] présentée dans le chapitre 2, permettant d'apporter un gain en temps d'exécution. La comparaison a été réalisée en utilisant un modèle composé de 6 compteurs qui s'incrémentent jusqu'à une valeur *val* et se décrémentent jusqu'à l'état initial. L'expérience a été réalisée en faisant varier le nombre de configurations de 2 jusqu'à 20 par pas de 2.
Nous avons effectué la comparaison en utilisant dans un premier temps l'algorithme parallèle executor. La *Figure 5.1*, présente la variation du temps d'exécution en fonction du nombre de configurations. Nous remarquons que l'approche proposée affiche de meilleurs résultats que l'approche présentée dans [39] pour un nombre de configurations variant de 729 à 85766121. Le nombre de *threads* dans l'approche Spin a été fixé à 6 afin d'avoir le même nombre de *threads* dans les 2 approches. Le gain obtenu avec notre solution est dû au fait qu'il n'y a pas d'attente/réveil entre les *threads*, par contre, dans l'approche de Spin, un processus qui termine son traitement attend que tous les autres processus aient terminés leurs traitements pour passer à l'exploration des états suivants (futurs). Notre approche proposée affiche de bons résultats si le nombre de

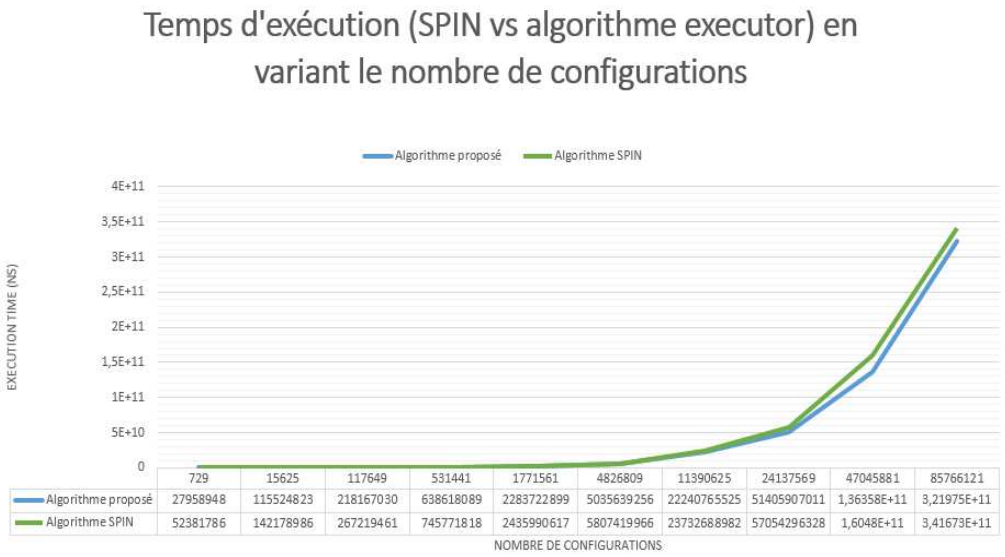

FIG. 5.1 – Temps d'exécution des deux approches en faisant varier le nombre de configurations

threads générés est minimal, donc, pour un modèle construit en largeur dont l'état initial possède plusieurs successeurs, plusieurs *threads* seront générés.

Dans la *Figure* 5.2 , nous présentons le résultat de l'exploration réalisée par les deux approches Spin et explorator sur un modèle composé de 10 compteurs qui s'incrémentent de 2 jusqu'à 5 par pas de 1. Nous pouvons remarquer d'après cette expérience, que le temps d'exécution pour notre approche augmente rapidement en variant le nombre d'états, à cause du nombre de *threads* générés.

Dans la deuxième approche parallèle, nous avons réalisé des expériences sur 4 modèles différents, 3 modèles du benchmark BEEM [51] et un modèle de compteurs. Cette base de données est composée de plusieurs modèles classés en 4 catégories :

- Exclusion mutuelle
- Protocoles de communication
- Algorithmes d'élection de leader
- Planification et ordonnancement

Nous avons utilisé 3 modèles d'exclusion mutuelle, la spécification des quatre modèles a été réalisée sous *Java* . La génération des *threads* dans *Java* a été réalisée en utilisant la classe *thread* . Le nombre de *threads* est fixé au moment de l'exécution. Une exécution consiste à explorer un modèle, à la fin de l'exploration, le temps d'exécution est retourné ainsi que le nombre d'états explorés K. La taille de cet ensemble est égale au nombre d'états. Tous les états sont stockés en mémoire. Le nombre de *threads* pour effectuer l'exploration est le même pour les deux algorithmes dans les quatre expériences.

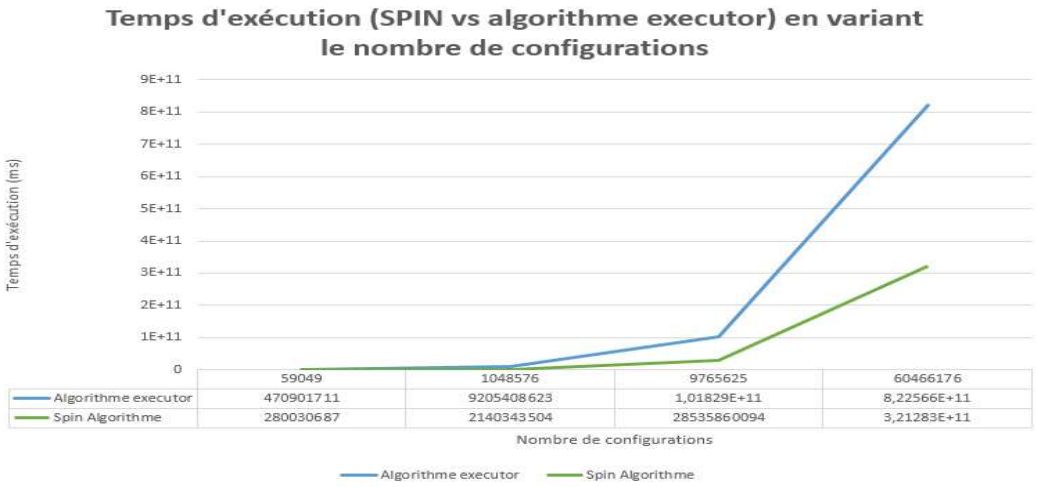

FIG. 5.2 – Temps d'exécution des deux approches en faisant varier le nombre de configurations

1. **Expérience 1** : Modèle de *Peterson* . L'algorithme de *Peterson* [53] est un algorithme d'exclusion mutuelle pour la programmation concurrente, il consiste en deux phases : la phase d'entrée en section critique et la phase de sortie de celle-ci. *L'algorithme de Peterson* se compose de N processus voulant entrer en section critique. Les étapes d'exécution de ce modèle sont définies dans l'aglorithme. Chaque processus passe par plusieurs niveaux afin d'atteindre la section critique (ligne 4). *processus_dernier* est une structure de données contenant le numéro de processus se trouvant à chaque niveau. Lorsqu'un processus atteint le niveau $N-1$, il entre en section critique.

Algorithm 10 Algorithme de Peterson

1: niveau_processus : tableau de N entiers
2: processus_dernier : tableau de N entiers
3: chaque processus i
4: **for** n de 0 à $N-1$ **do**
5: $niveau\_processus[i] \leftarrow n$
6: $processus\_dernier[n] \leftarrow i$
7: **while** (processus_dernier[n]==i and $\exists k \neq i$ and $niveau\_processus[k] \geq n$) **do**
8: wait
9: **end while**
10: **end for**

Nous avons réalisé 6 tests différents en variant le nombre de processus concurrents vers la section critique. Nous avons fixé ce nombre à 5 pour le premier test puis à 10, 15, 16, 17, 18 respectivement pour les tests suivants. Le *Tableau 5.1* et la *Figure 5.3* présentent le

temps d'exécution obtenu par les deux algorithmes (SPA et SPIN) en effectuant l'analyse de l'accessibilité sur les 6 tests. Le nombre d'états est obtenu à la fin de chaque test, ce nombre évolue en augmentant le nombre de processus concurrents.

Pour ce premier modèle, le nombre de configurations varie entre 352 et 40370176. Le nombre d'états est égal à 352 et le nombre de *threads* (processus machine) est fixé à 2 pour le premier test. Pour le dernier test, le nombre d'états et 40370176 et le nombre de *threads* générés est équivalent à 8. La machine sur laquelle les expériences ont été réalisées possède 8 cœurs, donc nous avons généré 8 *threads* pour le dernier test afin d'exploiter toutes les ressources de la machine. D'après la *Figure* 5.3 , nous remarquons que l'algorithme SPA est plus performant que l'algorithme de Spin dans l'étape d'exploration de ce modèle.

Afin d'interpréter ces résultats, nous avons calculé le gain obtenu par notre approche à

TAB. 5.1 – Temps d'exécution, nombre de configurations et de processus pour les deux algorithmes

Processus	**Configurations**	**temps d'exécution SPA (secondes)**	**Temps d'exécution SPIN (secondes)**
5	352	0.017	0.022
10	47104	0.20	0.21
15	3473408	9	11
16	7929856	29	33
17	17956864	91	95
18	40370176	279	327

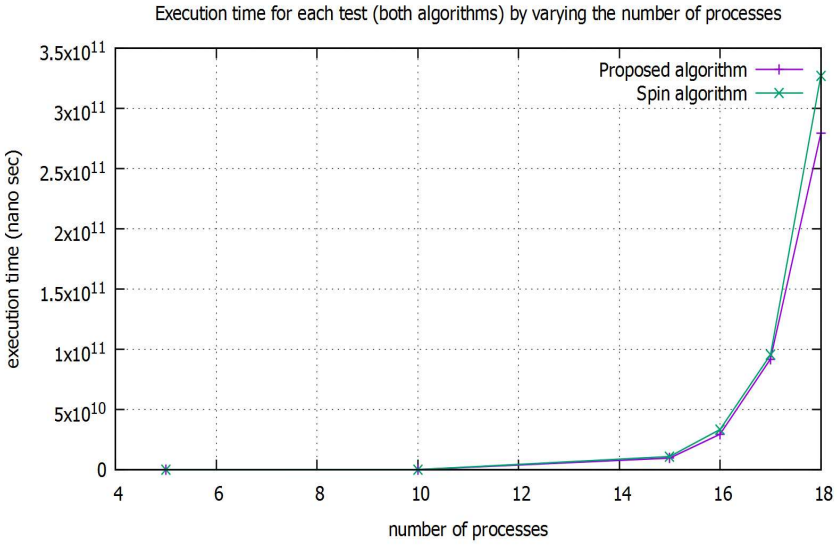

FIG. 5.3 – Temps d'exécution des deux approches en faisant varier le nombre de processus (modèle de Peterson)

chaque test. Le résultat en pourcentage est le suivant : 21.45, 3.31, 12.10, 11.75, 3.64 et 14.63. Le gain moyen de ces 6 test est de 11.15%. Cela est dû au déroulement de l'algorithme de Spin qui alterne entre traitement/attente/réveil entre les *threads* ce qui va augmenter le temps d'exécution, aussi, dans notre algorithme la file contenant les états non visités est une structure de donnée concurrente qui gère la synchronisation entre les *threads* , donc il n'y a pas de synchronisation au niveau de la file.

2. **Expérience 2** : Modèle des philosophes. Le dîner des philosophes [17] est une méthode souvent utilisée dans la conception d'algorithmes concurrents pour illustrer les problèmes de synchronisation et les techniques pour les résoudre. L'algorithme des philosophes est présenté dans *l'Algorithme 11* . Si tous les philosophes prennent en même temps leur fourchettes , il y a interblocage. Un philosophe entre en section critique si sa fourchette droite et sa fourchette gauche sont disponibles. Nous avons réalisé la spécification de ce modèle sous *Java* ensuite nous avons effectué l'exploration des états. Nous avons réalisé 5 tests, les expériences ont été exécutées sur la même machine. Nous avons varier le nombre de processus (philosophes) de 11 à 15 par pas de 1 et à chaque test, nous avons estimé le temps d'exécution par les deux algorithmes (SPA et SPIN). Le résultat est présenté dans le *Tableau 5.2* et la *Figure 5.4* .

   Durant l'exploration de ce modèle, tous les états possibles sont visités. Le nombre de configurations augmente en augmentant le nombre de processus concurrents. Nous avons calculé le gain moyen de ces 5 tests apporté par notre approche parallèle qui est estimé à 10%. Le gain augmente à chaque test en augmentant le nombre de processus concurrents, donc notre algorithme reste performant en terme de passage à l'échelle.

Algorithm 11 Algorithme des philosophes

```
1: Penser()
2: while (Fourchette_droite == faux || Fourchette_gauche == faux) do
3:     wait()
4:     Fourchette_droite = faux
5:     Fourchette_gauche = false
6:     Manger ()
7:     Fourchette_droite = true
8:     Fourchette_gauche = true
9: end while
```

TAB. 5.2 – Temps d'exécution, nombre de configurations, nombre de processus

Processus	Configurations	Temps d'exécution SPA (secondes)	Temps d'exécution SPIN (secondes)
11	393660	0.86	0.88
12	1240029	2	3
13	3897234	11	14
14	12223143	44	51
15	38263752	249	266

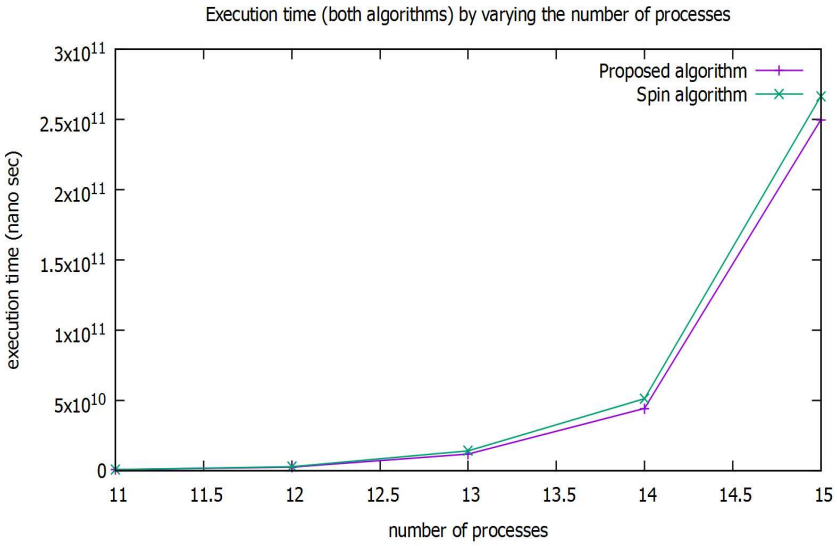

FIG. 5.4 – Temps d'exécution des deux approches en faisant varier le nombre de processus (modèle des philosophes)

3. **Expérience 3** : Modèle des producteurs/consommateurs. Le modèle des producteurs/consommateurs [40] est un exemple classique d'un problème de synchronisation. Le problème décrit deux processus, le producteur et le consommateur, qui partagent un buffer commun utilisé comme file d'attente. Le travail du producteur consiste à générer des données, à les mettre dans le buffer et à recommencer. Dans un même temps, le consommateur consomme les données (c'est-à-dire en les supprimant du buffer), une donnée à la fois. Le problème est de s'assurer que le producteur n'essaye pas d'ajouter des données dans le buffer s'il est plein et que le consommateur n'essaye pas de supprimer les données d'un buffer vide. les étapes d'exécution de ce modèle sont présentées dans les *Algorithmes 12 et 13* . Nous avons modélisé le comportement de chaque producteur et consommateur. Pour les expériences, nous avons utilisé un modèle composé de plusieurs producteurs et d'un seul consommateur. Nous avons effectué 5 tests en variant le nombre de producteurs et la taille du buffer. La comparaison a été réalisée sur le temps d'exécution estimé par les deux algorithmes parallèles (SPA

et SPIN). Le résultat de cette expérience est présenté dans les *Tableaux* 5.3 et 5.4 et la *Figure* 5.5 . Dans le premier test, nous avons utilisé un modèle composé de 600 producteurs, la taille du buffer a été fixée à 10000. D'après la *Figure* 5.5 , l'exploration du modèle des producteurs/consommateur en utilisant notre algorithme donne de meilleurs résultats en temps d'exécution. Nous avons calculé le gain moyen du temps qui est estimé à 8.62%.

Algorithm 12 Algorithme du producteur

```
1: while (true) do
2:     while (buffer.length == taille_tableau_déclaré) do
3:         wait
4:     end while
5:     Déposer_message_buffer()
6: end while
```

Algorithm 13 Algorithme du consommateur

```
1: while (true) do
2:     while (buffer.isEmpty == true) do
3:         wait
4:     end while
5:     Consommer_message_buffer()
6: end while
```

TAB. 5.3 – Temps d'exécution et nombre de configurations pour les 2 approches parallèles

Configurations	**Temps d'exécution SPA (secondes)**	**Temps d'exécution SPIN (secondes)**
6020000	68	70
12040001	146	162
16040001	260	278
30060001	586 ($\simeq 10min$)	656 ($\simeq 11min$)
20020001	777 ($\simeq 13(min)$)	885 ($\simeq 15(min)$)

TAB. 5.4 – Paramètres de l'exécution et le nombre d'états à chaque test

Producteurs	Taille du buffer(cellule)	Nombre de configurations
600	10000	6.020.000
600	200000	12.040.001
800	20000	16.040.001
1000	30000	30.060.001
2000	20000	20.020.001

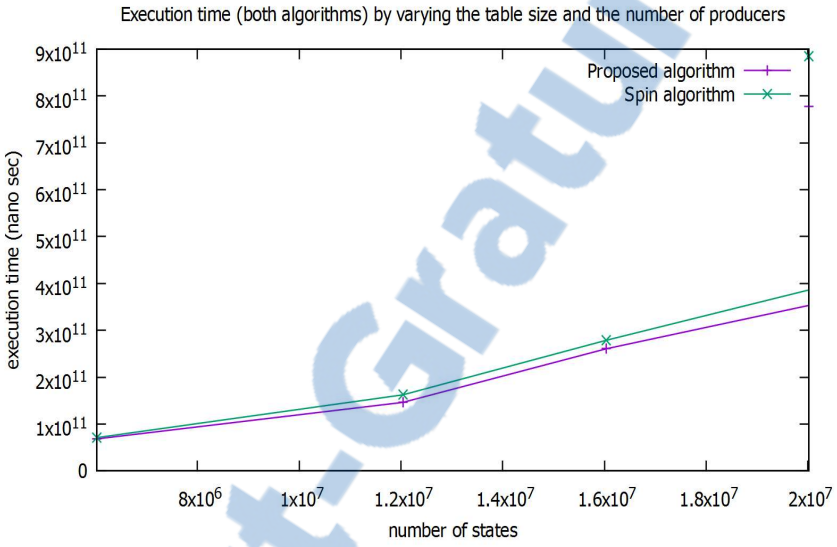

FIG. 5.5 – Temps d'exécution des deux approches en faisant varier le nombre de producteurs et la taille du buffer (modèle des producteurs/consommateurs)

4. **Expérience 4** : Modèle des compteurs. Nous avons réalisé une comparaison entre les deux algorithmes parallèles SPA et Spin. Nous avons effectué 6 tests en utilisant un modèle composé de 5 compteurs. Les compteurs s'incrémentent jusqu'à une valeur maximale fixée lors de l'exécution et se décrémentent jusqu'à l'état initial. La valeur maximale val_max a été fixée à 22 pour le premier test et à 39 pour le dernier test. A chaque test, on incrémente cette valeur par pas de 3. Pour calculer le nombre d'états, il faut avoir comme données le nombre de compteurs ainsi que la valeur maximale val_max. Ce nombre est calculé comme suit : Nombre_config = $(val\_max + 1)^{numbre\_de\_compteurs}$. Le résultat de l'exploration présenté dans le *Tableau 5.5* et la *Figure 5.6*, montre que notre algorithme est plus performant que l'algorithme de Spin en temps d'exécution. Le gain apporrté par notre approche lors ce cette expérience est de 10%.
Nous remarquons dans le dernier test, que la limite de l'expérience parallèle par Spin est fixée à val_max égale à 38 (90.224.199 configurations). Au delà de cette valeur, l'explora-

tion est confrontée au problème d'explosion combinatoire en espace mémoire. L'application *Java* s'arrête en affichant à l'écran le message "out of memory". Les ressources en espace mémoire sont exploitées au maximum. Pour ce qui est de l'exécution parallèle en utilisant notre algorithme, et, en fixant le *heap* [2] de l'application *Java* au même paramètre que celui de l'exécution en utilisant l'exploration parallèle, plus d'états sont explorés, correspondants à 102.400.000 configurations. D'après ce résultat, nous pouvons constater que notre algorithme, explore plus d'états, à cause des structures des données utilisées : nous avons utilisé des files concurrente, l'accès à cette structure se fait en parallèle sans utiliser de verrous. un autre avantage de notre algorithme est qu'il n' y a pas d'attente/ réveil entre les threads.

TAB. 5.5 – Temps d'exécution, nombre de configurations, et val_max pour les deux algorithmes

val_max	**Configurations**	**Temps d'exécution SPA (secondes)**	**Temps d'exécution SPIN (secondes)**
22	6436343	11	13
25	11881376	24	27
28	20511149	43	48
31	33554432	80	86
34	52521875	139	164
37	79235168	268	285
38	90224199	302	412
39	102400000	880	out of memory

Nous avons présenté ci-dessous, les expériences réalisées par nos deux approches parallèles. Dans ce qui suit, nous étudions l'approche distribuée, les expériences ont été réalisées sur un cloud composé de 2 machines virtuelles s'échangeant des configurations afin d'effectuer l'exploration d'un modèle donné. Nous avons effectué deux expériences, la première consiste en une comparaison entre une exploration utilisant deux machines physiques et une exploration dans un environnement de cloud. Dans la deuxième exploration, nous avons réalisé une étude comparative entre l'algorithme distribué dans le cloud et l'algorithme SPA. Pour les deux expériences, nous avons réalisé les tests sur un modèle de compteurs variant de 0 jusqu'à une valeur maximale fixée lors des expériences. Un compteur s'incrémente jusqu'à une valeur désirée et peut se décrémenter jusqu'à 0.

– **Expérience 1** : Nous avons effectué une comparaison entre deux exécutions similaires sur deux architectures différentes. La première [8], concerne une exécution utilisant 2 machines physiques, l'une disposant de 16 giga de mémoire et l'autre de 4 giga. La seconde [4], concerne une exécution dans un cloud computing. Pour cela, nous avons utilisé deux machines virtuelles disposant chacunes de 20 giga de mémoire. La communication dans

[2] le heap représente une zone mémoire partagée par tous les threads de la JVM : elle stocke toutes les instances des objets créés.

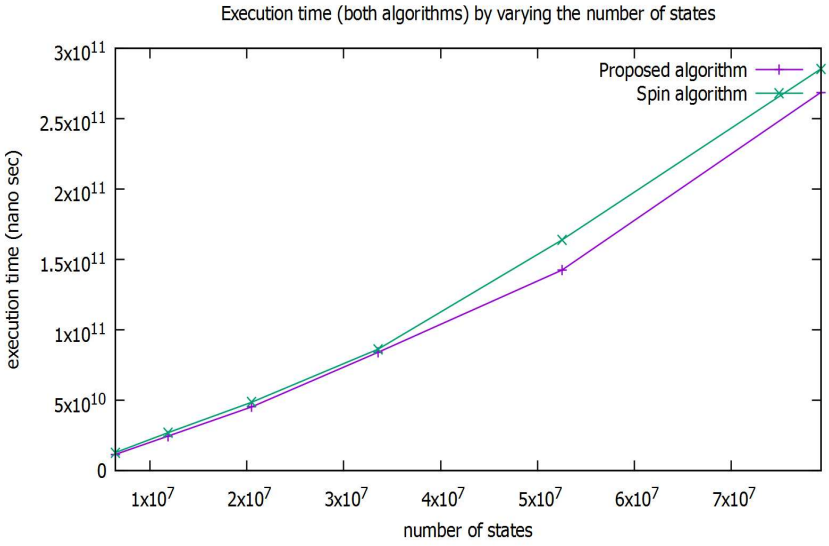

FIG. 5.6 – Temps d'exécution des deux approches en faisant varier le nombre de configurations (modèle de compteurs)

les 2 architectures se fait par sockets sous *Java* . Toutes les configurations du modèle sont parcourues [6].

Cette première expérience concerne l'exploration d'un modèle composé de 5 compteurs qui s'incrémentent de 17 jusqu'à 27 par pas de 1. Tous les états sont visités, en commençant par l'état initial. La *Figure 5.7* présente le temps d'exécution nécessaire pour réaliser l'exploration de cette première expérience (la valeur maximale varie de 17 à 27). Nous remarquons que pour le même nombre d'états, l'exécution dans une architecture de cloud computing donne de meilleurs résultats. Cela s'explique par l'absence d'interférences car le réseau qui relie les deux VM est virtuel. Le *Tableau 5.6* illustre les résultats en temps d'exécution (en minutes) pour chaque expérience.

Nous remarquons une grande différence de temps entre les deux exécutions, à cause du réseau, de sa fiabilité, de la bande passante et des interférences. Le gain moyen de l'approche distribuée dans le cloud, comparativement à l'approche distribuée utilisant une communication entre deux machines réelles est de 15 minutes.

– **Expérience 2** : Dans ce qui suit, nous présentons une étude comparative entre deux algorithmes : parallèle (SPA) et distribuée dans le cloud. Dans cette expérience, nous avons effectué des tests en utilisant le même modèle de compteurs (5 compteurs), et en faisant varier la valeur val_max de 30 à 39 (voir *Tableau 5.7*).
La *Figure 5.8* présente le temps nécessaire pour l'exploration d'un modèle de compteurs, variant de 28.629.151 à 102.400.000 configurations. Nous remarquons que nous obtenons

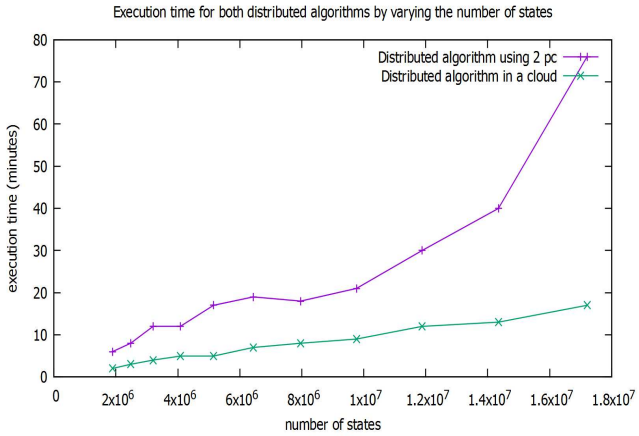

FIG. 5.7 – Temps d'exécution pour les deux algorithmes distribués en variant le nombre de configurations

TAB. 5.6 – Temps d'exécution pour les deux approches distribuées

val_max	**Configurations**	**Temps d'exécution utilisant 2 nœuds (min)**	**Temps d'exécution dans un cloud computing (min)**
17	1889568	6	2
18	2476099	8	3
19	3200000	12	4
20	4084101	12	5
21	5153632	17	5 min and 28 sec
22	6436343	19	7
23	7962624	18	8
24	9765625	21	9
25	11881376	30	12
26	14348907	40	13
27	17210368	76	17

un gain remarquable en temps d'exécution par l'approche parallèle, à cause du temps d'accès aux données se trouvant dans la mémoire partagée, comparativement avec l'approche distribuée, où l'échange se fait par envoi de messages à travers le réseau. L'approche distribuée est efficace pour des calculs intensifs, seulement, plusieurs facteurs peuvent ralentir l'exéution, par exemple, la qualité de service du réseau (bande passante et débit) qui peut ralentir l'envoi/réception de messages, ainsi que la panne d'une machine qui interrompt l'exploration.

TAB. 5.7 – Temps d'exécution pour les deux approches : parallèle et distribuée

val_max	**Configurations**	**Temps d'exécution dans le cloud (min)**	**Temps d'exécution par l'algorithme SPA (min)**
30	28629151	6	1
31	33554432	8	2
32	39135393	12	2
33	45435424	17	2
34	52521875	19	3
35	60466176	18	3
36	69343957	21	4
37	79235168	30	5
38	90224199	40	6
39	102400000	70	14

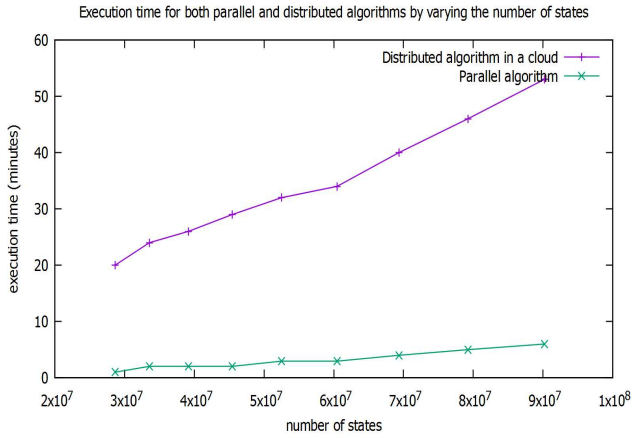

FIG. 5.8 – Temps d'exécution pour les deux algorithmes parallèle et distribué en variant le nombre de configurations

5.4 Espace mémoire utilisé à l'étape d'exploration

La vérification des modèles est une technique basée sur la vérification de l'exactitude d'un système, par rapport à un ensemble de propriétés. L'exploration consiste à explorer chaque état (nous devons itérer l'algorithme plusieurs fois), donc, avoir des systèmes composés d'un grand nombre d'états, qui ont tendance à croître de façon exponentielle, dans le nombre de ses processus et variables, ce cas conduit à un état d'explosion combinatoire. Dans cette section, nous avons analysé l'espace mémoire utilisé dans les deux algorithmes parallèle (SPA et SPIN), en comptant la taille des structures de données utilisées.

– **Espace mémoire utilisé par l'algorithme SPA :** Une valeur entière est spécifiée dans le code source de n'importe quel programme en tant que séquence de chiffres. Habituellement, les variables sont stockées sur 32 bits, donc, pour analyser l'espace mémoire utilisé, il faut

compter le nombre de structures de données déclarées dans l'algorithme. Les structures de données spécifiées dans l'algorithme sont listées dans le *Tableau 5.8*. *Exploration_done* est une variable booléenne, elle est stockée sur 32 bits, nous avons N pointeurs de la file d'attente Q, N se réfère au nombre de *threads* et chaque $Q[i]$ (i varie de 1 à N) pointe vers une file concurrente. Le nombre d'états successeurs est inconnu. Nous avons N files, par conséquent, la mémoire utilisée pour le stockage des états dans la file d'attente Q est 32 * s * N (s est le nombre maximum d'états dans une liste de *thread*). Un ensemble K contient les états explorés. La taille de K est déterminée par : (32 + 32) * t, avec t le nombre total d'états explorés. En ayant ces informations, nous pouvons estimer le l'espace mémoire utilisé par notre approche, exprimé en bits :

$$\begin{aligned} Memory\_used_{SPA} &= 32 + (32 * N) + ((32 * s * N) + ((32 + 32) * t) \\ &= 32 * (1 + N + (s * N) + (t * 2)) \\ &= 32 * (1 + N(1 + s + (t * 2)). \end{aligned}$$

TAB. 5.8 – Structures de données utilisées dans l'algorithme SPA

Structures de données	**Espace mémoire utilisé**
exploration_done	32 bits
pointeur de la file Q	32 bits
file concurrente	32 * s bits
K (ensemble des états explorés)	(32 + 32) * t bits (clés et données)

– **Espace mémoire utilisé par l'algorithme de Spin :** Dans l'algorithme parallèle développé pour le vérificateur de modèle *Spin*, 2 ensembles sont spécifiés, Q (file à 3 dimensions) et S (ensemble des états atteints). Les structures de données spécifiées dans l'algorithme sont listées dans le *Tableau 5.9*. *Done* est une variable booléenne qui indique si tous les

TAB. 5.9 – Structures de données utilisées dans l'algorithme de SPIN

Structures de données	**Espace mémoire utilisé**
done	32 bits
t	32 bits
pointeur de la file Q	32 bits
liste des pointeurs vers les listes	32 * N bits
liste chaînée (pour un processus)	(32 + 32) * s * N bits
S (ensemble des états explorés)	(32 + 32) * t bits (clés et données)

états ont été atteints ou pas. La variable t, est stockée sur 32 bits. Il y a N pointeurs de la file Q, chacun d'eux pointe vers une liste de pointeurs (N pointeurs), en liaison avec des listes chaînées. Chaque *thread* maintient N listes d'états, l'espace mémoire utilisé pour le stockage des états dans la file Q est (32 + 32) * s * N (s est le nombre maximum d'états

dans une liste d'un *thread*). L'espace mémoire est alloué pour le stockage des états actuels et futurs. Par conséquent, l'espace est alloué pour Q[0][i][j] et Q[1][i][j].

La taille de S est déterminée par : (32 + 32) * t (t est le nombre total d'états explorés).

Nous pouvons estimer l'espace de stockage utilisé par l'algorithme parallèle proposé dans [38] en bits comme suit :

$$\begin{aligned} Memory\_used_{SPIN} &= 32+32+(2*32)+2*(32*N)+(2*N)*((32+32)*s*N)+((32+32)*t) \\ &= 32*(1+1+2+(2*N)+(4*s*N^2)+(t*2)) \\ &= 32*(4+N*(2+(4*s*N))+(t*2)). \end{aligned}$$

Dans l'analyse de l'accessibilité, on cherche toujours à manipuler le moins de données possible, afin de gagner en espace mémoire. À partir de cette analyse, on peut conclure que notre algorithme, utilise moins de mémoire que l'algorithme parallèle développé pour $Spin$, ce qui permet de confirmer le résultat pour l'exploration d'un modèle de 5 compteurs s'incrémentant jusqu'à 39.

5.5 Conclusion

Dans ce présent chapitre, nous avons proposé 2 solutions parallèles et une solution distribuée pour l'exploration de l'espace d'états visant à apporter un gain en temps d'exécution. Nous avons réalisé quelques expériences pour comparer entre deux algorithmes parallèles (SPA et SPIN). Les résultats obtenus montrent un meilleur comportement de nos algorithmes en tenant compte du nombre de processus et du nombre de configurations, ainsi qu'une meilleure manipulation des données en mémoire, car il y a moins d'instructions à exécuter par nos algorithmes.

Chapitre 6

Conclusion générale

La vie, c'est comme une bicyclette, il faut avancer pour ne pas perdre l'équilibre.

Albert Einstein

.

Notre dépendance de la vie quotidienne à l'égard des applications informatiques (matérielles et logicielles) a motivé les chercheurs en informatique à développer de nouvelles techniques pour accroître notre confiance en leur justesse. Beaucoup de ces applications sont sans aucun doute critiques et une défaillance peut causer des dommages importants, tant sur le plan économique que sur le plan physique. Le 4 juin 1996, la fusée européenne Ariadne a explosé quelques secondes après son lancement, coûtant des millions de dollars et de nombreuses années de travail. Une enquête a révélé que l'erreur aurait pu être évitée grâce à l'utilisation de techniques formelles pour détecter l'erreur logicielle qui a causé cette catastrophe.

Comment peut-on être sûr de la justesse des systèmes critiques, avec des milliers (et parfois des millions) de composants interagissant de manière complexe ? Dans certains cas, il est possible de construire le système considéré, puis d'injecter au système des données sensibles pour vérifier qu'il se comporte réellement comme il est supposé le faire. C'est le principe du "test". Une telle approche est largement utilisée et extrêmement utile dans la pratique, bien qu'il ne soit clairement pas possible de l'utiliser dans des systèmes hautement critiques, si les données de test pouvaient causer des dommages en cas d'erreurs avant le déploiement réel.

Une autre solution consiste à simuler le comportement du système sur un ordinateur. La simulation ne fonctionne pas directement sur le système réel, mais sur un modèle. Un modèle est une représentation abstraite du système réel, généralement écrit en mathématiques ou en logique. L'un des avantages de la simulation est que l'on n'a pas besoin de construire le système réel pour être appliquée.

Les tests et la simulation sont très répandus dans les applications industrielles et leur utilisation s'est révélée très utile. Un inconvénient, cependant, est qu'il n'est pas possible, en général, de simuler ou de tester tous les scénarios ou comportements possibles d'un système donné. C'est-à-dire que ces techniques ne sont généralement pas exhaustives, en raison du grand nombre de cas possibles à prendre en compte, et que les cas de défaillance peuvent être parmi ceux non testés ou simulés.

Cette thèse porte sur la vérification formelle des systèmes informatiques, en particulier sur la technique du model checking qui représente une autre technique formelle. Cette vérification repose sur une idée simple : si on énumère toutes les situations possibles auxquelles peut mener le programme, on pourra s'assurer qu'aucune de ces situations n'est en contradiction avec les comportements désirés. Seulement, il est impossible d'effectuer une énumération exhaustive d'un nombre d'états infini, car la représation de tous les états possibles du système conduit à un dépassement des capcités de la machine pour stocker tous ces cas possibles. Ce problème est connu sous le nom d'explosion combinatoire de l'espace d'états.
Dans cette recherche, nous avons étudié le temps d'exécution nécessaire pour effectuer l'exploration d'un modèle, car si nous avons à vérifier plusieurs états, le temps d'exploration sera très grand ce qui nous conduit à traiter l'explosion en temps d'exécution.

Dans le chapitre 2, nous avons décrit un état de l'art sur quelques solutions au problème d'explosion en espace mémoire.

Le chapitre 3 constitue l'étude abordée dans cette thèse. Nous avons proposé deux algorithmes parallèles et un algorithme distribué, en utilisant 2 machines qui communiquent par échange de messages. Les résultats montrent qu'en utilisant l'approche parallèle, nous avons de meilleurs résultats en temps d'exécution à cause de la communication entre threads qui se fait par le biais de la mémoire partagée de la machine. En réalisant des expériences avec les deux algorithmes parallèles SPA et SPIN, nous avons remarqué que notre algorithme était plus rapide que l'algorithme de Spin d'une part, et que pour un certain nombre d'états, notre algorithme aboutit à la fin de l'exploration, alors qu'avec l'algorithme de Spin, la machine atteint ses capacités mémoire et l'exploration ne se termine pas. Afin de confirmer ce résultat, nous avons calculé une estimation de l'espace mémoire nécessaire pour la manipulation des structures de données durant la phase d'exploration. Nous pouvons conclure que le choix des structures de données utilisées est important afin d'aboutir à un temps d'accès minimal. L'exploration distribuée est très utilisée pour la manipulation de plusieurs données, des traitements intensifs qui nécessitent un large espace mémoire, seulement, le temps de traitement est trés grand.
Les résultats expérimentaux sont détaillés dans le chapitre 4.

Les perspectives envisageables en prolongement direct de cette thèse sont : l'exécution de l'algorithme distribué sur plusieurs machines en gérant les échanges de messages entre les différents nœuds, l'exécution de notre explorateur SPA sur des modèles réels, ainsi que l'intégration de cet explorateur dans un outil de vérification. Nous proposons aussi d'étendre notre recherche, en prenant en compte la spécification de l'environnement qui itéragit avec le système, cette extension va permettre d'avoir une vérification plus affrofondie lors de la phase de vérification des propriétés sur le modèle.

Bibliographie

[1] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3) :507–535, May 1995.

[2] G. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences*, volume 45, pages 1259–1263, 1962.

[3] L Allal, G Belalem, and P Dhaussy. Towards distributed solution to the state explosion problem. In *Satapathy S., Mandal J., Udgata S., Bhateja V. (eds) Information Systems Design and Intelligent Applications.*, volume 433. Springer, New Delhi, 2016.

[4] L Allal, G Belalem, P Dhaussy, and C Teodorov. Distributed algorithm to fight the state explosion problem. *International Journal of Internet Technology and Secured Transactions, in press* .

[5] L. Allal, G. Belalem, P. Dhaussy, and C. Teodorov. A parallel algorithm for the state space exploration. *Scalable Computing : Practice and Experience*, 17(2) :129–142, 2016.

[6] L Allal, G Belalem, P Dhaussy, and C Teodorov. Proposed algorithms to the state explosion problem. In SmartCom 2016, editor, *In : Unal A., Nayak M., Mishra D., Singh D., Joshi A. (eds) Smart Trends in Information Technology and Computer Communications*, volume 628. Springer, Singapore, 2016.

[7] L Allal, G Belalem, P Dhaussy, and C Teodorov. Sequential and parallel algorithms for the state space exploration. *Cybernetics and Information Technologies*, 16(1) :3–18, (2016).

[8] L Allal, G Belalem, P Dhaussy, and C Teodorov. Using parallel and distributed reachability in model checking. In 2^{nd} *International Conference on Computer, Communication and Computational Sciences (RACCCS-2017)*, volume 628. Aryabhatta College of Engineering & Research Center, Ajmer, India., 2017.

[9] R. Alur, C. Courcoubetis, and D-L. Dill. Model-checking in dense real-time. *Inf. Comput.*, 104(1) :2–34, 1993.

[10] T. Amnell, G. Behrmann, J. Bengtsson, P-R. D'Argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K. Guldstrand Larsen, M-O. Möller, P. Pettersson, C. Weise, and W. Yi. UPPAAL - now, next, and future. In *Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000, Nantes, France, June 19-23, 2000* , pages 99–124, 2000.

[11] M. Armbrust, A. Fox, R. Griffith, A-D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM* , 53(4) :50–58, April 2010.

[12] K. Arnold and J. Gosling. *The Java Programming Language (2Nd Ed.)* . ACM Press/Addison-Wesley Publishing Co., New York, USA, 1998.

[13] C. Baier and J-P. Katoen. *Principles of Model Checking (Representation and Mind Series)* . The MIT Press, 2008.

[14] J. Barnat, L. Brim, and P. Rockai. Divine multi-core - A parallel LTL model-checker. In *Automated Technology for Verification and Analysis, 6th International Symposium, ATVA 2008, Seoul, Korea, October 20-23, 2008. Proceedings* , pages 234–239, 2008.

[15] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification : Model-Checking Techniques and Tools* . Springer Publishing Company, Incorporated, 1st edition, 2010.

[16] R-E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.* , 24(3) :293–318, September 1992.

[17] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.* , 6(4) :632–646, October 1984.

[18] A. Cimatti, E-M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV : A new symbolic model checker. *STTT* , 2(4) :410–425, 2000.

[19] E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science* , pages 353–362, Piscataway, NJ, USA, 1989. IEEE Press.

[20] E-M. Clarke. 25 years of model checking. chapter The Birth of Model Checking, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2008.

[21] E-M. Clarke and E-A Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop* , pages 52–71, London, UK, 1982. Springer-Verlag.

[22] E-M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design* , 10(1) :47–71, 1997.

[23] E M. Clarke, O Grumberg, and D-E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.* , 16(5) :1512–1542, September 1994.

[24] E-M. Clarke, O. Grumberg, M. Minea, and D-A. Peled. State space reduction using partial order techniques. *STTT* , 2(3) :279–287, 1999.

[25] E-M. Clarke, W. Klieber, M. Novácek, and P. Zuliani. Model checking and the state explosion problem. In *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures* , pages 1–30, 2011.

[26] O. Coudert, C. Berthet, and J-C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems* , pages 365–373, London, UK, 1990. Springer-Verlag.

[27] P. Cousot and R. Cousot. A gentle introduction to formal verification of computer systems by abstract interpretation. In Javier Esparza, Bernd Spanfelner, and Orna Grumberg, editors, *Logics and Languages for Reliability and Security* , volume 25 of *NATO Science for Peace and Security Series - D : Information and Communication Security* , pages 1–29. IOS Press, 2010.

[28] J. Daugherty. Java concurrency framework. *CSCI 5448, Spring 2011* , May 2011.

[29] P. Dhaussy, P-Yves. Pillain, S. Creff, A. Raji, Y. Le Traon, and B. Baudry. Evaluating context descriptions and property definition patterns for software formal validation. In *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings* , pages 438–452, 2009.

[30] F. Fages and A. Rizk. On temporal logic constraint solving for analyzing numerical data time series. *Theor. Comput. Sci.* , 408(1) :55–65, November 2008.

[31] C. Flanagan and P. Godefroid. Dynamic partial order reduction for model checking software. *SIGPLAN Not.* , 40(1) :110–121, January 2005.

[32] D. Giannakopoulou. *Model checking for concurrent software architectures* . PhD thesis, Imperial College London, UK, 1999.

[33] E. Goldberg. On bridging simulation and formal verification. In *Proceedings of the 9th International Conference on Verification, Model Checking, and Abstract Interpretation* , VMCAI'08, pages 127–141, Berlin, Heidelberg, 2008. Springer-Verlag.

[34] O. Grumberg, T. Heyman, and A. Schuster. A work-efficient distributed algorithm for reachability analysis. *Form. Methods Syst. Des.* , 29(2) :157–175, September 2006.

[35] M. Gueffaz. *ScaleSem : model checking and semantic web* . Theses, Université de Bourgogne, December 2012.

[36] G. Havas and C. Ramsay. Breadth-first search and the andrews-curtis conjecture. *International Journal of Algebra and Computation* , 13(1), 2003.

[37] G-J. Holzmann. Automated protocol validation in argos : Assertion proving and scatter searching. *IEEE Trans. Softw. Eng.* , 13(6) :683–696, June 1987.

[38] G-J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering* , 23(5) :279–295, May 1997.

[39] G. J. Holzmann. Parallelizing the spin model checker. In *Proceedings of the 19th International Conference on Model Checking Software* , SPIN'12, pages 155–171, Berlin, Heidelberg, 2012. Springer-Verlag.

[40] K. Jeffay. The real-time producer/consumer paradigm : A paradigm for the construction of efficient, predictable real-time systems. In *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing : States of the Art and Practice* , SAC '93, pages 796–804, New York, USA, 1993. ACM.

[41] C. Jegourel. *Rare event simulation for statistical model checking* . PhD thesis, Université de Rennes 1, Novembre 2014.

[42] K. Karimi and H-J. Hamilton. Generation and interpretation of temporal decision rules. *CoRR* , abs/1004.3334, 2010.

[43] V. King, J. Saia, and M. Young. Conflict on a communication channel. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* , PODC '11, pages 277–286, New York, USA, 2011. ACM.

[44] L-M. Kristensen and L. Petrucci. An approach to distributed space exploration for coloured petri nets*. In *Proceedings of the 25th International Conference on Applications and Theory of Petri Nets* , volume 3099 of *ICATPN '04* , pages 474–483, June 2004.

[45] R-P. Kurshan. *Computer-aided Verification of Coordinating Processes : The Automata-theoretic Approach* . Princeton University Press, Princeton, NJ, USA, 1994.

[46] D. Long. *Model checking, abstraction and compositional verification* . PhD thesis, School of Computer Science Carnegie Mellon University, Pittsburgh, PA, march 1993.

[47] K-L. McMillan. *Symbolic Model Checking : An Approach to the State Explosion Problem* . PhD thesis, Pittsburgh, PA, USA, 1992.

[48] K-L. McMillan. *Symbolic Model Checking* . Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[49] A Mukherjee, Z Tari, and P Bertok. Memory efficient state-space analysis in software model-checking. In *Proceedings of the Thirty-Third Australasian Conferenc on Computer Science - Volume 102* , ACSC '10, pages 23–32, Darlinghurst, Australia, 2010.

[50] V. Natarajan and Gerard J. Holzmann. Outline for an operational semantics of promela. In *The Spin Verification System, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, August, 1996* , pages 133–152, 1996.

[51] R. Pelánek. *BEEM : Benchmarks for Explicit Model Checkers* , pages 263–267. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[52] R. Pelánek. Fighting state space explosion : Review and evaluation. In *Formal Methods for Industrial Critical Systems, 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, Revised Selected Papers* , pages 37–52, 2008.

[53] G-L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.* , 12(3) :115–116, 1981.

[54] S. Peyronnet. *Model checking et vérification probabiliste* . PhD thesis, Université Paris 11, Paris, France, 2003.

[55] J-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming* , pages 337–351, London, UK, 1982. Springer-Verlag.

[56] A. Raji. *Integrating Formal Verification Techniques into Software Development Process for Embedded Systems* . Theses, Télécom Bretagne, Université de Bretagne-Sud, March 2012.

[57] R. T. Saad, S. D. Zilio, and B. Berthomieu. A general lock free algorithm for parallel state space construction. In *Proceedings of the 9th International Workshop on Parallel and Distributed Methods in Verification* , PDMC-HIBI '10, pages 8–16, October 2010.

[58] R. T. Saad, S. D. Zilio, and B. Berthomieu. Mixed shared-distributed hash tables approaches for parallel state space construction. In *Proceedings of the 10th International Symposium on Parallel and Distributed Computing* , ISPDC '11, pages 9–16, July 2011.

[59] H. Subramoni, F. Petrini, V. Agarwal, and D. Pasetto. Intra-socket and inter-socket communication in multi-core systems. *Computer Architecture Letters* , 9(1) :13–16, 2010.

[60] C. Teodorov, L. Leroux, Z. Drey, and P. Dhaussy. Past-free[ze] reachability analysis : reaching further with dag-directed exhaustive state-space analysis. *Softw. Test., Verif. Reliab.* , 26(7) :516–542, 2016.

[61] A. Valmari. On-the-fly verification with stubborn sets. In *Proceedings of the 5th International Conference on Computer Aided Verification* , CAV '93, pages 397–408, London, UK, 1993. Springer-Verlag.

[62] W. Visser. Memory efficient state storage in spin. In *Proceedings of the 2nd SPIN Workshop* , pages 21–35, August 1996.

[63] V. Wiels, R. Delmas, D Doose, P.L. Garoche, J. Cazin, and G. Durrieu. Formal Verification of Critical Aerospace Software. *AerospaceLab* , (4) :1–8, May 2012.

[64] P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In *Proceedings of the 4th International Conference on Concurrency Theory* , CONCUR '93, pages 233–246, London, UK, 1993. Springer-Verlag.

[65] J. Yang, A-K. Mok, and F. Wang. Symbolic model checking for event-driven real-time systems. *ACM Trans. Program. Lang. Syst.* , 19(2) :386–412, March 1997.

[66] S. Yovine. KRONOS : A verification tool for real-time systems. *STTT* , 1(1-2) :123–133, 1997.

Résumé

Le Model checking a longtemps été utilisé comme moyen de vérification des spécifications formelles. C'est une technique de vérification qui explore tous les états possibles du système. Cette technique détermine si un système donné satisfait sa spécification. Cette technique souffre du problème d'explosion en temps et en espace mémoire. Dans cette thèse, nous avons proposé d'exploiter les ressources machines afin d'augmenter le gain en temps d'exécution en utilisant des algorithmes parallèles. Nous avons réalisé une comparaison entre deux approches parallèles et nous avons montré que notre approche était plus avantageuse en temps d'exploration et en espace mémoire nécessaire pour le stockage des états. Nous avons par la suite effectué une comparaison entre une exploration parallèle et une exploration distribuée, nous avons conclu par une étude expérimentale, que l'approche parallèle affichait de meilleurs résultats et que le choix des structures de données était très important pour diminuer la complexité spatiale et temporelle d'un algorithme.

Mots clés :

Model checking; Vérification formelle; Exploration; Algorithmes parallèles; Temps d’exécution; Algorithmes distribues; Espace mémoire; Spin; Gain en temps; Structures de données.