Université Joseph Fourier - Grenoble

Thèse

pour obtenir le grade de Docteur
spécialité « Informatique »

par

Meriem ZIDOUNI

Modélisation et analyse des performances de la bibliothèque MPI en tenant compte de l'architecture matérielle

Thèse soutenue le 25 Mai 2010 devant le jury composé de :

M.	Jean-François Méhaut	Président du jury
MM.	Françoise Simonot-Lion	Rapporteur
M.	Jean-Michel Fourneau	Rapporteur
M.	Anne Benoît	Examinateur
M.	Ghassan Chehaibar	Examinateur
M.	Radu Mateescu	Directeur de thèse

À ma mère, À mon père.

Remerciements

Cette thèse constitue une riche expérience qui ne peut s'achever sans remercier les personnes qui m'ont encadré, aidé et soutenu pendant ces trois dernières années. Pourtant il existe une difficulté, celle de n'oublier personne. C'est pourquoi je tiens à remercier par avance ceux dont le nom n'apparaît pas dans cette page. Je tiens à remercier :

Les membres du jury :

Jean-François Méhaut, Françoise Simonot-Lion, Jean-Michel Fourneau et Anne Benoît

Mes encadrants :

Radu Mateescu et Ghassan Chehaibar

L'équipe VASY :

Hubert Garavel, Holger Hermanns, Frédéric Lang, Wendelin Serwe, Gwen Salaün, Helen Pouchot, Etienne Lantreibecq, Nicolas Coste, Damien Thivolle, Yves Guerte, Romain Lacroix, David Champelovier et Olivier Ponsini

Les membres du projet MULTIVAL :

Richard Hersemeule, Yvain Thonnart, François Bertrand et Sahar Foroutan

Les membres de la société BULL-SAS (les clayes-sous-bois) :

Sylvie Lesmanne, Jean Perraudeau, Martine Turon, Philippe Oddo, Stephane Thoison, Jacques Abily, Laurent Delmas, Monika Obrebska, Claude Herve, Guy Rival, Alain Debreil, Patrice Bulot, Roger Stioui et Benoit Coutin

Ma famille :

Ma parents, mon mari, mes frères et sœurs

Titre Modélisation et analyse des performances de la bibliothèque MPI en tenant compte de l'architecture matérielle

Résumé Dans le cadre de son offre de serveurs haut de gamme, la société Bull conçoit des multiprocesseurs à mémoire distribuée partagée avec un protocole de cohérence de cache *CC-DSM* (*Cache-Coherent Distibuted Shared Memory*), et fournit une implémentation de la bibliothèque *MPI* (*Message Passing Interface*) pour la programmation parallèle. L'évaluation des performances de cette implémentation permettra, d'une part, de faire les bons choix d'architecture matérielle et de la couche logicielle au moment de la conception et, d'autre part, fournira des éléments d'analyse nécessaires pour comprendre les mesures faites au moment de la validation de la machine réelle. Nous proposons et mettons en œuvre dans ce travail de thèse une méthodologie permettant d'évaluer les performances des algorithmes de la bibliothèque MPI (*ping-pong* et *barrières*) en tenant compte de l'architecture matérielle. Cette approche est basée sur l'utilisation des méthodes formelles, elle consiste en 3 étapes principales :

1. la modélisation en langage LOTOS des aspects matériels (topologie d'interconnexion et protocole de cohérence de cache) et logiciels (algorithmes MPI) ;
2. la vérification formelle de la correction fonctionnelle du modèle obtenu ;
3. l'évaluation des performances après l'extension du modèle par des informations quantitatives (latences des transferts des données) en utilisant des méthodes numériques et de la simulation.

Mots-clés Bibliothèque MPI · Architectures multiprocesseurs · Méthodes formelles · Model checking · Evaluation des performances · Chaînes de Markov interactives

Title Modeling and performance analysis of the MPI library by taking into account the hardware architecture

Abstract The range of high-end servers designed and manufactured by Bull includes cache-coherent distributed shared memory (*CC-DSM*) multiprocessor systems equipped with an implementation of the Message Passing Interface (*MPI*) library for parallel programming. The evaluation of the performances of this implementation will enable, on the one hand, to make the right choices of the hardware architecture and of the software layer at design time and, will provide, on the other hand, the analysis elements necessary for understanding the experimental measures performed when the real machine is validated. In this thesis, we propose and develop a methodology allowing to evaluate the performances of MPI library primitives (send/receive and barriers) by taking into account the hardware architecture. This approach is based on formal methods and consists of three main phases :

1. modeling in the LOTOS language the hardware aspects (interconnection topology, cache coherency protocol) and the software aspects (MPI primitives and benchmark algorithms) ;
2. formal verification of the functional correctness of the model obtained ;
3. performance evaluation after extending the model with quantitative information (data transfer latencies) by using numerical methods and simulation

Keywords MPI library · Multiprocessor architectures · Formal methods · Model checking · Performance evaluation · Interactive Markov chains

Table des matières

Introduction

Depuis le début de l'informatique, les processeurs et les réseaux de communications n'ont pas cessé de progresser. Cette progression a permis la naissance d'infrastructures matérielles avec des capacités accrues de calcul et de stockage, notamment avec l'apparition des plus récents supercalculateurs [1] destinés au *calcul scientifique haute performance* (HPC : *High Performance Computing*). Les supercalculateurs sont utilisés pour toute tâche nécessitant une très forte puissance de calcul, comme la simulation de phénomènes réels, qui touche à de nombreux domaines scientifiques comme la physique nucléaire, l'astronomie, la biologie, la chimie, etc.

Dans le cadre de son offre de serveurs haut de gamme, la société Bull[2] conçoit des supercalculateurs à mémoire distribuée partagée avec un protocole de cohérence de caches CC-DSM (*Cache-Coherent Distributed Shared Memory*).

Les performances des supercalculateurs ne se résument pas uniquement à l'évolution technologique de leur architectures matérielles, mais concernent également leur couche logicielle qui offre l'interface de programmation parallèle. En effet, la programmation de telles machines à plusieurs degrés de parallélisme implique la résolution de plusieurs problèmes, tels que : la cohérence des données en milieu distribué, l'accès concurrent à une mémoire partagée, l'entrelacement des calculs et des communications qui garantissent la performance du programme parallèle, l'ordonnancement des tâches sur une machine hétérogène, ..., etc. Ces écueils sont d'autant plus cruciaux que les machines parallèles sont avant tous destinées à des applications demandant de gros volumes mémoire et un temps de calcul réduit : la performance des programmes est prioritaire pour un calcul de haute performance.

Le calcul parallèle repose sur l'idée que plusieurs processeurs effectuent de manière coopérative un calcul ou un travail bien précis. Nous pouvons illustrer cette *définition* à l'aide d'un exemple simple, comme le calcul d'un produit de matrice vecteur : $v' = Mv$, avec $M \in \mathbb{R}^{n \times n}$ et $v \in \mathbb{R}^n$. Ce calcul nécessite n^2 opérations lorsqu'il est effectué de façon séquentielle. Nous disposons de n processeurs, c'est à dire autant que la matrice a de lignes. Il est donc logique de demander à chaque processeur de calculer le produit scalaire d'une ligne de matrice par le vecteur v. Pour obtenir v', il suffit alors de rassembler les résultats obtenus par chacun des processeurs. Ainsi si tous les processeurs effectuent les n opérations nécessaires au produit scalaire en même temps, nous aurons obtenu notre résultat en un temps proportionnel à n et à non n^2.

[1] http://www.top500.org
[2] http://www.bull.com

Le calcul parallèle sur les architectures CC-DSM nécessite un mécanisme de communication par *passage de message* entre les différents processeurs. Cela consiste à demander aux processeurs d'émettre ou de recevoir des messages vers ou en provenance des autres processeurs de la machine. La bibliothèque MPI (*Message Passing Interface*) [13] implémente le modèle de programmation parallèle par passage de messages en permettant des communications entre les processeurs moyennant des primitives d'envoi (*send*) et de réception (*receive*) et des synchronisations par des primitives de *barrières*. Revenons à notre exemple précédent, celui du produit matrice vecteur. Nous souhaitons que le résultat soit écrit, à la fin du calcul, sur un seul emplacement, soit la mémoire locale du processeur P_0. À la fin du calcul réalisé par chaque processeur, le résultat à ce moment est éparpillé entre les mémoires associées à chaque processeur. Ils doivent par conséquent communiquer leur résultat en l'envoyant par message au processeur P_0 par la primitive de communication *send*. On peut supposer également que les processeurs ne doivent effectuer aucun autre traitement avant que P_0 ait rassemblé la totalité du vecteur v', dans ce cas, les processeurs doivent se synchroniser à la fin du calcul v' moyennant la primitive de synchronisation de *barrière*.

La communication par passage de message implique la considération d'un autre facteur impactant les performances des applications parallèles : l'implémentation de MPI (algorithmes implémentant les primitives de communication *send* et *receive*, et de synchronisation *barrière*). En effet, la communication par échange de messages s'avère coûteuse en matière de temps d'exécution, par conséquent, il est primordial de procéder à son optimisation.

Notre objectif dans ce travail de thèse est de définir et de mettre en œuvre une méthodologie permettant *la prédiction des performances de la bibliothèque MPI sur une architecture CC-DSM et l'analyse de leur sensibilité par rapport aux aspects logiciels et matériels*. Ceci pour deux raisons cruciales :

1. Faire les bons choix d'architecture matérielle et d'implémentation logicielle au moment de la conception.
2. Comprendre les mesures faites au moment de la validation de la machine réelle.

L'optimisation des algorithmes MPI est un domaine d'études approfondies suivant des directions différentes : optimisation et implémentation sur des architectures génériques comme les clusters [54, 47], ou sur des machines spécifiques [34, 16], ou sur des constructions configurables [80]. Tous ces travaux reposent sur la *mesure* de performances de benchmarks MPI sur des machines réelles *déjà construites*, de telle sorte que l'on peut montrer qu'une telle implémentation est plus efficace qu'une autre. Cependant, quand on veut optimiser une implémentation sur une architecture CC-DSM, nous avons besoin d'analyser l'interaction avec le protocole de cohérence de caches afin de savoir le nombre de *défauts de cache* (*miss*) effectués. Cette information ne peut pas être obtenue avec les mesures, ou bien elle nécessite une instrumentation complexe. Un autre inconvénient

de la mesure est que l'optimisation ne peut être faite pendant la phase de développement de la machine réelle.

Un point clé de notre approche est l'utilisation de méthodes formelles pour la modélisation et l'évaluation des performances. Nous commençons par développer un modèle formel M1 dont nous vérifions sa correction fonctionnelle, puis nous en dérivons un modèle M2 augmenté d'informations quantitatives (latences). Une fois que M1 et M2 sont disponibles, nous nous assurons de leur cohérence en vérifiant que leurs comportements sont équivalents (bisimilaires) du point de vue fonctionnel et ensuite nous effectuons l'analyse markovienne de M2.

Les mécanismes de communication entre les processus MPI utilisent des tampons de messages et éventuellement des verrous pour gérer les exclusions d'accès à ces tampons. Dans une architecture CC-DSM, ces structures de communication résident dans la mémoire distribuée et leurs accès sont régis par le protocole de cohérence de caches. Ceci implique des transferts entre les caches et les mémoires et entre les différents caches, dépendant de l'implantation physique de ces structures et du protocole de cohérence de caches. Les performances de l'implémentation de MPI sur une architecture CC-DSM dépendent de trois paramètres :

1. La succession d'accès définie par l'algorithme des primitives de communication.
2. Le protocole de cohérence de caches, qui régit les changements d'états dans les caches et les transferts des données entre caches et mémoires et entre caches.
3. La topologie de connexion de l'architecture multiprocesseur, qui détermine les latences des transferts.

Ce document est divisé en sept chapitres :

Chapitre 1 présente le contexte de l'étude, à savoir l'environnement matériel des architectures des supercalculateurs en général et celles de Bull en particulier, avec une description détaillée du protocole de cohérence de caches. Dans la présentation de l'environnement logiciel concernant la bibliothèque MPI, nous décrivons les deux benchmarks de MPI auxquels nous nous sommes intéressés pour l'évaluation de leur performances, il s'agit de l'algorithme de communication *ping-pong* et de synchronisation *barrière*.

Chapitre 2 décrit les différents concepts que nous avons utilisés pour la réalisation cette étude : le langage de modélisation *LOTOS*, la théorie IMC pour l'évaluation des performances et les outils de l'environnement CADP que nous avons utilisés.

Chapitre 3, 4, 5 décrivent la mise en œuvre de l'étude : la méthodologie de modélisation, la vérification fonctionnelle des modèles obtenus et finalement leur transformation en chaînes de Markov à temps continu et l'obtention des indices de performances par analyse numérique.

Chapitre 6 traite le problème d'explosion combinatoire de l'espace d'états des modèles et présente les différentes tentatives pour l'éviter tout en préservant la méthode numérique pour l'analyse des performances.

Chapitre 7 Présente l'analyse des performances par simulation dans le but de contourner le problème d'explosion combinatoire de l'espace d'états des modèles.

Finalement, une conclusion générale résume nos contributions et précise des pistes de recherche futures.

Contexte d'étude

1

Il est plus facile de conduire un attelage avec deux bœufs qu'avec mille poulets
– Vieux proverbe de l'Ouest

Sommaire

Ce chapitre est consacré à la présentation du contexte de notre étude, plus précisément les données du problème : l'environnement matériel et l'environnement logiciel.

La première partie de ce chapitre décrit les architectures multiprocesseurs en mettant en évidence les architectures FAME et MESCA de la société Bull. Nous présentons pas la suite le mécanisme assurant la cohérence des données dans ces systèmes tout en expliquant son impact sur la latence d'accès aux données.

Nous présentons dans la seconde partie le principe de communication par passage de message au moyen des primitives du standard MPI. Nous décrivons les deux benchmarks de MPI auxquels nous nous sommes intéressés pour l'évaluation de leur performances, à savoir l'algorithme de communication *ping-pong* et l'algorithme de synchronisation *barrière*.

1.1 Environnement matériel

1.1.1 Architecture des supercalculateurs

Depuis plusieurs années la taxonomie de Flynn [19] est utilisée pour la classification des supercalculateurs, elle est fondée sur le nombre de flux de données (*Single Data stream* ou *Multiple Data stream*) et sur le nombre de flux d'instructions (*Single Instruction stream* ou *Multiple Instruction stream*) présents sur une architecture donnée. De la combinaison de ces possibilités résultent quatre catégories :

- SISD : un seul flot d'instructions, un seul flot de données.
- SIMD : un seul flot d'instructions, plusieurs flots de données. La même instruction est exécutée par plusieurs processeurs en utilisant le même flot de données.
- MISD : plusieurs flots d'instructions, un seul flot de données. Chaque processeurs lit ses propres instructions, mais ils utilisent tous le même flot de données.
- MIMD : plusieurs flots d'instructions, plusieurs flots de données. Chaque processeur lit ses propres instructions et opère sur ses propres données.

Du fait de ces critères trop généraux, la classification de Flynn devient inadéquate pour les supercalculateurs d'aujourd'hui, car ils sont tous considérés comme des MIMD quelles que soient leurs différences. À présent, il est plus courant de subdiviser la classe MIMD en fonction de la structure de la mémoire : centralisée partagée, distribuée partagée et distribuée [92, 76].

Définition 1.1 *Une **mémoire centralisée partagée** désigne un large bloc de mémoire vive partagé entre plusieurs processeurs.*

Les architectures multiprocesseurs à mémoire centralisée partagée sont connues sous deux noms indiquant que tous les processeurs accèdent aux données en mémoire avec la même latence :

Architecture SMP, acronyme de *Symmetric MultiProcessing*.

Architecture UMA, acronyme de *Uniform Memory Access*.

Les architectures SMP représentent une évolution directe des architectures multiprocesseurs. Pour augmenter la puissance de la machine, plusieurs processeurs identiques sont connectés au même bus mémoire comme on peut le voir sur la figure 1.1. Ces processeurs disposent chacun de leur propre mémoire cache et pourront travailler en parallèle pour accélérer l'exécution des processus, ou en exécuter plusieurs de manière concurrente.

La facilité et la portabilité de programmation sur de tels systèmes réduisent considérablement le coût de développement des applications parallèles. En revanche, ils perdent en efficacité lorsque le nombre de processeurs augmente. En effet, le protocole d'accès au bus mémoire interdit le recouvrement des communications. Autrement dit, le bus ne peut être

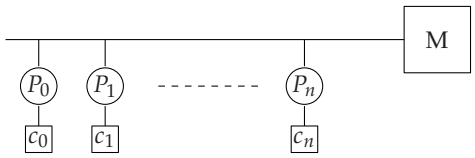


FIG. 1.1 – *Architecture à mémoire centralisée partagée*

utilisé que par un seul processeur à la fois. Ainsi, plus on augmente le nombre de processeurs, plus la contention au niveau du bus est potentiellement importante. En revanche, l'architecture SMP représente la brique de base de la majorité des supercalculateurs.

Définition 1.2 *Une **mémoire distribuée partagée** désigne plusieurs blocs de mémoire vive physiquement distribués mais avec un seul espace d'adressage.*

Généralement un seul espace d'adressage implique des communications implicites entre les processeurs via des lectures et des écritures en mémoire.

Les architectures multiprocesseurs à mémoire distribuée partagée sont référencées sous plusieurs noms mettant en évidence la nature distribuée partagée de la mémoire ou la variation de la latence d'accès aux données en mémoire, ainsi que l'absence ou la présence d'un mécanisme pour assurer la cohérence des caches [76] :

- DSM, acronyme de *Distributed Shared Memory*.
- CC-DSM, acronyme de *Cache Coherent - Distributed Shared Memory*. Elle représente une DSM munie d'un mécanisme assurant la cohérence des caches.
- NUMA, acronyme de *Non Uniform Memory Access*. La latence d'accès à la mémoire varie en fonction de la proximité géographique des processeurs.
- CC-NUMA, acronyme de *Cache Coherent - Non Uniform Memory Access*, est une NUMA avec un munie d'un mécanisme assurant la cohérence des caches.

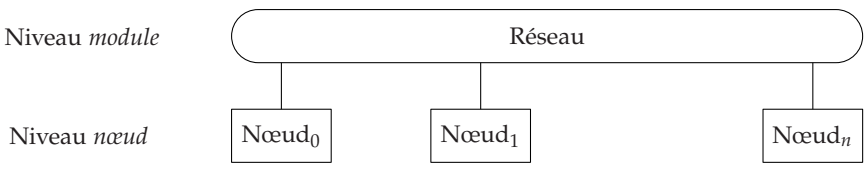


FIG. 1.2 – *Architecture DSM*

Les architectures multiprocesseurs de type DSM possèdent une structure hiérarchique à deux niveaux, comme illustré dans la figure 1.2 :

1. *Nœud* : un multiprocesseur à mémoire centralisée partagée : SMP.
2. *Module* : plusieurs nœuds reliés entre eux via un réseau d'interconnexion.

Définition 1.3 *Une **mémoire distribuée** désigne plusieurs blocs de mémoire vive physiquement distribués dont chaque bloc dispose de son propre espace d'adressage.*

On appelle généralement, les multiprocesseurs à mémoire distribuée, *grappe* ou *cluster*. Un cluster est un ensemble de modules connectés par un réseau, comme illustré dans la figure 1.3.

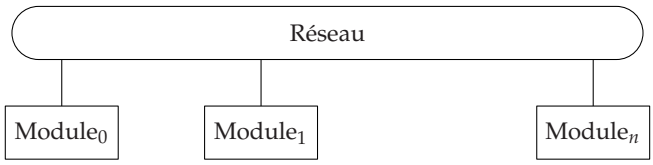
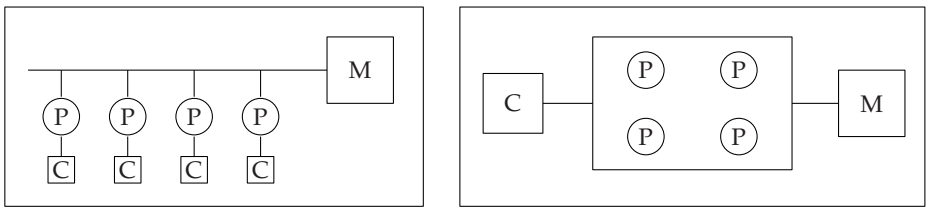


FIG. 1.3 – *Architecture d'un cluster*

La mémoire d'un module est vue comme un bloc de la mémoire distribuée, par conséquent la communication entre les processeurs appartenant à des modules distincts nécessite l'utilisation des modèles de communication par *échange de messages*.

Deux architectures multiprocesseurs de la société Bull ont fait l'objet de ce travail de thèse :

1. FAME (*Flexible Architecture for Multiple Environments*). Les serveurs NovaScale^{TM} bénéficient de l'architecture FAME développée par Bull pour réaliser des systèmes multiprocesseurs à mémoire partagée évolutifs. Le concept de l'architecture FAME repose sur l'interconnexion des nœuds par un switch conçu et développé par Bull. Ce switch assure une vision cohérente d'une mémoire globale formée de la réunion des mémoires associées à chaque nœud.
2. MESCA (*Multiple Environments on Scalable Architecture*) : en cours de développement.



Nœud dans FAME Nœud dans MESCA

FIG. 1.4 – *La structure du nœud dans FAME et MESCA*

La différence de base entre ces deux architectures réside dans la topologie des nœuds et l'attribution des caches aux différents processeurs. Dans le cas de FAME, chaque processeur possède son propre cache. Quant à l'architecture MESCA, un seul cache est attribué à tous les processeurs du même nœud. La figure 1.4 illustre la structure des nœuds dans FAME et MESCA.

Un module est constitué de 4 nœuds connectés par un réseau, comme illustré dans la figure 1.5.

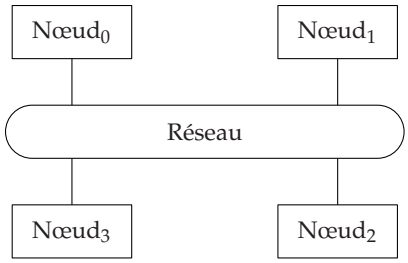


FIG. 1.5 – *La structure du module dans FAME et MESCA*

Un cluster comporte 4 modules connectés par un réseau, comme illustré dans la figure 1.6.

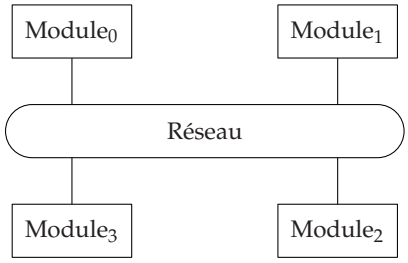


FIG. 1.6 – *Architectures FAME et MESCA*

La topologie d'interconnexion entre les différents composants de l'architecture (processeurs, mémoires, caches) est fixe dans FAME et MESCA. Elle décrit l'emplacement des processeurs dans le système en fonction de l'identificateur du processeur, du nœud et du module.

Comme l'exécution de l'algorithme MPI ne nécessite pas forcément l'utilisation de tous les processeurs du système, nous avons proposé par conséquent une nouvelle définition de la topologie prenant en compte uniquement l'emplacement des processeurs participant à l'exécution de l'algorithme MPI dans le système. Ainsi, pour un certain nombre de processeurs, on peut définir plusieurs topologies.

Le tableau 1.1 décrit les 3 principales topologies que nous avons utilisées dans notre étude :

1. $topology_0$: tous les processeurs se trouvent dans le même nœud.
2. $topology_1$: la plupart des processeurs se trouvent dans des nœuds différents du même module.
3. $topology_2$: la plupart des processeurs se trouvent dans des modules différents.

Chaque module est défini par le quadruple (n_0, n_1, n_2, n_3), avec n_i représente le nombre de processeurs dans le nœud i, où $i \in \{0, 1, 2, 3\}$.

Topologie	Module	Nombre processeurs				
		2	3	4	5	6
$topology_0$	$module_0$	(2,0,0,0)	(3,0,0,0)	(4,0,0,0)	(5,0,0,0)	(6,0,0,0)
	$module_1$	(0,0,0,0)	(0,0,0,0)	(0,0,0,0)	(0,0,0,0)	(0,0,0,0)
	$module_2$	(0,0,0,0)	(0,0,0,0)	(0,0,0,0)	(0,0,0,0)	(0,0,0,0)
	$module_3$	(0,0,0,0)	(0,0,0,0)	(0,0,0,0)	(0,0,0,0)	(0,0,0,0)
$topology_1$	$module_0$	(1,1,0,0)	(1,1,1,0)	(1,1,1,1)	(2,1,1,1)	(2,2,1,1)
	$module_1$	(0,0,0,0)	(0,0,0,0)	(0,0,0,0)	(0,0,0,0)	(0,0,0,0)
	$module_2$	(0,0,0,0)	(0,0,0,0)	(0,0,0,0)	(0,0,0,0)	(0,0,0,0)
	$module_3$	(0,0,0,0)	(0,0,0,0)	(0,0,0,0)	(0,0,0,0)	(0,0,0,0)
$topology_2$	$module_0$	(1,0,0,0)	(1,0,0,0)	(1,0,0,0)	(1,1,0,0)	(1,1,0,0)
	$module_1$	(1,0,0,0)	(1,0,0,0)	(1,0,0,0)	(1,0,0,0)	(1,1,0,0)
	$module_2$	(0,0,0,0)	(1,0,0,0)	(1,0,0,0)	(1,0,0,0)	(1,0,0,0)
	$module_3$	(0,0,0,0)	(0,0,0,0)	(1,0,0,0)	(1,0,0,0)	(1,0,0,0)

TAB. 1.1 – *Les topologies en fonction du nombre de processeurs*

1.1.2 Notion de mémoire cache et cohérence des caches

La conception de la mémoire a suscité une attention considérable des concepteurs des ordinateurs à cause de son importance dans les calculs effectués par les processeurs. En effet, l'exécution d'une simple instruction comporte un ou plusieurs accès à la mémoire.

Avec l'explosion de la taille de la mémoire et le dépassement des vitesses d'accès mémoire par les vitesses des processeurs, la mémoire est devenue rapidement un goulot d'étranglement (*bottleneck*). La latence d'accès à la mémoire dépend fortement de sa taille : plus la taille est grande, plus l'accès est lent. Ceci a impliqué la nécessité d'une structure hiérarchique pour minimiser le temps d'accès effectif et améliorer ainsi les performances. Ces différents niveaux hiérarchiques sont représentés sur la figure 1.7 pour le cas le plus général. Le niveau supérieur (éloigné du processeur) correspond à des mémoires de grande capacité avec un temps d'accès important contrairement à celles du niveau inférieur. En effet, les mémoires sont d'autant plus petites et rapides qu'elles sont proches du processeur.

FIG. 1.7 – *Classification hiérarchique des mémoires*

La hiérarchie de la mémoire se base sur le principe de la localité [88]. L'accès fréquent à des instructions et à des données placées dans des mémoires du niveau supérieur de la hiérarchie implique une pénalité dans le

temps d'accès qu'on évite en plaçant ces instructions et ces données dans les mémoires du niveau inférieur (les caches) qui sont plus rapides.

Les mémoires *caches* (dites aussi *antémémoires*) sont basées sur une idée proche de la notion de mémoire virtuelle : quelques sections actives de données sont stockées en les dupliquant dans un module plus rapide [30, 77]. La différence essentielle se situe au niveau de l'implémentation de ces mécanismes, sur un cache, la gestion des données est faite par le *matériel*. En effet le mot *cache* est souvent utilisé dans un sens plus large : dispositif matériel ou logiciel stockant dans une zone d'accès rapide des données qui se trouvent en grande quantité dans une zone beaucoup plus vaste mais d'accès plus lent.

Physiquement, le cache est organisé en *lignes* de données, représentant des unités de transfert entre la mémoire et le cache. Quand une requête de lecture est envoyé vers la mémoire principale, le ligne est transférée vers le cache, tout en assurant la correspondance entre la ligne de la mémoire principale et du cache.

La requête sur une donnée est d'abord adressée au cache. On dit qu'il y a un *succès de cache* (*cache hit*) si le cache peut satisfaire la requête, sinon, il s'agit d'un *défaut de cache* (*cache miss*) qui désigne un accès à une donnée ne se trouvant pas dans le cache, ou plus généralement, n'étant pas valide en cache.

Remarque 1.1 *Afin de faciliter la modélisation de la mémoire et les caches de notre système, on considère qu'une* ligne *mémoire ou cache comporte une seule donnée.*

Une architecture multiprocesseur est une architecture *multi-caches* du fait que chaque processeur possède son propre cache, par conséquent, une donnée peut se trouver dans plusieurs caches simultanément. En vue d'assurer la cohérence entre les données présentes dans les différents caches, les accès sont orchestrés par un *protocole de cohérence de caches*.

Suivant l'évolution des architectures multiprocesseurs, de nombreux mécanismes de cohérence de caches ont vu le jour, regroupant un ensemble d'avantages et d'inconvénients d'implémentation et d'efficacité. Nous citons dans ce qui suit deux protocoles de cohérence de caches, le premier s'appliquant sur des architectures à mémoire partagée et le second sur des architectures à mémoire distribuée partagée.

Cohérence par espionnage de bus (*snooping coherence*) est un mécanisme basé sur l'utilisation de la connexion physique préexistante reliant les processeurs et la mémoire [75, 52, 28]. Les processeurs surveillent toutes les requêtes émises sur le bus. Quand une demande d'écriture est diffusée sur le bus, tous les caches qui détiennent une copie valide, l'invalident.

Bien que ce soit un mécanisme relativement simple à mettre en œuvre, le partage de bus peut créer un goulot d'étranglement et altérer les performances. Plusieurs méthodes ont été proposées pour réduire la charge

sur un seul bus [29, 1, 93], elles consistent en général à ajouter des bus supplémentaires pour augmenter la bande passante entre les processeurs et la mémoire partagée, mais le problème de la charge sur les bus se pose toujours quand le nombre de processeurs devient important.

Par conséquent, l'efficacité du mécanisme de cohérence par espionnage de bus se limite à des architectures de faible capacité en nombre de processeurs.

Cohérence par répertoire (*directory based*) est l'un des premiers mécanismes de cohérence de caches [7], il a été même proposé avant celui de l'espionnage de bus. Il consiste en l'utilisation d'une structure supplémentaire décrite sous forme d'un répertoire centralisé qui maintient l'état des données dans les caches de tous les processeurs du système. Avant l'accès à une donnée pour une écriture, le processeur envoie sa demande au répertoire. Ce dernier invalide par la suite tous les caches qui détiennent une copie valide de la donnée en question, garantissant ainsi un accès exclusif au processeur demandeur.

Cependant, l'utilisation d'un seul répertoire centralisé limitera le nombre de processeurs. Afin de pallier à ce problème d'extensibilité, les auteurs de [37] proposent une solution basée sur la répartition du répertoire sur les toutes les parties de la mémoire, c'est à dire associer à chaque fragment de mémoire un répertoire, comme décrit dans la figure 1.8.

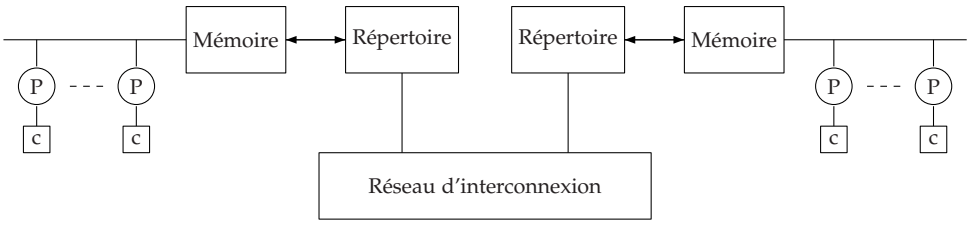


FIG. 1.8 – *Cohérence de caches par répertoire distribué*

Dans les architectures de Bull, la cohérence des caches est maintenue par la combinaison des deux mécanismes de cohérence ci-dessus :

- la cohérence par espionnage de bus pour les processeurs du même nœud.
- la cohérence par répertoire distribué pour les processeurs situés dans des nœuds différents.

Les droits d'accès aux données sont décrits sous forme *d'état*. De façon générale, un état d'une donnée indique si elle est valide ou non dans le cache.

On définit le protocole de cohérence de caches comme étant un ensemble de règles indiquant les changements des états de caches et le type de transfert suite à une demande d'accès en lecture ou en écriture.

Le protocole de cohérence de caches implémenté dans le système que nous étudions est basé sur le protocole MESI classique [75], dans lequel une ligne de cache se trouve dans l'un des quatre états suivants :

M (*Modified*) : la ligne est présente et valide dans le cache, qui en est le seul détenteur, mais potentiellement différente de celle en mémoire.

E (*Exclusive*) : la ligne est présente et valide dans le cache, qui en est le seul détenteur. La ligne a la même valeur qu'en mémoire.

S (*Shared*) : la ligne est présente et valide dans le cache. Elle est potentiellement présente et valide dans d'autres caches également, et sa valeur est partout la même qu'en mémoire.

I (*Invalid*) : la ligne est invalide (ou elle n'est pas présente dans le cache), ainsi toute référence émise par le processeur provoque une demande de transfert depuis la mémoire ou un autre cache.

On désigne par le *type de transfert* l'endroit d'où la donnée est chargée pour satisfaire la demande d'accès. On distingue 3 sortes de transfert :

- *Internal* : la donnée est chargée depuis le cache du demandeur. Il s'agit d'un accès direct au cache sans transaction sur le bus.
- *Memory* : la donnée est chargée depuis la mémoire.
- *Cache* : la donnée est chargée depuis le cache où elle se trouve valide.

Les règles de changement d'état des caches ainsi que les types de transfert appropriés suite à une lecture (resp. écriture) sont décrites dans le tableau 1.2 (resp. le tableau 1.3), où C_i représente l'état de cache du processeur demandeur P_i et $C_{j\neq i}$ est l'état de cache d'un processeur $P_{j\neq i}$ dans le système, qui en fonction de son état peut satisfaire ou non la demande de P_i.

Règle	État courant		État suivant		Type de transfert
	C_i	$C_{j\neq i}$	C_i	$C_{j\neq i}$	
RL_1	I	I	E	I	*Memory*
RL_2	I	S	S	S	*Memory*
RL_3	I	E	S	S	*Memory*
RL_4	I	M	E	I	*Cache*(C_j)
RL_5	I	M	S	S	*Cache*(C_j)
RL_6	S/ E / M	*	S / E /M	*	*Internal*

TAB. 1.2 – *Protocole de cohérence de caches pour accès en lecture*

Règle	État courant		État suivant		Type de transfert
	C_i	$C_{j\neq i}$	C_i	$C_{j\neq i}$	
RS_1	I / S	I	E	I	*Memory*
RS_2	I / S	S / E	S	I	*Memory*
RS_3	I	M	S	I	*Cache(C_j)*
RS_4	M	*	M	I	*Internal*

TAB. 1.3 – *Protocole de cohérence de caches pour accès en écriture*

Quand un processeur fait une demande d'écriture d'une donnée qui se trouve en état E dans son cache, il change son état en M silencieusement sans transaction sur le bus, et par conséquent sans invalidation de la donnée dans les autres caches. Afin de s'assurer que les autres processeurs vont accéder par la suite à la bonne valeur :

- en lecture : ça correspond à la règle RL_3 du tableau 1.2. La donnée est chargée depuis la mémoire avec la *consultation* du cache C_j afin de vérifier que P_j n'a pas changé son état silencieusement.
- en écriture : ça correspond à la règle RS_2 du tableau 1.3. A la différence pour un accès en lecture qui ne nécessite qu'une consultation de cache, en écriture le cache C_j est *consulté* et *invalidé*.

Afin de faciliter la référence aux accès mémoire nécessitant la consultation ou l'invalidation de cache, nous définissons un quatrième type de transfert *MemoryInv* signifiant que la donnée est chargée depuis la mémoire avec la consultation du cache où elle se trouve en état E s'il s'agit d'un accès en lecture et l'invalidation des caches où elle se trouve en état E ou S pour un accès en écriture.

Définition 1.4 *Nous définissons deux protocoles de cohérence de caches en fonction des règles de changement d'états utilisées :*

1. *Protocole de cohérence de caches A, avec*

$$A = \{RL_1, RL_2, RL_3, RL_4, RL_6\} \cup \{RS_1, RS_2, RS_3, RS_4\}$$

2. *Protocole de cohérence de caches B, avec*

$$B = \{RL_1, RL_2, RL_3, RL_5, RL_6\} \cup \{RS_1, RS_2, RS_3, RS_4\}$$

En fonction de l'ordre d'accès aux variables partagées défini dans l'algorithme MPI, ses performances peuvent varier d'un protocole de cohérence de caches à un autre. Le protocole A est plus performant que le protocole B dans le cas ou les variables partagées sont souvent écrites après avoir été lues, c'est à dire, dans le cas ou l'algorithme comporte des séquences d'accès à la même variable partagée, de la forme

$$store(P_j);\ load(P_i);\ store(P_i)$$

En revanche l'utilisation du protocole B est plus avantageuse que celle de A dans le cas ou les variables partagées sont rarement écrites mais souvent lues, c'est à dire l'algorithme comporte des séquences d'accès à la même variable partagée, de la forme

$$store(P_j);\ load(P_i);\ load(P_j)$$

où $load(P_i)$ (resp. $store(P_i)$) représente un accès en lecture (resp. en écriture) réalisé par le processeur P_i.

Pour plus de clarté, nous proposons l'exemple suivant 1.1.

Exemple 1.1 *Soient Seq_0 et Seq_1 deux séquences d'accès réalisées par P_0 et P_1 sur une variable partagée v :*

$$Seq_0 = store(P_1);\ load(P_0);\ store(P_0)$$
$$Seq_1 = store(P_1);\ load(P_0);\ load(P_1)$$

Le tableau 1.4 (resp. 1.5) décrit l'état de cache après chaque accès de la séquence Seq_0 (resp. Seq_1) ainsi que le nombre de miss dans le cas des deux protocoles de caches A et B.

Instant	Protocole	P_0			P_1		
		Opération	Cache	Miss	Opération	Cache	Miss
t_0	A	–	I	0	Store	M	1
	B	–	I	0	Store	M	1
t_1	A	Load	E	1	–	I	0
	B	Load	S	1	–	S	0
t_2	A	Store	M	0	–	I	0
	B	Store	M	1	–	I	0

TAB. 1.4 – *Nombre de miss réalisé dans la séquence Seq_0 pour le protocole A et B*

Le nombre total de miss de la séquence Seq_0 dans le cas du protocole A est inférieur à celui réalisé dans le protocole B (2 vrs 3), ainsi l'exécution de la séquence Seq_0 dans une architecture utilisant le protocole A est plus avantageuse.

Suite à l'accès en lecture réalisé par P_0 à l'instant t_1 , la variable est en état E dans le cache de P_0 ce qui implique un transfert interne pour l'accès en écriture réalisé à l'instant t_2 selon la règle RL_4 dans le cas du protocole A, en guise d'un transfert mémoire dans le cas du protocole B suivant la règle RL_5.

L'exécution de la séquence Seq_1 dans un architecture utilisant le protocole B est plus avantageuse que celle utilisant le protocole A en raison du nombre de miss engendré, qui est égal à 2 dans le cas de B et à 3 dans le cas du protocole A.

Dans le cas du protocole A, l'accès en lecture par P_0 à l'instant t_1 mène la variable à l'état E dans son cache et à l'état I dans le cache de P_1 selon la règle RL_4, impliquant par conséquent un transfert depuis la mémoire pour l'accès en écriture de P_1 à l'instant t_2, alors que dans le protocole B, P_1 réalise un transfert interne à l'instant t_2, car P_1 a mené la variable à l'état S dans tous les caches.

Instant	Protocole	P_0			P_1		
		Opération	Cache	Miss	Opération	Cache	Miss
t_0	*A*	–	*I*	0	Store	*M*	1
	B	–	*I*	0	Store	*M*	1
t_1	*A*	Load	*E*	1	–	*I*	0
	B	Load	*S*	1	–	*S*	0
t_2	*A*	–	*S*	0	Load	*S*	1
	B	–	*S*	0	Load	*S*	0

TAB. 1.5 – *Nombre de miss réalisé dans la séquence Seq_1 pour le protocole A et B*

1.1.3 Les latences d'accès

Le temps consacré à la communication entre processeurs ou entre les processeurs et mémoires est l'un des paramètres fondamentaux de l'efficacité des algorithmes parallèles, ce temps représente une fraction non négligeable du temps nécessaire à la résolution du problème, et l'algorithme peut souffrir d'une perte d'efficacité plus au moins liée à la communication.

Définition 1.5 *La latence d'accès représente la durée de temps nécessaire pour satisfaire une demande d'accès. Il s'agit du temps qui s'écoule entre l'envoi de la demande d'accès et la réception effective de la donnée demandée.*

La latence d'accès n'est connue qu'au moment de l'accès et il est impossible de la prédire avant. Ceci s'explique par le fait que l'emplacement d'une donnée valide est déterminé - suivant le principe de la cohérence des caches - par le dernier accès à cette donnée. Et comme la plupart des algorithmes consistent en des exécutions parallèles entre processus et des accès aux données partagées, la latence d'un accès ne peut être évaluée statiquement puisqu'elle dépend de l'entrelacement des accès qui l'auront précédée.

La latence d'accès dépend de deux paramètres :

1. Le *type de transfert*, qui est donné par le protocole de cohérence de caches suivant l'état de la donnée dans les caches.
2. Le *niveau de transfert*, représentant la distance entre la source et la destination de la donnée demandée.

La structure hiérarchique des architectures FAME et MESCA nous a permis de définir 4 niveaux de transfert en fonction de la distance entre la source et la destination de la donnée :

1. *Internal* : la source de la donnée se trouve dans le cache du demandeur.
2. *Intra_Node* : la source et la destination se trouvent dans le même nœud.

3. *Inter_Node* : la source et la destination se trouvent dans des nœuds différents du même module.
4. *Inter_Module* : la source et la destination se trouvent dans des modules différents.

Le tableau 1.6 ci-dessous résume les différentes latences d'accès possibles dans la cas de FAME et MESCA. Les valeurs numériques des latences étant des données confidentielles de la société Bull, nous utilisons des annotations symboliques pour les référencer.

Type de transfert	Niveau de transfert			
	Internal	*Intra_Node*	*Inter_Node*	*Inter_Module*
Internal	C_INT	–	–	–
Memory	–	M_FSB_1	M_SP	M_XSP
Cache	–	C_FSB	C_SP	C_XSP

Tab. 1.6 – *Latences des différents accès dans les architectures FAME et MESCA*

A la différence des latences d'accès des différents types de transferts du tableau 1.6, qui dépendent uniquement de la distance entre la source et la destination de la donnée, la latence d'un transfert *MemoryInv* dépend aussi de la distance entre la source de la donnée et le cache de consultation et/ou d'invalidation, comme illustré dans le tableau 1.7.

		Niveau de consultation et/ou invalidation		
		Intra_Node	*Inter_Node*	*Inter_Module*
Niveau de transfert	*Intra_Node*	M_FSB_1	M_FSB_2	M_FSB_3
	Inter_Node	M_SP	M_SP	M_SP
	Inter_Module	M_XSP	M_XSP	M_XSP

Tab. 1.7 – *Latences d'un transfert* MemoryInv *dans les architectures FAME et MESCA*

1.2 Environnement logiciel

1.2.1 Communication par échange de messages

Dans une architecture à mémoire distribuée, la machine parallèle est vue comme un ensemble de modules équivalents, chaque module étant constitué de plusieurs nœuds et de sa mémoire associée. Les mémoires des différents modules étant physiquement distribuées, pour accéder aux données d'un autre module, il est nécessaire de réaliser une communication. L'*échange de messages* (*message passing*) représente l'un des principaux paradigmes de communication : les processus ne partagent pas des données, la coopération ne se fait que par émission et réception de messages.

On reconnaît deux classes d'opérateurs de communication selon les interlocuteurs impliqués. On parle de communication *point à point* (*bipoint*) s'il y a exactement un émetteur et un récepteur, sinon on parle de communication *collective* (*multipoint*).

Remarque 1.2 *Les besoins des applications les plus exigeantes en terme de puissance de calcul dépassent les capacités des plus grands SMP. Ceci à mené l'industrie à combiner le modèle de programmation multiprocesseur à mémoire partagée à une programmation par* passage de message.

• ***Communication point à point***

Dans une communication par passage de messages en mode point à point, un émetteur envoie les données et le récepteur les reçoit explicitement. Cette communication est assurée par les primitives *send* et *receive*. Le nombre et la complexité des paramètres de ces primitives varient selon la bibliothèque d'échange de messages utilisée. Cependant ils doivent permettre de résoudre les problèmes suivants :

- L'identification des interlocuteurs.
- La description des zones mémoire dédiées à l'émission et celles dédiées à la réception.
- La garantie de l'ordre de délivrance des messages.

Les communications point à point peuvent être *bloquantes* ou *non-bloquantes* :

- *La communication bloquante* empêche la poursuite de l'exécution d'un programme tant que la communication n'est pas entièrement réalisée.
- *La communication non-bloquante* laisse le programme poursuivre son exécution, que la communication soit réellement faite ou non. Elle présente un avantage incontestable en terme de temps d'exécution puisqu'un processus peut travailler tout en envoyant des données.

Au-delà du caractère bloquant ou non-bloquant des primitives, il existe quatre modes de communication :

1. *Standard* : le mode asynchrone de communication, la terminaison d'une requête d'émission ne dépend pas nécessairement de la présence d'une requête de réception sur le récepteur.
2. *Synchrone* : l'envoi ne se termine que lorsque la requête de réception correspondante a été postée (il y a une synchronisation entre l'émetteur et le récepteur).
3. *Ready* : une émission ne doit être faite que si l'application est certaine que la réception correspondante a été postée.
4. *Bufferisé* : l'émetteur demande explicitement une sauvegarde intermédiaire des données dans un tampon réservé aux émissions bufferisées, l'envoi se termine dès que les données ont été recopiées dans ce tampon.

• ***Communication collective***

La communication collective peut être divisées en trois catégories : *synchronisation*, *diffusion* et *réduction*.

1. *Synchronisation*. Il s'agit essentiellement d'une *barrière* où tous les processus attendent tous les autres processus. En effet, dans ce mode de communication il n'y a pas d'échange de messages mais tous les processus sont assurés que tous ont rallié le point de synchronisation. Ces synchronisations sont assurées par des primitives dites primitives de *barrière*.
2. *Diffusion*. On distingue deux variantes : la *diffusion totale* (*broadcast*). Ici un processus, dit racine (*root*), appartenant à un groupe, envoie un message à tous les autres processus participants au groupe. Le processus racine peut être choisi parmi ceux qui composent le groupe. La deuxième version est la distribution (*scatter*) qui consiste à répartir les données d'un tableau présent sur un processus (la racine) sur les autres processus du groupe. Le contenu de la n^{me} position du tableau est envoyé au n^{me} processus du groupe. L'opération de *regroupement* (*gather*) est l'inverse de la distribution et permet à un processus de récupérer un ensemble de données à partir d'autres processus. Essentiellement, une donnée du n^{me} processus est reçu par le processus racine et mise sur la n^{me} position d'un tableau destiné à la réception de données. Finalement, l'*échange total* (*scatter/gather*) où les processus d'un groupe envoient et reçoivent les données de tous les autres processus participants du groupe.
3. *Réduction*. Elle représente une variation de l'opération de regroupement où les valeurs reçues par le processus racine sont combinées en une seule valeur par le biais d'une opération associative et commutative. Par exemple, la recherche du minimum d'un tableau distribué sur tous les processus. Le résultat du calcul est ensuite communiqué à un ou plusieurs processus.

Dans notre étude, nous nous intéressons particulièrement à deux types de communication : la communication *point à point asynchrone* et la communication collective de catégorie *synchronisation* (les barrières).

1.2.2 Le standard MPI

La programmation par échange de messages est largement utilisée sur des machines à mémoire distribuée. Plusieurs bibliothèques ont été mises à disposition soit par les constructeurs de machines soit par des laboratoires de recherche, citons par exemple, MPL [83], Express [55], Parmacs [5], Intel NX [78] et P4 [62].

Initialement la plus répandue était la PVM [27], acronyme pour *Parallel Virtual Machine*, développée par le *Oak Ridge National Laboratories* (USA). Le succès de PVM était dû, d'une part, au fait qu'il proposait un environnement de programmation par échange de messages pour des milieux hétérogènes et homogènes ainsi qu'une collection de primitives pour des applications écrites en langage C et en Fortran. D'autre part, PVM est

accessible gratuitement à partir du web. PVM était disponible pour une grande gamme de machines et de systèmes d'exploitation.

Pour éviter la prolifération de différentes versions de bibliothèques de communication et corriger des insuffisances de l'interface de PVM, un groupe d'universités et de constructeurs informatiques ont défini un *standard* de communication par messages : MPI (*Message Passing Interface*) [13].

MPI est un ensemble de spécifications et de fonctionnalités qui ne fait aucune supposition sur le matériel sous-jacent. Cette indépendance lui a assuré son succès : de multiples implémentations sont disponibles, aussi bien libres que commerciales. Les premières visent la portabilité et le support des configurations hétérogènes tandis que les secondes ont pour objectif une exploitation optimale du matériel. MPI est une réussite car des programmes peuvent effectivement fonctionner sur des architectures très différentes tout en ayant de bonnes performances.

Parmi les implémentations les plus populaires de MPI, nous citons MPICH [33]. C'est une implémentation libre MPI qui dispose de nombreuses versions suivant le type de matériel réseau utilisé mais aussi d'un module mémoire partagée qui lui permet d'être utilisée pour des communications entre les nœuds. C'est donc une implémentation polyvalente adaptée aux architectures mémoire distribuée partagée et mémoire distribuée. La communication point à point que nous avons étudiée repose sur une implémentation MPICH.

Dans l'approche adoptée pour la programmation parallèle par la bibliothèque MPI, un ensemble de processus exécutent un programme écrit dans un langage séquentiel dans lequel ont été rajoutés des appels aux différentes primitives contenues dans la bibliothèque permettant l'envoi et la réception de messages.

Au sein d'un calcul décrit avec MPI, plusieurs processus communiquent ensemble via ces appels aux primitives de la bibliothèque. Dans la plupart des implantations de MPI, un nombre fixé de processus est créé à l'initialisation, et, généralement, un seul processus est créé par processeur. Le modèle de programmation parallèle MPI est souvent référencé en tant que SPMD (*Single Process, Multiple Data*) dans lequel tous les processus exécutent le même programme, à distinguer des modèles MPMD (*Multiple Process, Multiple Data*) dans lesquels les processus peuvent exécuter des programmes distincts.

Les processus peuvent utiliser des opérations de communication point à point pour envoyer un message depuis un processus nommé vers un autre. Un groupe de processus peut également employer des opérations de communications collectives bien utiles pour les opérations de diffusion ou de réduction ou de distribution ou redistribution de données. La bibliothèque MPI permet également l'utilisation de communications asynchrones.

1.2.3 Les benchmark MPI étudiés

Nous nous intéressons dans ce travail de thèse à l'évaluation de la durée de l'exécution des primitives MPI, en particulier les primitives *send* et *re-*

ceive pour un benchmark de communication point à point, et la primitive *barrière* pour un benchmark de synchronisation par barrière.

Le benchmark de communication point à point

Le benchmark de communication point à point, dit *ping-pong*, consiste à des échanges de messages entre deux processus en utilisant la primitive *send* pour l'envoi d'*un seul* message et la primitive *receive* pour la réception d'*un seul* message également.

La figure 1.9 illustre le principe de fonctionnement de l'algorithme de *ping-pong* entre les processus P_0 et P_1. Pour chacun des processus, l'exécution de la primitive *send* est toujours suivie d'une exécution de la primitive *receive* et inversement, c'est à dire qu'un envoi est toujours suivi d'une réception et une réception est toujours suivie d'un envoi.

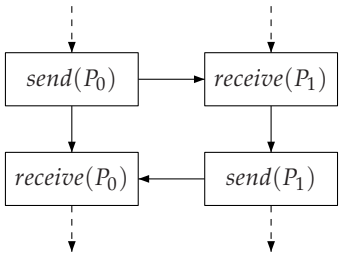


FIG. 1.9 – *Communication point à point*

Objectif *Dans un mode de communication point à point, notre but est l'évaluation de la latence d'un échange de message qui correspond à la somme des durées de l'exécution des primitives* send *et* receive.

Le benchmark de synchronisation par barrière

Le benchmark de synchronisation par barrière, qu'on appelle algorithme de *barrière*, consiste à réaliser une infinité de synchronisations entre plusieurs processus en utilisant une primitive de barrière.

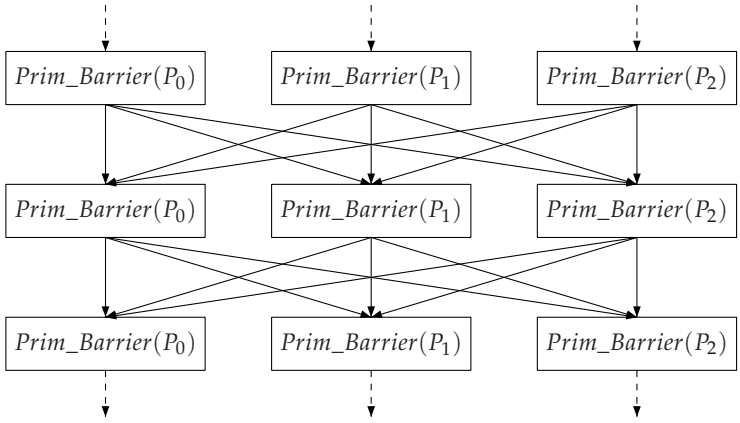


FIG. 1.10 – *Le fonctionnement du benchmark de barrière entre 3 processus*

La figure 1.10 illustre le principe de fonctionnement de l'algorithme de barrière entre 3 processus, P_0, P_1 et P_2. Un processus ne peut entamer la

$(n+1)^{eme}$ exécution de la primitive *Prim_Barrier*() que si tous les autres processus ont bien fini leur n^{eme} exécution de la primitive *Prim_Barrier*().

Objectif *Dans un mode de communication collective de synchronisation par barrière, notre but est l'évaluation de la latence d'une synchronisation qui correspond à la durée de l'exécution de la primitive Prim_Barrier*().

Nous avons étudié l'algorithme de barrière pour 3 primitives différentes : *centralized*, *tournament* et *combining tree*.

La primitive *centralized* adopte une définition très simpliste du point de synchronisation auquel tous les processus participant à la barrière vont être ralliés. En effet, l'état de synchronisation dépend du contenu d'une seule variable partagée, si son contenu est égal à 0, cela signifie l'arrivée de tous les processus au point de synchronisation.

Algorithme 1 : La barrière *centralized*

```
Data :
  shared count : integer := P
  shared sense : boolean := true
  processor private local_sense : boolean := true
begin
    procedure central_barrier
      local_sense := not local_sense
      if fetch_and_decrement (&count) = 0
        count := P
        sense := local_sense
      else repeat until sense = local_sense
end
```

La variable partagée *count* est initialisée au nombre de processus, par la suite à chaque fois qu'un processus arrive à la barrière il décrémente son contenu de 1 pour marquer son arrivée et, si la nouvelle valeur de *count* est égale à 0, il informe les autres processus en attente active de la fin de la barrière, sinon il attend qu'il soit informé de la fin de la barrière.

Conclusion du chapitre

Dans ce chapitre, nous avons présenté le contexte de la thèse. Le mécanisme de communication entre les processus MPI utilisent des tampons de messages et éventuellement des verrous pour gérer les exclusions d'accès à ces tampons. Dans une architecture CC-DSM, ces structures de communication résident dans la mémoire distribuée et leurs accès sont régis par le protocole de cohérence de caches. Ceci implique des transferts entre les caches et les mémoires, et entre les différents caches qui dépendent de l'implantation physique de ces structures et du protocole. La latence de ces transferts détermine la performance des communications et des synchronisations entre les processus. Ainsi les performances de l'implémentation de MPI sur une architecture CC-DSM dépendent de trois paramètres : 1) la succession d'accès définie par l'algorithme des primitives de communication ; 2) le protocole de cohérence de caches, qui régit les changements d'états dans les caches et les transferts entre caches et mémoires, et entre

caches ; 3) la topologie de connexion de l'architecture multiprocesseur, qui détermine les latences des transferts.

Langage, théorie et outils

2

Sommaire

Nous présentons dans ce chapitre les concepts que nous avons utilisés pour l'évaluation des performances d'algorithmes MPI sur une architecture multiprocesseurs.

Nous entamons ce chapitre par la présentation du langage de modélisation LOTOS avec lequel nous avons spécifié le comportement fonctionnel et stochastique de notre système.

Nous consacrons la seconde section à la description de la théorie sur laquelle se base la phase d'évaluation des performances : les chaînes de Markov interactives (IMC). Nous abordons cette section par un rappel détaillé sur les processus stochastiques et les chaînes de Markov à temps continu. Ce rappel est indispensable pour une présentation claire des IMC ainsi qu'une bonne interprétation de la procédure d'évaluation des performances présentée dans le chapitre 5. Nous achevons cette section par la description du principe de la modélisation des IMC en langage LOTOS.

Nous clôturons ce chapitre par la description des principaux outils d'environnement CADP que nous avons utilisés pour mettre on œuvre notre étude : de la spécification formelle, en passant par la vérification fonctionnelle à l'évaluation des performances.

2.1 Langage LOTOS

LOTOS [48] (*Language Of Temporal Ordering Specification*) est une technique de description formelle normalisée par ISO. Il a été défini à l'origine pour la description des protocoles de télécommunication, mais il est aussi bien adapté pour la spécification formelle des architectures matérielles et du fonctionnement des systèmes distribués. L'idée de départ du langage LOTOS est qu'un système peut être spécifié en définissant les relations temporelles entre les interactions qui constituent son comportement observable [3].

La première partie de LOTOS traite les données internes aux processus et les traitements qui s'y attachent en s'appuyant sur le langage ActOne [14]. La seconde partie traite les comportements parallèles et leurs interactions, elle est basée sur les algèbres de processus et s'inspire en cela fortement des langages CCS [71] et CSP [46].

2.1.1 Partie *données*

Cette partie concerne la définition et la manipulation des structures de données. A la différence des langages classiques, LOTOS utilise le formalisme des types abstraits algébriques. Mise à part la définition directe des types de données, LOTOS autorise également l'importation et la combinaison des types, dont la généricité est acquise par l'emploi de types paramétrés par des sortes, des opérateurs et des équations formelles. Ces types formels peuvent alors être instanciés. Notons qu'il est possible de renommer les types, les sortes et les opérations, permettant ainsi une forme de réutilisation.

Dans la construction suivante, on déclare un type T obtenu par combinaison de types existants T_0, ..., T_n, en lui associant des domaines de valeurs (*sorts*), des opérations (*opns*) et des équations (*eqns*) :

type T **is** $T_0, \ldots, T_n$
  sorts $S_0, \ldots, S_p$
  opns $op_0, \ldots, op_q$
  eqns eq
endtype

La boite à outils CADP offre la possibilité de modéliser les types de données en combinant LOTOS et le langage C, comme elle le permet également en LOTOS pur. Par souci d'efficacité, nous avons choisi la combinaison des deux langages (LOTOS, C), ainsi nous avons implémenté plusieurs types (décrivant des aspects matériels ou logiciels) au moyen des types et des fonctions C externes. Ces dernières sont incorporées par les compilateurs CAESAR.adt [21] et CAESAR [20] dans le code généré à partir de la spécification LOTOS.

Exemple 2.1 *On déclare dans la construction suivante le type abstrait* ***ID_Action*** *avec quatre constantes :* Load, Store, Test_and_Set *et* Fetch_and_Decrement. *Chacune de ces constantes correspond à un accès mémoire ou cache pour une lecture, une écriture ou une opération lecture-écriture atomique.* ***ID_Action*** *possède des opé-*

rateurs binaires infixés de comparaison == et <> dont le résultat est de sorte Bool.

```
type ID_Action is Natural
 sorts ID_Action
 opns
   Load : -> ID_Action
   Store : -> ID_Action
   Test_and_Set : -> ID_Action
   Fetch_and_Decrement : -> ID_Action
   _==_ , _<>_ : ID_Action, ID_Action -> Bool
 eqns
   forall A, B : ID_Action
   ofsort Bool
     A == A = true;
     A == B = false;
     A <> B = not (A == B);
endtype
```

L'implémentation *en externe* des types abstraits de LOTOS permet une représentation compacte des données en mémoire et des exécutions déterministes des opérations. Le principe de cette implémentation est d'associer à chaque *sorte* une implémentation sous forme de *type* C et à chaque *opération* une implémentation sous forme de *fonction* C. Par exemple, les chiffres peuvent être codés par des entiers au lieu de les coder par des termes algébriques à base de 0 et de l'opération SUCCESSEUR.
En outre, pour chaque sorte S en LOTOS, il faut définir :

- une *relation de comparaison* qui est une fonction à deux arguments v_1 et v_2 de sorte S renvoyant un résultat booléen *true* si et seulement si v_1 et v_2 sont congrues selon la relation d'égalité définie par les équations algébriques.
- un *itérateur* qui est une macro-définition C prenant comme argument une variable x de sorte S. A l'aide de la variable x l'itérateur décrit successivement les valeurs du domaine de S.
- une *procédure d'impression* qui est une fonction C convertissant une valeur v de sorte S en une chaîne de caractères ASCII et l'imprimant dans un fichier.

Les liens entre les noms des types (resp. fonctions) C et les noms des sortes (resp. opérations) LOTOS correspondantes sont réalisés au moyen de commentaires spéciaux qui constituent un sous-ensemble des commentaires LOTOS. Ils existent sous deux formes :

- Lorsqu'une définition de sorte S est suivie du commentaire :

  (*! **implementedby** ADT_S **printedby** ADT_PRINT_S **external** *)

  cela signifie que la sorte S est implémentée par un type C de nom ADT_S, et que la procédure d'impression de S est la fonction C ADT_PRINT_S.
- Lorsqu'une définition d'une opération f est suivie du commentaire :

(*! **implementedby** ADT_F **constructor external** *)

cela signifie que l'opération f est implémentée par une fonction C de nom ADT_F. La présence du mot *constructor* indique que l'opérateur f est un constructeur.

Nous allons illustrer l'implémentation des types et des opérations LOTOS au moyen des types et des fonctions C externes par des exemples dans les sections qui suivent.

2.1.2 Partie *contrôle*

Cette partie concerne les structures de contrôle. Le contrôle en LOTOS est décrit syntaxiquement par des expressions algébriques dites *expressions comportementales* (ou *comportements*) construites à partir d'un ensemble d'opérateurs (composition séquentielle, composition parallèle, choix non déterministe, gardes, etc.). Ces dernières permettent la description des processus concurrents dont la synchronisation et la communication s'effectuent uniquement par rendez-vous, sans partage de mémoire.

On appelle en LOTOS une *porte* (*gate*) un canal de communication permettant la synchronisation par un rendez-vous entre plusieurs tâches qui se déroulent en parallèle.

Un rendez-vous est dit *simple* s'il n'y a pas d'échange de valeurs (pas d'émission ni de réception de données), il s'agit d'une *synchronisation pure*.

Dans le rendez-vous avec échange de valeurs, la porte est accompagnée d'*offres* d'émission (!val) et/ou d'offres de réception (?var : Sorte). L'exemple 2.2 décrit un rendez-vous sur la porte ACTION sur laquelle on précise le type d'accès (*Load*) à la donnée d'adresse *adr* et on reçoit son contenu dans la variable *var*.

Exemple 2.2 **ACTION** ! Load **of** ID_ACTION ! adr **of** Address ? val :Nat ;

Un rendez-vous est bloquant aussi bien pour l'émission que la réception : l'exécution d'un comportement qui attend un rendez-vous est suspendue et ne reprend qu'après que le rendez-vous a eu lieu.

LOTOS permet de conditionner le rendez-vous par une *garde* qui est soit une expression booléenne $[v_0]$, soit une équation simple $[v_0 = v_1]$. Le rendez-vous n'a pas lieu si la condition définie par la garde n'est pas satisfaite. Par exemple le rendez-vous de l'exemple 2.3 n'a pas lieu si la valeur *val* est différente de 0.

Exemple 2.3 **ACTION** *! Load* **of** *ID_ACTION ! adr* **of** *Address ? val :Nat [val == 0] ;*

Remarque 2.1 *On appelle un rendez-vous sur une porte une* action.

On trouve dans LOTOS deux portes spéciales prédéfinies qui n'appartiennent pas à l'ensemble des portes définies par l'utilisateur [20] :

- Une porte *invisible*, notée "*i*". Elle peut apparaître dans les programmes LOTOS, mais uniquement dans un contexte de l'opérateur " ;". Cette porte dénote une action interne modélisant un comportement *non-observable*.

- Une porte de *terminaison*, notée δ. Cette porte ne peut jamais être employée explicitement dans un programme LOTOS mais elle est utilisée dans la définition sémantique du langage.

La syntaxe des opérateurs LOTOS

Soient P et Q deux expressions comportementales, L l'univers des actions observables et a, a_0, a'_0, a_1, a'_1, ... a_n, a'_n des actions de $L \cup \{i\}$ et, soient $A \subset L$ et $A' \subset L$ deux ensembles d'actions, avec $n \geq 0$ et,

- $A = \{a_0, a_1, ... a_n\}$
- $A' = \{a'_0, a'_1, ... a'_n\}$
- $A := A = \{a_0 := a'_0, a_1 := a'_1, ... a_n := a'_n\}$

Le tableau 2.1 illustre la syntaxe des opérateurs LOTOS que nous avons utilisé dans notre modélisation (il existe bien d'autres opérateurs).

Interaction	**stop**
Terminaison avec succès	**exit**
Action-préfixe	$a;\ P$
Choix	$P\ [\ ]\ Q$
Composition séquentielle	$P >> Q$
Composition parallèle	$P\|[A]\|Q$
Renommage	**rename** $A := A'$ **in** P
Abstraction	**hide** A **in** P

TAB. 2.1 – *La syntaxe du langage LOTOS*

Le langage LOTOS possède des écritures simplifiées pour la composition parallèle :

- $P\ |||\ Q$ exprime $P\ |[\emptyset]|\ Q$: les deux comportements P et Q sont exécutés de manière complètement indépendante. Il s'agit d'une composition parallèle totalement *asynchrone*.
- $P\ ||\ Q$ exprime $P\ |[L]|\ Q$: les deux comportements P et Q sont exécutés en parallèle de manière entièrement synchronisée, ils doivent se synchroniser sur toutes leurs interactions. Il s'agit d'une composition parallèle totalement *synchrone*.

La sémantique dynamique de LOTOS

LOTOS possède une sémantique opérationnelle basée sur une relation de transition. Cette relation est définie par un système de dérivation dont les règles sont accompagnées d'explications en langage naturel. Elle spécifie comment construire le système de transitions étiquetées STE (aussi appelé *graphe* ou *automate*) correspondant à une spécification LOTOS.

La sémantique associée à chaque expression de comportement P décrite en LOTOS un état dans le graphe, où la relation de transition est définie comme étant la plus petite relation qui satisfait l'une des règles du tableau 2.2. Tout comportement représente un automate à un nombre fini ou infini d'états.

$$\frac{}{\textbf{exit} \xrightarrow{\delta} \textbf{stop}} \qquad \frac{}{a;P \xrightarrow{a} P}$$

$$\frac{P \xrightarrow{a} P'}{P|[A]|Q \xrightarrow{a} P'|[A]|Q} \; (a \notin A \cup \{\delta\}) \qquad \frac{P \xrightarrow{a} P'}{P[]Q \xrightarrow{a} P'} \; (a \notin A \cup \{\delta\})$$
$$Q|[A]|P \xrightarrow{a} Q|[A]|P' \qquad Q[]P \xrightarrow{a} P'$$

$$\frac{P \xrightarrow{a} P',\ Q \xrightarrow{a} Q'}{P|[A]|Q \xrightarrow{a} P'|[A]|Q} \; (a \in A \cup \{\delta\}) \qquad \frac{P \xrightarrow{a} P'}{P >> Q \xrightarrow{a} P' >> Q} \; (a \neq \delta)$$

$$\frac{P \xrightarrow{a} P'}{\textbf{hide } A \textbf{ in } P \xrightarrow{a} \textbf{hide } A \textbf{ in } P'} \; (a \notin A) \qquad \frac{P \xrightarrow{a} P'}{\textbf{hide } A \textbf{ in } P \xrightarrow{i} \textbf{hide } A \textbf{ in} P'} \; (a \in A)$$

$$\frac{P \xrightarrow{a} P'}{\textbf{rename } A := A' \textbf{ in } P \xrightarrow{a} \textbf{rename } A := A' \textbf{ in } P} \; (a \notin A) \qquad \frac{P \xrightarrow{a} P'}{\textbf{rename } A := A' \textbf{ in } P \xrightarrow{a'} \textbf{rename } A := A' \textbf{ in } P} \; (a := a' \in A := A')$$

$$\frac{P \xrightarrow{\delta} P'}{P >> Q \xrightarrow{i} P' >> Q} \; (a \neq \delta)$$

TAB. 2.2 – *La sémantique opérationnelle du langage LOTOS*

Remarque 2.2 *Le renommage n'est pas une opération du LOTOS standard, il est effectué par le passage de paramètres porte dans les processus.*

L'opérateur "**rename** $a_0 := a'_0, a_1 := a'_1, ...a_n := a'_n$ **in** P" dénote le comportement obtenu en modifiant les actions effectuées par P : l'action a_i est renommée en a'_i pour $i \in \{0, 1, ..., n\}$, les autres actions restent inchangées. Quant à l'opérateur "**hide** A **in** P" il représente un cas particulier de "**rename in**", il dénote le renommage de toutes les actions de l'ensemble A en action interne i.

Les processus LOTOS

Par analogie aux langages algorithmiques qui nomment généralement un bloc d'instructions *procédure*, LOTOS désigne un comportement au moyen

d'une définition de *processus* (**process**). Un processus est un objet qui dénote un comportement, il peut être paramétré par une liste de portes sur lesquelles il peut effectuer des rendez-vous, et/ou une liste de variables typées [20].

La construction suivante définit un processus PROC paramétré par les portes G_0, ..., G_n et la liste des variables formelles v_0, ..., v_m de sortes respectives T_0, ..., T_m. Le *corps* du processus est le comportement B. Le non-terminal *func* spécifie la fonctionnalité du comportement B : **exit** pour une terminaison avec succès, et **noexit** pour une exécution sans fin (cas d'une boucle infinie ou d'un blocage). Chaque non-terminal $bloc_i$ dénote une définition d'un sous-processus dont la visibilité est limitée à la définition du processus PROC.

process PROC [G_0, ..., G_n] (v_0 :T_0, ... , v_m :T_m) : *func* :=
B
where
$bloc_0$
...
$bloc_q$
endproc

Exemple 2.4 *Le processus* SEND_PROC *défini ci-dessous est paramétré par une seule porte* ACTION *et ayant pour fonctionnalité* **exit**. *Son comportement consiste en un rendez-vous sur la porte* ACTION *pour une demande de lecture du contenu de l'adresse* adr, *suivi d'un test : si la valeur lue est égale à* 0 *alors le contenu de l'adresse* adr *est positionné à 1 (par un rendez-vous sur la porte* ACTION *pour une écriture), sinon fin de l'exécution.*

```
process SEND_PROC [ACTION] : exit :=
 ACTION ! Load of ID_ACTION ! adr of Address ? val :Nat;
 (
  [val == 0] -> ACTION ! Store of ID_ACTION ! adr of Address ! 1 of Nat;
                exit
   []
  [val <> 0] -> exit
 )
endproc
```

Les spécifications LOTOS

LOTOS possède une structure de blocs imbriqués : chaque définition de processus peut contenir des définitions de processus ou de types qui lui sont locaux. Cependant, incorporer des types et des processus différents n'est possible qu'avec la définition de *spécification* :

specification SPEC [$G_0, \ldots, G_n$] (v_0 :T_0, ... , v_m :T_m) : *func* :=
type$_0$, ..., type$_p$
behaviour
B
where
bloc$_0$
...
bloc$_q$
endspec

Chaque non-terminal $type_i$ dénote une définition de type dont la visibilité s'étend à toute la spécification. Le comportement B est le corps de la spécification. Le non-terminal *func* spécifie la fonctionnalité de B. Chaque non-terminal $bloc_0$ dénote une définition de type ou de processus.

L'identificateur SPEC joue le rôle d'un commentaire ; ce n'est pas un identificateur de processus, il ne peut donc pas être utilisé dans une instanciation.

2.2 La théorie des chaînes de Markov interactives

Un processus stochastique décrit l'évolution temporelle de l'état d'un système à l'aide de variables aléatoires et de lois de probabilités.

D'un point de vue observation, un processus stochastique est constitué par l'ensemble de ses réalisations. Une réalisation est obtenue par l'expérience qui consiste à enregistrer une suite d´évènements au cours du temps. Le caractère aléatoire de l'évolution se montre par le fait que la répétition de l'expérience conduit à une autre séquence temporelle.

Définition 2.1 *Un **processus stochastique** $\{X_t, t \in T\}$ est une collection de variables aléatoires indexées par un paramètre t et définies sur un même espace de probabilités. Le paramètre est généralement interprété comme le temps et appartient à un ensemble ordonné T.*

On distingue deux classes de processus stochastiques en fonction de l'ensemble T :

- *processus stochastique à temps discret* si $T \subset \mathbb{N}$
- *processus stochastique à temps continu* si $T \subset \mathbb{R}_+$

Définition 2.2 *L'ensemble de toutes les valeurs que peuvent prendre les variables définissant un processus stochastique est appelé **espace d'états** du processus, il est généralement noté par S. Si cet ensemble est fini ou dénombrable alors le processus est appelé **chaîne**.*

Un processus stochastique permet de modéliser l'état d'un système évoluant d'une manière aléatoire dans le temps. L'observation du système au cours du temps peut se faire de manière continue ou discrète et son état à l'instant t est donné par la variable aléatoire X_t.

Définition 2.3 *Un processus stochastique $\{X_t, t \geq 0\}$ défini sur un espace d'états S satisfait la **propriété de Markov** si, pour toute séquence d'instant $t_{n+1} > t_n > t_{n-1} >$*

$t_{n+2} > \ldots > t_0$ et tout sous-ensemble d'états $I \subseteq S$, il est vrai que

$$Prob\{X_{t_{n+1}} \in I \mid X_{t_n} = i_n,\ X_{t_{n-1}} = i_{n-1},\ X_{t_{n-2}} = i_{n-2}, \ldots,\ X_{t_0} = i_0\} = Prob\{X_{t_{n+1}} \in I \mid X_{t_n} = i_n\} \quad (2.1)$$

Un processus stochastique vérifiant la propriété 2.1 est appelé un ***processus de Markov*** *ou* ***processus markovien****.*

L'évolution de l'état d'un système peut être modélisée par un processus markovien si, pour tout instant t, l'état courant X_t *résume, à lui seul, tout l'historique du système susceptible d'influencer son évolution future.*

2.2.1 Les chaînes de Markov à temps continu (CTMC)

Soit S un ensemble fini ou, plus généralement, dénombrable d'états et $\{X_t, t \geq 0\}$ un processus stochastique à temps continu prenant ses valeurs dans S. Ce processus vérifie la propriété de Markov, si pour tout $t_n + \Delta t > t_n > t_{n-1} > t_{n+2} > \ldots > t_0$ et tout état $j \in S$ il est vrai que

$$Prob\{X_{t_n+\Delta t} = j \mid X_{t_n} = i,\ X_{t_{n-1}} = i_{n-1},\ X_{t_{n-2}} = i_{n-2}, \ldots,\ X_{t_0} = i_0\} = Prob\{X_{t_n+\Delta t} = j \mid X_{t_n} = i\} = Prob\{X_{\Delta t} = j \mid X_0 = i\} \quad (2.2)$$

Cette expression est une réécriture de la propriété de Markov 2.1, adaptée à l'aspect fini ou dénombrable de l'espace d'états S du processus.

Définition 2.4 *Une* ***chaîne de Markov à temps continu*** *est un processus stochastique $\{X_t, t \geq 0\}$ satisfaisant les propriétés suivantes :*

- *le processus est à temps continu ;*
- *l'espace d'état S est fini ou dénombrable ;*
- *le processus satisfait la propriété de Markov 2.2.*

On dit qu'une chaîne de Markov est *homogène* dans le temps si la probabilité d'effectuer une transition d'un état à un autre est indépendante de l'instant auquel à lieu cette transition. En d'autres termes, la probabilité conditionnelle $Prob\{X_{t+u} = j \mid X_t\}$ apparaissant dans 2.1 ne dépend pas de l'instant t, c'est à dire :

$$Prob\{X_{t+u} = j \mid X_t = i\} = Prob\{X_u = j \mid X_0 = i\} \quad (2.3)$$

• *Probabilité de transition*

Soit $\{X_t, t \geq 0\}$ une CTMC homogène avec un espace d'état S fini ou dénombrable. Nous avons :

$$p_{ij}(t) = Prob\{X_t = j \mid X_t = i\} \quad (2.4)$$

La probabilité $p_{ij}(t)$ est appelée la *probabilité de transition* (ou de *passage*) de l'état i à l'état j en une seule étape. Ces probabilités peuvent être rangées dans des *matrices de transition* $P(t) = (p_{ij}(t))$ aux lignes et aux colonnes indexées par les états de S.

Les probabilités de transition dans les CTMC sont spécifiées par une famille de matrices stochastiques $\{P(t), t \geq 0\}$, vérifiant les deux propriétés suivante pour tout $t \geq 0$:

$$p_{ij}(t) \geq 0, \ \ \forall i,j \in S \quad \text{et} \quad \sum_{j \in S} p_{ij} = 1, \ \ \forall i \in S$$

• ***Temps de séjour***

Considérons une chaîne de Markov à temps continu $\{X_t, t \geq 0\}$ et, pour tout $t \geq 0$, notons D le temps de séjour du processus à l'état qu'il occupe à l'instant t. En d'autres termes, D représente la durée de l'intervalle de temps avant que le processus quitte l'état qu'il occupe à l'instant t. Le théorème suivant montre que la variable aléatoire D est *sans mémoire* et par conséquent distribuée selon la *loi exponentielle*.

Théorème 2.1 *Pour tout $i \in S$ et $t \geq 0$,*

$$Prob\{D \leq u \mid X_t = i\} = 1 - e^{-\lambda_i u}, \ u \geq 0$$

avec $\lambda_i \in \mathbb{R}_+ \cup \{+\infty\}$. Si $\lambda_i = +\infty$ alors $e^{-\lambda_i u} = 0$ pour tout $u \geq 0$

Une CTMC se retrouvant à l'instant t dans l'état i reste dans cet état pendant une durée aléatoire D distribuée selon une loi exponentielle d'un paramètre λ_i, avec :

λ_i : le taux selon lequel on quitte l'état i (indépendant de l'état i)

$1/\lambda_i$: le temps *moyen* passé dans l'état i

Les lois exponentielles sont caractérisées par un ensemble de propriétés [41] que nous citons ci-dessous :

Propriété 2.1 *Une distribution exponentielle $Prob\{D \leq t\} = 1 - e^{-\lambda t}$, est caractérisée par un seul paramètre appelé le taux (*rate*) de la distribution $\lambda > 0$.*

Propriété 2.2 *Les distributions exponentielles sont des distributions de probabilités sans mémoire (*memory less*)*

$$Prob\{D \leq t + t_0 | D > t_0\} = Prob\{D \leq t\}$$

Propriété 2.3 *Le minimum de distributions exponentielles est une distribution exponentielle d'un paramètre représentant la somme des taux*

$$Prob\{min(D_1, D_2) \leq t\} = 1 - e^{-(\lambda_1 + \lambda_2)t}$$

avec D_1 (resp. D_2) une distribution exponentielle d'un taux λ_1 (resp. λ_2)

Propriété 2.4 *La probabilité que D_1 est inférieure à D_2 (et vice versa) peut être évaluée directement depuis leur taux λ_1 et λ_2*

$$Prob\{D_1 < D_2\} = \frac{\lambda_1}{\lambda_1 + \lambda_2} \quad et \quad Prob\{D_2 < D_1\} = \frac{\lambda_2}{\lambda_1 + \lambda_2}$$

Propriété 2.5 *La nature continue de la distribution exponentielle assure que la probabilité pour que plusieurs durées s'écoulent au même instant est 0.*

Remarque 2.3 *Dans le cas des variables aléatoires continues, la loi de probabilité associe une probabilité à chaque ensemble de valeurs définies dans un intervalle donné, par conséquent la probabilité d'une durée égale à une valeur fixe d est nulle : $Prob\{D = d\} = 0$, car il est impossible d'observer exactement cette valeur. Et d'autre part, l'occurrence des événements est strictement ordonnée avec le temps, et la simultanéité de deux événements ne se produit donc jamais, ce qui justifie par conséquent la propriétés 2.5.*

• ***Le choix d'un nouvel état***

Lorsque une CTMC quitte un état i, elle se déplace dans un état j ($\neq i$) avec une probabilité q_{ij} indépendante de l'instant t et du temps de séjour D à l'état i :

$$Prob\{X_{t+D} = j \,|X_t = i\} = q_{ij} \quad j \in S$$

$$\text{avec} \quad q_{ij} \geq 0, \quad q_{ii} = 0 \quad \text{et} \quad \sum_{j \in S} q_{ij} = 1$$

La probabilité q_{ij} ne dépend pas de t car la chaîne est homogène. De plus, le processus étant markovien, cette probabilité est également indépendante du temps de séjour (D) dans l'état i.

La suite des états atteints après chaque transition de la chaîne $\{X_t, t \geq 0\}$ définit, une chaîne de Markov à temps discret de matrice de transition $Q = (q_{ij})$. Cette chaîne est appelée la *chaîne de Markov sous-jacente* (ou *induite*) associée à $\{X_t, t \geq 0\}$.

• ***Matrice génératrice***

Le théorème suivant définit la relation existante entre les paramètres λ_i, les probabilités q_{ij} et les probabilités de transition p_{ij} de la CTMC $\{X_t, t \geq 0\}$.

Théorème 2.2

Pour tout $i \in S$,

$$\lim_{t \to 0} \frac{p_{ii}(t) - 1}{t} = -\lambda_i \tag{2.5}$$

Pour tout $i, j \in S$, avec $i \neq j$,

$$\lim_{t \to 0} \frac{p_{ij}(t)}{t} = \lambda_i q_{ij} \tag{2.6}$$

Le théorème 2.2 montre que les probabilités de transition $p_{ij}(t)$ admettent une dérivée à droite en $t = 0$. Définissant ainsi les quantités :

$$a_{ij} = \begin{cases} -\lambda_i & \text{si } i = j \\ \lambda_i q_{ij} & \text{si } i \neq j \end{cases} \tag{2.7}$$

La matrice $A = (a_{ij})$ est appelée la *matrice génératrice* ou le *générateur* de la chaîne de Markov à temps continu. La quantité a_{ij} est appelée l'*intensité de transition* entre i et j et représente la fréquence avec laquelle un processus se trouvant dans l'état i effectue une transition vers l'état j. Quant à la quantité $-a_{ii}$, elle est appelée l'*intensité de passage* hors de l'état

i et représente la fréquence avec laquelle un processus quitte l'état i qu'il occupe actuellement.

La matrice génératrice A joue un rôle essentiel dans l'analyse d'une CTMC. Elle caractérise, en effet, de manière unique la famille $\{P(t), t \geq 0\}$ des matrices de transitions définissant le processus.

• ***Représentation graphique***

Souvent, la matrice génératrice A d'une CTMC est représentée par un graphe orienté $G = (S, E)$, dont les sommets correspondent aux états du processus et dont l'ensemble des arcs est donné par :

$$E = \{(i, j) \mid a_{ij} > 0\}$$

Dans le but d'avoir une représentation graphique fixe de CTMC, indépendante des probabilités q_{ij}, nous avons opté pour la représentation proposée dans [41] qui consiste en la définition d'une relation de transition sur l'espace d'états S basée uniquement sur le taux des transitions : $i \xrightarrow{\lambda_i} j$. Cette relation est appelée *relation de transition markovienne* ou *relation de transition stochastique*.

Ainsi la CTMC est représentée par un système de transitions markoviennes défini comme suit :

Définition 2.5 *Un système de transition markovien est un couple (S, TS) avec,*

S est un ensemble fini non vide d'états,

TS est la relation de transition markovienne et un sous-ensemble de $S \times \mathbb{R}_+ \times S$

• ***Classification des états***

La classification des états de CTMC joue un rôle important dans l'étude de son comportement à long terme.

Définition 2.6 *Soient i et j deux états d'une CTMC. L'état j est* ***accessible*** *depuis l'état i s'il existe $t \geq 0$ tel que $p_{ij}(t) > 0$.*
En se basant sur la représentation graphique de la chaîne, l'état j est ***accessible*** *depuis l'état i soit si $i = j$, soit s'il existe, dans le graphe représentatif de la chaîne, au moins un chemin de i à j.*

Définition 2.7 *Deux états i et j d'une CTMC* ***communiquent*** *s'il existe $t_1 \geq 0$ et $t_2 \geq 0$ tel que $p_{ij}(t_1) > 0$ et $p_{ji}(t_2) > 0$.*
En se basant sur la représentation graphique de la chaîne, les états i et j ***communiquent*** *s'ils appartiennent à la même composante fortement connexe du graphe représentatif de la chaîne.*

On dit qu'une CTMC est *irréductible* si tous ses états communiquent et *réductible* dans le cas contraire.

Dans un graphe G, tout sommet appartient exactement à une même composante fortement connexe[1]. L'ensemble de ces composantes $C_1, \ldots, C_r$ définit donc une partition des sommets du graphe. On peut associer au graphe G un graphe G_R dont les sommets correspondent aux composantes fortement connexes de G et où un arc relie deux sommets u et v s'il existe $i \in C_u$ et $j \in C_v$ avec $p_{ij} > 0$. Le graphe G_R est appelé le *graphe réduit* de la CTMC et contient toutes les informations concernant l'accessibilité entre les états de la chaîne.

Définition 2.8 *Une classe d'états d'un graphe est **récurrente** si elle correspond à un sommet sans successeur de G_R. Dans le cas contraire, la classe est **transitoire**.*

Définition 2.9 *Un état est*

***récurrent** s'il appartient à une classe récurrente ;*

***transitoire** s'il appartient à une classe transitoire ;*

***absorbant** s'il forme à lui seul une classe récurrente.*

• ***La distribution stationnaire***

Soit $\{X_t, t \geq 0\}$ une CTMC et A sa matrice génératrice. Si la chaîne est irréductible, alors les propriétés suivantes sont vérifiées :

1. Les matrices de transition $P(t)$ convergent vers une même matrice P^* lorsque $t \to \infty$.
2. Les lignes de la matrice P^* sont toutes égales à un même vecteur π^*
3. Nous avons,
   - soit
     $$\pi^* = 0, \quad \forall j \in S$$
     et la chaîne est *transitoire* ou *récurrente nulle*
   - soit
     $$\pi^* > 0, \quad \forall j \in S$$
     et la chaîne est *récurrente non-nulle* ou *récurrente nulle*
4. Si la chaîne est *récurrente non-nulle*, π^* est une distribution de probabilités et l'unique solution du système

$$\begin{aligned} \pi A &= 0 \\ \pi 1 &= 1 \end{aligned} \tag{2.8}$$

La limite π^* des probabilités de transition d'une CTMC est appelée la distribution *stationnaire* de la chaîne (dite aussi la distribution *invariante*).

Si $\pi(0)$ est la distribution de l'état initial d'une CTMC, nous avons :

$$\lim_{t\to\infty} \pi_j(t) = \lim_{t\to\infty} Prob\{X_t = j\} = \pi_j^*, \quad \forall j \in S \tag{2.9}$$

La quantité π_j^* représente la proportion de temps passé dans l'état j.

[1] En théorie des graphes, une composante fortement connexe d'un graphe orienté G est un sous-graphe maximal de G tel que pour toute paire de sommets u et v dans ce sous-graphe, il existe un chemin de u à v et un chemin de v à u. Un graphe est dit fortement connexe s'il est formé d'une seule composante fortement connexe. De manière générale, un graphe se décompose de manière unique comme union de composantes fortement connexes disjointes

Le théorème 2.3 suivant montre qu'il est possible d'estimer les performances d'un système modélisé par une CTMC ergodique à l'aide d'une seule réalisation du processus.

Théorème 2.3 *Soit $\{X_t, t \geq 0\}$ une chaîne de Markov à temps continu ergodique et π^* sa distribution stationnaire. Soit f une fonction réelle définie sur l'espace d'états S de la chaîne et vérifiant :*

$$\sum_{i \in S} \pi_i^* |f(i)| < \infty$$

Alors, pour une réalisation quelconque de la chaîne,

$$\lim_{t \to \infty} \frac{1}{t} \int_0^t f(X_u)\, du = \sum_{i \in S} \pi_i^* f(i)$$

Exemple 2.5 *Soit un processus P_0 effectuant une suite infinie d'accès en écriture sur une variable privée qui se trouve initialement en mémoire. Avant tout accès, le processus doit attendre un certain délai. Ainsi le processus peut se trouver dans deux états, un* état d'accès *modélisant la réalisation d'un accès, et un* état d'attente *modélisant l'attente. Cependant, le protocole de cohérence de caches impose des latences différentes aux accès, notamment le premier accès. Comme il s'agit d'une variable privée, le premier accès implique un transfert depuis la mémoire avec une latence moyenne μ_M. Tous les accès qui suivent impliquent des transferts* internes *(voir la section 1.1.2) avec une latence moyenne μ_I. Le processus reste dans l'état attente pendant une durée moyenne μ_{wait}. Nous avons :*

$$\mu_M = \frac{1}{\lambda_M} = 0,200\ \mu s, \quad \mu_I = \frac{1}{\lambda_I} = 0,00111\ \mu s, \quad \mu_{wait} = \frac{1}{\lambda_{wait}} = 0,00001\ \mu s$$

On distingue en effet 4 états. Les états 0 et 1 correspondent à la première attente et le premier accès mémoire respectivement. Les états 2 et 3 correspondent aux attentes pour des accès internes, comme illustré dans la figure 2.1.

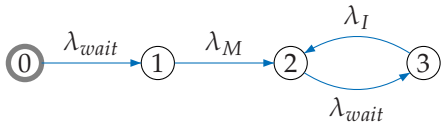


FIG. 2.1 – *Comportement stochastique d'une suite infinie d'accès en écriture*

A chaque changement d'état, le processus P_0 se trouve dans un état j avec une probabilité q_{ij}

$$Q = (q_{ij}) = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Soit X_t l'état modélisant la réalisation du processus P_0 à l'instant t (accès ou attente). La famille $\{X_t, t \geq 0\}$ définit une CTMC de matrice génératrice

$$A = (a_{ij}) = \begin{pmatrix} 0 & 100000 & 0 & 0 \\ 0 & 0 & 0,005 & 0 \\ 0 & 0 & 0 & 100000 \\ 0 & 0 & 900 & 0 \end{pmatrix}$$

La distribution stationnaire de cette chaîne, solution du système

$$\pi A = 0$$
$$\pi 1 = 1$$

est

$$\pi^* = \begin{pmatrix} 0 & 0 & \frac{9}{1009} & \frac{1000}{1009} \end{pmatrix}$$

Chaque probabilité π_i^* représente la proportion du temps pendant laquelle le processus P_0 se trouve à l'état i.

Le débit moyen des accès réalisé par le processus P_0 est égal à

$$DM = \pi_1^* f_1 \ + \ \pi_3^* f_3 = \frac{900000}{1009} \approx 891,972$$

avec

$$f_0 = \lambda_{wait},\ f_1 = \lambda_M,\ f_2 = \lambda_{wait},\ f_3 = \lambda_I$$

Notre étude de performances est basée sur l'évaluation du débit DM, mais cette fois on s'intéresse au débit de l'exécution de l'algorithme MPI, afin de déduire par la suite sa *latence moyenne* à long terme. Par exemple dans le cas de l'algorithme de *ping-pong* il s'agit d'évaluer la latence moyenne d'échange d'un message.

Nous détaillons tout le mécanisme d'évaluation des performances dans le chapitre 5.

2.2.2 Algèbre de processus × chaînes de Markov = IMC

IMC (*Interactive Markov Chains*) [41] est une algèbre de processus stochastiques qui peut être considérée comme une extension de l'algèbre de processus classique pour la description des chaînes de Markov, en associant à chaque action une variable aléatoire représentant la durée de cette action. Ces variables aléatoires sont exponentiellement distribuées, ce qui établit clairement une relation entre le modèle de l'algèbre

L'idée de base consiste à étendre la syntaxe de l'algèbre de processus avec un préfixe temporel, dit *taux markovien*.

L'écriture suivante :

$$\lambda;\ P \quad \text{avec} \ \ \lambda \in \mathbb{R}_+$$

exprime que le comportement P se produit après un délai d'attente t déterminé par une variable aléatoire continue X qui suit une loi exponentielle de paramètre λ.
La probabilité pour réaliser P après une durée t est donnée par l'équation suivante :

$$Prob\{X \leq t\} = \begin{cases} 1 - e^{\lambda t} & \text{si } t > 0 \\ 0 & \text{sinon} \end{cases}$$

L'espérance $1/\lambda$ de la variable aléatoire X représente la durée moyenne avant la réalisation de P.

Définition 2.10 *Une IMC est un système de transitions étiquetées (S, A, TI, TS, s_0), où :*

- *S est un ensemble fini non vide d'états,*
- *A est un ensemble fini d'actions,*
- *$TI \subseteq S \times A \times S$ est un ensemble de* Transitions Interactives *dites aussi des transitions ordinaires. Elles expriment l'interaction avec l'environnement. La transition $(s_1, a, s_2) \in TI$, notée également par $s_1 \xrightarrow{a} s_2$, signifie que le système peut passer de l'état s_1 à l'état s_2 en exécutant l'action a.*
- *$TS \subseteq S \times \mathbb{R}_+ \times S$ est un ensemble de* Transitions Stochastiques *dites aussi des transitions markoviennes ou encore des taux markoviens. Elles expriment l'écoulement du temps.*
  Une transition stochastique $(s_1, \lambda, s_2) \in TS$, notée également par $s_1 \xrightarrow{\lambda} s_2$, signifie que le système peut passer de l'état s_1 à l'état s_2 en une durée aléatoire, distribuée selon une loi exponentielle de paramètre λ.
- *$s_0 \in S$ est l'état initial*

Dans une IMC

Les IMC constituent un formalisme très général pour plusieurs aspects [23] :

- Il décrit un STE en l'unique présence d'actions interactives et une CTMC en l'unique présence de taux markoviens. Ceci est dû à la présentation des aspects fonctionnels et stochastiques par des transitions distinctes (transitions interactives et stochastiques).
- Il autorise le non-déterminisme dans les états : plusieurs (≥ 2) transitions identiques sortantes d'un même état. Ceci est d'une grande importance pour la génération automatique de IMC depuis un langage haut niveau comme l'algèbre de processus.

La sémantique des transitions interactives correspond aux règles définies dans le tableau 2.2, quant à la sémantique des transitions stochastiques qui décrivent l'écoulement de temps est décrite dans le tableau 2.3, avec la nouvelle interprétation des opérateurs de la composition parallèle et du choix.

$$\frac{}{\lambda; P \xrightarrow{\lambda} P} \qquad \frac{P \xrightarrow{\lambda} P'}{\textbf{rename } A := A' \textbf{ in } P \xrightarrow{\lambda} \textbf{rename } A := A' \textbf{ in } P'}$$

$$\frac{P \xrightarrow{\lambda} P'}{\begin{array}{l} P|[A]|Q \xrightarrow{\lambda} P'|[A]|Q \\ Q|[A]|P \xrightarrow{\lambda} Q|[A]|P' \end{array}} \qquad \frac{P \xrightarrow{\lambda} P'}{\begin{array}{l} P[]Q \xrightarrow{\lambda} P' \\ Q[]P \xrightarrow{\lambda} P' \end{array}}$$

$$\frac{P \xrightarrow{\lambda} P'}{\textbf{hide } A \textbf{ in } P \xrightarrow{\lambda} \textbf{hide } A \textbf{ in } P'} \qquad \frac{P \xrightarrow{\lambda} P'}{P >> Q \xrightarrow{\lambda} P' >> Q}$$

TAB. 2.3 – *La sémantique opérationnelle des IMC*

• ***La composition parallèle asynchrone (|||)***

Les processus parallèlement composés avec l'opérateur $|||$ peuvent être différés de façon complètement indépendante [44, 42]. Ceci est justifié par la propriété sans-mémoire de la distribution exponentielle (un exemple de démonstration illustré dans [41]).

Exemple 2.6 *Soient a et b deux actions interactives et $\lambda > 0$ et $\mu > 0$ deux taux markoviens :*

$$\lambda;\ a;\ \textbf{\textit{stop}} \ ||| \ \mu;\ b;\ \textbf{\textit{stop}}$$

On suppose que la durée pour entamer l'action a, notée par D_λ, s'achève en premier (avec le taux λ). Selon la propriété sans-mémoire, le temps restant pour activer l'action b correspond exactement au temps restant *avant* que D_λ prenne fin et il est déterminé par une distribution exponentielle de taux μ.

La composition parallèle avec l'opérateur $|||$ est réalisée sans aucun ajustement des taux markoviens. Le non-déterminisme ne peut pas apparaitre [41] parce que, d'une part, chaque transition possède sa propre probabilité de sélection, et d'autre part, la probabilité de choisir plusieurs transitions en même temps est égale à 0 en raison de la continuité de la distribution exponentielle (voir la propriété 2.5).

• ***La composition parallèle synchrone (|[A]|, $A \neq \emptyset$)***

Une synchronisation ne peut avoir lieu que si tous les participants sont prêts :

Exemple 2.7 *Soient a et b deux actions interactives, $\lambda > 0$ et $\mu > 0$ deux taux markoviens :*

$$\lambda;\ a;\ \textbf{\textit{stop}} \ ||[a]|| \ \mu;\ a;\ \textbf{\textit{stop}}$$

L'interprétation naturelle et stochastique de cette synchronisation est la suivante :

La synchronisation sur l'action a n'est possible qu'après un délai déterminé par la distribution exponentielle de taux λ ***et*** *un délai déterminé par la distribution exponentielle de taux μ, c'est à dire, après le maximum des deux variables aléatoires exponentiellement distribuées de taux respectifs λ et μ.*

Cependant, il est impossible d'utiliser le maximum, car la classe des distributions exponentielles n'est pas fermée par le maximum, c'est-à-dire que le maximum de distributions exponentielles n'est pas une distribution exponentielle [12, 73].

Plusieurs approches ont été proposées dans les algèbres de processus stochastiques (SPA) pour l'évaluation du taux résultant d'une synchronisation, par exemple TIPP [31] propose le produit des taux, EMPA [2] interdit ce genre de synchronisation et nécessite la détermination du taux par un seul participant à la synchronisation et PEPA [45] évalue le maximum de la moyenne des deux délais tout en intégrant les capacités individuelles de synchronisation des processus. En plus du manque d'une interprétation claire et stochastique, l'application de ces approches est limitée à des cas particuliers.

Les IMC résolvent ce problème en maintenant l'interprétation de la synchronisation (le maximum des délais) par la séparation explicite entre les taux markoviens et les actions interactives [41, 4, 44, 40].

La figure 2.2, illustre comment la représentation des taux markoviens λ et μ et l'action a dans les comportement E_1 et E_2 permet à l'action a de survenir après l'écoulement des deux délais correspondant aux taux λ et μ dans $E_1 \,||[a]||\, E_2$.

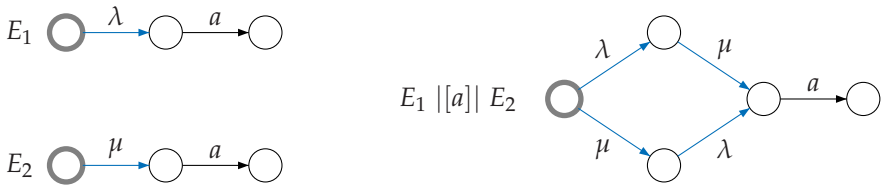


FIG. 2.2 – *Exemple de synchronisation en IMC*

Cette distinction mène à un comportement dont deux phases distinctes sont mixées. Les phases dite *directes* durant lesquelles, une ou plusieurs actions interactives surviennent (ainsi que leur changement d'état correspondant) sans aucun taux markovien, alternées par des phases dite *continues*, décrivant l'écoulement du temps par des taux markoviens et durant lequel aucune action interactive ne peut se produire.

Cette séparation entre les phases directes et continues est similaire à celle des algèbres de processus temporisées (TPA) proposées dans [72, 96, 84]

Notons que contrairement aux IMC, les autres algèbres de processus stochastiques comme TIPP, EMPA et PEPA considèrent les actions et les taux markoviens comme une seule entité, comme décrit dans la figure 2.3.

IMC Autres SPA

FIG. 2.3 – *La représentation des transitions interactives et markoviennes dans IMC et les SPA*

• ***Le choix*** **([ ])**

L'interprétation stochastique du choix est basée sur la notion de "*race condition*" [44] qui consiste à sélectionner l'action la *plus rapide*. Cependant la sélection est non-déterministe s'il est impossible de déterminer le processus le plus rapide.

On distingue le choix entre :

- **taux markoviens**. Par exemple : λ; *stop* [ ] μ; *stop*. L'application de la notion de *race condition* dans ce cas revient à l'évaluation du minimum des deux distributions exponentielles de taux respectives λ et μ, qui correspond à une distribution exponentielle de taux $\lambda + \mu$.
- **taux markovien et action interne**. Par exemple : λ; *stop* [ ] *i*; *stop*. L'action interne *i* est traitée en tant qu'action *instantanée*, puisque rien ne peut ni la prévenir (aucune interaction avec son environnement) ni la retarder, étant donné qu'elle n'est pas conditionnée par un délai markovien, par conséquent le choix revient à la sélection du comportement *i*; *stop*. Cette notion est connue sous le nom du *progrès maximal* (*maximal progress*), elle est largement utilisée dans les algèbres de processus temps-réel (RTPA) [38, 97, 74].
- **actions internes**. Par exemple : *i*; *stop* [ ] *i*; *stop*. Il est impossible d'appliquer la notion de *race condition*, puisque les deux actions sont instantanées, il s'agit d'un comportement non-déterministe. Généralement, ce type de choix est éliminé par la réduction stochastique qui applique la relation de bisimulation faible (voir le tableau 2.4). Cependant, il n'existe pas de solutions standard pour traiter le non-déterminisme dans le cas ou la réduction stochastique est incapable de l'éliminer, étant donné que cela dépend en premier lieu du principe suivi dans la spécification du système.
- **actions interactives**. Par exemple *a*; *stop* [ ] *b*; *stop*. Généralement, ce choix est résolu par des synchronisations (interactions) avec l'environnement, sinon on dit qu'il s'agit d'un choix non-déterministe.

Remarque 2.4 *La propriété du progrès maximal est réservée aux actions internes, car l'environnement peut avoir des influences temporelles sur les actions.*

Le tableau 2.4 décrit un ensemble d'axiomes permettant l'élimination des actions internes dans une IMC.

A1)	$a;\ P\ [\ ]\ a;\ P$	$=$	$a;\ P$
A2)	$P\ [\ ]\ i;\ P$	$=$	$i;\ P$
A3)	$a;\ i;\ P$	$=$	$a;\ P$
A4)	$\lambda;\ P\ [\ ]\ i;\ Q$	$=$	$i;\ Q$
A5)	$\lambda;\ i;\ P$	$=$	$\lambda;\ P$
A6)	$a;\ (P\ [\ ]\ i;\ Q)\ [\ ]\ a;\ Q$	$=$	$a;\ (P\ [\ ]\ i;\ Q)$
A7)	$\lambda;\ P\ [\ ]\ \mu;\ P$	$=$	$(\lambda\ +\ \mu);\ P$

TAB. 2.4 – *Axiomes de la bisimulation faible en IMC*

Les axiomes A2 et A3 sont connues sous le nom de la bisimulation faible (*weak bisimulation*) [70] dans les algèbres de processus standards comme CCS. Les actiomes A4 et A5 décrivent l'application de la notion du progrès maximal. Notons que A5 est une extension *stochastique* de A3 par un taux markovien, quant à l'axiome A7, elle décrit l'application de la notion de *race condition*.

Notons qu'une étude stochastique d'une IMC et / ou sa transformation en CTMC impose :

- l'absence de toutes interactions avec l'environnement, par conséquent toutes les actions interactives doivent être transformées en action interne i par l'application de l'opérateur **hide**.
- l'absence de choix non-déterministe : $i;\ P\ [\ ]\ i;\ Q$.

2.2.3 Les IMC en LOTOS

Il est possible de spécifier les aspects temporels directement au niveau des IMC, mais ceci n'est pas toujours licite à l'égard de la complexité du système qui nécessite souvent une description par un langage haut niveau.

Les premières extensions stochastiques de LOTOS ont été proposées par Rico et Von Bochmann [81] par l'utilisation des semi-chaînes de Markov. Ainsi que la dérivation des réseaux de files d'attente à partir de spécifications LOTOS dans les travaux de Valderrutten [91]. Une approche similaire a été proposée par Schot [85] et Ajmone Marsan [64], elle consiste en une extension stochastique permettant des distributions plus générales. Cependant, la spécification des délais stochastiques s'effectue à un très haut niveau, réduisant ainsi la compositionnalité de manière significative.

La théorie des IMC proposée par Hermanns [41] est basée sur l'algèbre de processus du langage LOTOS.

L'extension de LOTOS par des aspects stochastiques pour l'obtention de IMC n'engendre aucune modification de la syntaxe du langage. Le principe de l'extension est le suivant [23] :

1. On part d'une spécification LOTOS dont les aspects fonctionnels ont été vérifiés.
2. Les taux markoviens λ_j sont insérés dans les *endroits appropriés* dans la spécification LOTOS au moyen de nouvelles actions interactives par :
   - la déclaration d'une porte pour chaque taux DELAY$_j$,
   - ou bien, par la déclaration d'une seule porte pour tous les taux et la distinction s'effectue au niveau des offres "DELAY ! j".
3. Les actions stochastiques DELAY$_j$ ou DELAY ! j vont être transformées en taux markoviens de la forme "*rate* λ_j" par l'opérateur "**rename** appliqué sur le système de transition généré à partir de la spécification LOTOS.

Remarque 2.5 *Les endroits appropriés dans la spécification LOTOS pour l'insertion des taux markoviens dépendent des aspects temporels du système. Par exemple dans notre cas, un taux markovien représente l'inverse de la latence d'un accès en mémoire ou en cache, ou l'inverse d'un délai d'attente (plus de détails dans le chapitre 5).*

Remarque 2.6 *Par souci de clarté, on appelle les actions interactives modélisant les taux markoviens dans un STE (avant le renommage) les* ***actions stochastiques****.*

Il important de s'assurer que l'insertion des taux markoviens ne perturbe pas le comportement fonctionnel généré à partir de la spécification originelle :

- Par l'abstraction des taux markoviens (**hide**) on obtient un comportement équivalent à celui généré à partir de la spécification LOTOS originelle modulo une relation d'équivalence.
- Valider la spécification enrichie par les taux markoviens en utilisant la même procédure de vérification du comportement fonctionnel appliquée sur la spécification d'origine.

Dans une spécification LOTOS augmentée de taux markoviens, toutes les actions sont interactives (avec une présence éventuelle d'actions internes) , par conséquent, l'application de tous les opérateurs LOTOS (composition parallèle, choix, ...) est possible. Dans le but de préserver l'aspect stochastique des actions décrivant des taux markoviens (actions stochastiques), leur traitement doit suivre deux règles importantes, la première 2.1 concerne la composition parallèle et la seconde 2.2 concerne le choix entre les actions stochastiques identiques.

Règle 2.1 *La composition parallèle avec synchronisation ($||$ ou $|[A]|$, $A \neq \emptyset$) est* ***interdite*** *pour les actions stochastiques .*

Règle 2.2 *En présence de plusieurs actions stochastiques identiques dans le STE généré à partir d'une spécification LOTOS, il faut :*

- *soit appliquer l'optimisation* ***stochastique*** *(bisimulation stochastique faible ou forte)* ***après*** *la transformation du STE en IMC.*
- *soit distinguer les actions stochastiques au niveau STE* ***avant*** *l'optimisation (bisimulation forte ou faible).*

L'exemple 2.8 ci-dessous est une clarification de la règle 2.2.

Exemple 2.8 *Le comportement E3 de la figure 2.4 décrit un choix entre deux actions stochastiques identiques* `DELAY !j`. *On distingue deux cas possibles de la transformation en IMC :*

Cas 1 : optimisation de E3 (par une relation de bisimulation forte), le comportement résultant E4 est par la suite transformé en IMC (par un renommage), ainsi on obtient E5.

Cas 2 : transformation de E3 en IMC (par un renommage), le comportement résultant E6 est optimisé par l'application de l'axiome A7 du tableau 2.4, ainsi on obtient E7.

Les procédures de transformation 1 et 2 produisent deux comportements stochastiques E5 et E7 complètement différents, alors qu'ils sont issus du même origine E3. (E7 représente la CTMC appropriée de E3)

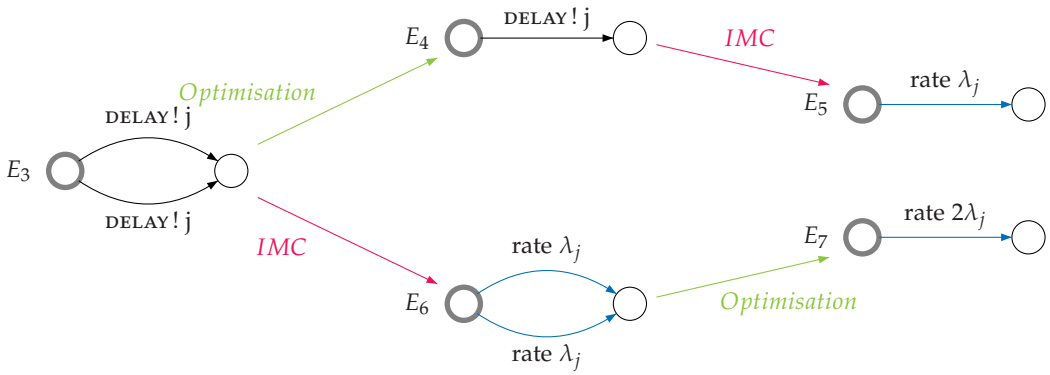


FIG. 2.4 – *Exemple de transformation d'un STE en IMC*

2.3 La boîte à outils CADP

CADP[2](*Construction and Analysis of Distributed Processes*) [25] est une boîte à outils pour l'ingénierie des protocoles de communication et des systèmes distribués. En plus des spécifications LOTOS, elle accepte également des descriptions de plus bas niveau, tels que les réseaux d'automates communicants. CADP offre un éventail de fonctionnalités assistant l'utilisateur tout au long du processus de conception : compilation et prototypage rapide, simulation interactive et guidée par des objectifs, exécution aléatoire, vérification par logiques temporelles et par équivalences, génération automatique de tests et évaluation de performances. Plusieurs techniques sont mises en œvre pour traiter le problème d'explosion d'espace d'états : vérification à la volée, construction compositionnelle de l'espace d'états, vérification massivement parallèle.

Nous présentons dans ce qui suit les principaux outils de CADP que nous avons utilisés dans nos études de cas.

[2]http://www.inrialpes.fr/vasy/cadp

CAESAR.ADT [21] est un compilateur qui traduit les types abstraits LOTOS en langage C. Il traite les définitions des types abstraits figurant dans un programme LOTOS (les définitions de processus étant ignorées), et produit une bibliothèque en langage C contenant, pour chaque sorte un type C qui l'implémente, et pour chaque opération une fonction. Cette bibliothèque est destinée à être incluse dans le programme principal, qui effectue la vérification.

La traduction des définitions des types abstraits en une bibliothèque en langage C, nous a facilité la définition de types complexes en C au sein de cette bibliothèque, par souci d'efficacité et dans le but d'éviter la lourdeur de leur définition en types abstraits. On les distingue dans la spécification LOTOS par le mot clé *external* pour évoquer leur implémentation en externe.

Nous traitons en détail la définition des types (abstraits et externes) dans le chapitre 3.

CAESAR [20] est un outil de compilation et de vérification des programmes LOTOS. Il prend en entrée un programme LOTOS ainsi qu'une implémentation C de ses types. En sortie il produit un STE (*Système de Transitions Etiquetées*) sous format BCG (*Binary Coded Graphs*) [22]. Le format BCG utilise des techniques efficaces de compression permettant de stocker des graphes (représentés sous forme explicite) sur disque de manière très compacte. En outre, il contient suffisamment d'informations pour que les outils qui l'exploitent puissent fournir à l'utilisateur des diagnostics précis dans les termes du programme source.

BCG_MIN [3] [66] est un outil de réduction de graphes BCG modulo la relation de bisimulation forte [53, 50] ou de branchement [32, 40]. Il peut être appliqué à différentes sortes de STE :

- Systèmes de transitions *ordinaires*, tels que ceux produits à partir des descriptions LOTOS.
- Systèmes de transitions *probabilistes* dont chaque transition peut être étiquetée soit par une action a ordinaire, soit par une probabilité p, soit par un couple (a, p).
- Systèmes de transitions *stochastiques* dont chaque transition peut être étiquetée, soit par une action a ordinaire, soit par un paramètre réel $\lambda > 0$, soit par un couple (a, λ).

EVALUATOR [4] est un évaluateur (*model checker*) permettant la vérification à la volée de formules de *mu*-calcul régulier sans alternance sur des STEs. Il met en œuvre des algorithmes de vérification à la volée basés sur les systèmes d'équations booléennes et fournit des mécanismes pour créer des bibliothèques de macro opérateurs. Il génère également des diagnostics (exemples ou contre-exemples) sous forme de fragments du STE illustrant la valeur de vérité des formules de logique temporelle.

[3] http://www.inrialpes.fr/vasy/cadp/man/bcg_min.html
[4] http://www.inrialpes.fr/vasy/cadp/man/evaluator.html

BCG_STEADY [5] [43] Effectue l'analyse des états récurrents de chaînes de Markov à temps continu encodées en format BCG. Il calcule l'état d'équilibre de la distribution de probabilité sur le long terme en utilisant l'algorithme Gauss/Seidel [89]. Il permet également de fournir des mesures de débit (*throughput*) des étiquettes des transitions du graphe.

Notre analyse numérique des performances est basée sur l'utilisation de cet outil.

SVL [6] (*Script Verification Language*) [59, 24] est un langage destiné à simplifier et à automatiser la mise en œuvre de la vérification compositionnelle et à la volée.

Le compilateur pour SVL produit, à partir d'un programme SVL, un *shell script* UNIX contenant la liste des commande à exécuter ainsi que les différents fichiers auxiliaires. Ce compilateur s'appuie sur divers formats et outils pour représenter et manipuler des STE.

[5] http://www.inrialpes.fr/vasy/cadp/man/bcg_steady.html
[6] http://www.inrialpes.fr/vasy/cadp/man/svl.html

3 Méthodologie de modélisation

Sommaire

Ce chapitre présente notre premier pas vers l'évaluation des performances : la modélisation du système. Nous avons proposé à l'occasion une méthodologie de modélisation en LOTOS des aspects matériels et logiciels des systèmes multiprocesseurs.

Nous discutons dans la première section la description formelle en LOTOS des composants de l'environnement matériel qui ont un impact primordial sur les performances : les *processeurs* sur lesquels s'exécutent les primitives MPI, la *mémoire* où les données des algorithmes sont sauvegardées, les *caches* et le *protocole de cohérence de caches* à partir duquel les types de transfert sont évalués et la *topologie du système* qui définit les niveaux des transferts.

Dans la seconde section nous présentons la traduction des structures de données et de contrôle des primitives MPI en LOTOS. Sachant que le critère de fiabilité dans cette traduction ne se limite pas seulement à la reproduction exacte du comportement de l'algorithme mais aussi bien dans le nombre d'accès et de mises à jour des données qui sont d'une très grande importance dans l'évaluation des performances .

Nous avons consacré la dernière section de ce chapitre à la présentation de l'application de notre méthodologie de modélisation sur trois algorithmes MPI : le *ping-pong*, la barrière *tournament* et la barrière *combining tree*.

3.1 Modélisation de l'environnement matériel

Nous avons modélisé l'architecture matérielle en faisant abstraction de certains détails liés à l'implémentation physique, afin d'obtenir des spécifications à la fois efficaces et suffisantes pour la nature du problème que nous traitons. Nous considérons que :

- Les processeurs sont référencés par un identificateur qu'on génère à partir du nombre de processeurs existants dans le système.
- Les processeurs et les caches sont deux entités indépendantes.
- La mémoire est d'une structure centralisée.
- Tous les caches du système sont regroupés dans une seule structure centralisée.
- Les caches ne comportent que les états des caches sans les données.

3.1.1 Les processeurs

Nous avons modélisé les identificateurs des processeurs par le type **ID_Processor** implémenté au moyen des entiers naturels en externe sous le nom ADT_ID_PROCESSOR. L'implémentation du type **ID_Processor** en externe nous a permis d'éviter l'énumération et l'insertion manuelle des identificateurs des processeurs.

```
type ID_Processor is Natural
 sorts ID_Processor  (*! implementedby ADT_ID_PROCESSOR
                        printedby ADT_PRINT_ID_PROCESSOR external *)
 opns
   Nil_Proc  (*! implementedby ADT_NIL_PROC external *) : -> ID_Processor
   Proc  (*! implementedby ADT_PROC external *) : Nat -> ID_Processor
   Previous_Proc  (*! implementedby ADT_PREVIOUS_PROC external *),
   Next_Proc  (*! implementedby ADT_NEXT_PROC external *)
                : ID_Processor -> ID_Processor
   _==_ , _<>_  : ID_Processor, ID_Processor -> Bool
 eqns
   forall A, B : ID_Processor
   ofsort Bool
     A == A = true ;
     A == B = false ;
     A <> B = not (A == B) ;
endtype
```

La constante *Nil_Proc* représente le nombre de processeurs utilisés pour l'exécution de l'algorithme MPI, elle sert à la comparaison (avec les opérateurs de comparaison $==$ et $<>$) et la vérification des identificateurs de processeurs. L'appel aux processeurs dans la spécification LOTOS s'effectue par l'opération *Proc* qui permet la conversion d'un entier naturel en un **ID_Processor**. Par exemple, *Proc(0)* désigne le processeur d'identificateur 0 (P_0) . L'opération *Previous_Proc* (resp. *Next_Proc*) permet une utilisation implicite des identificateurs de processeurs. Elle prend en entrée l'identificateur du processeur P_i et retourne en sortie l'identificateur du processeur P_{i-1} (resp. P_{i+1}) ou *Nil_Proc* si $i = 0$ (resp. si $i = Nil\_Proc - 1$).

La sorte **ID_Processor** et ses opérations sont implémentées en C, comme suit :

```
1.  #define ADT_NB_PROC  2
2.  #define ADT_NIL_PROC()  ADT_NB_PROC
3.  typedef unsigned char  ADT_ID_PROCESSOR;
4.  #define CAESAR_ADT_ITR_FIRST_ID_PROCESSOR()  0
5.  #define CAESAR_ADT_ITR_NEXT_ID_PROCESSOR(x)  (x++ < ADT_NB_PROC)

6.  void ADT_PRINT_ID_PROCESSOR(ADT_F, ADT_P)
7.  FILE *ADT_F;
8.  ADT_ID_PROCESSOR ADT_P;
9.  {
10.   if (ADT_P < ADT_NB_PROC)  fprintf (ADT_F,"P%D",ADT_A);
11.   else  fprintf (ADT_F,"NIL_PROC",ADT_A);
12. }

13. int  CAESAR_ADT_CMP_ID_PROCESSOR(ADT_A1,ADT_A2)
14. ADT_ID_PROCESSOR  ADT_A1,ADT_A2;
15. {
16.   if (ADT_A1 == ADT_A2)  return (ADT_TRUE());
17.   else  return (ADT_FALSE());
18. }

19. ADT_ID_PROCESSOR  ADT_PROC(ADT_N)
20. int ADT_N;
21. return ADT_N;

22. ADT_ID_PROCESSOR  ADT_PREVIOUS_PROC(ADT_P)
23. ADT_ID_PROCESSOR  ADT_P;
24. {
25.   if (ADT_P == 0)  return (ADT_NIL_PROC());
26.   else  return (ADT_P - 1);
27. }

28. ADT_ID_PROCESSOR  ADT_NEXT_PROC(ADT_P)
29. ADT_ID_PROCESSOR  ADT_P;
30. {
31.   if  (ADT_P == (ADT_NIL_PROC() - 1))  return (ADT_NIL_PROC());
32.   else  return (ADT_P + 1);
33. }
```

Dans la *ligne 1* on déclare le nombre de processeurs participant à l'exécution de l'algorithme MPI (dans cet exemple il s'agit de 2 processeurs), ce qui permet de définir par la suite le domaine de la sorte **ID_Processor** à l'aide de l'itérateur défini par les instructions des *lignes 4-5* en précisant qu'un identificateur d'un processeur est toujours supérieur ou égal à 0 et strictement inférieur à ADR_NB_PROC. Les instructions des *lignes 6-12* et des *lignes 13-18* correspondent à l'implémentation de la fonction d'impression et de comparaison respectivement, quant aux *lignes 19-21*, *lignes*

22-27 et les *lignes 28-33*, elles décrivent l'implémentation des opérations *Proc*, *Previous_Proc* et *Next_Proc* respectivement.

Remarque 3.1 *Nous définissons l'exécution d'une primitive MPI sur un processeur par un* processus *qu'on identifie par le nom de la primitive et l'identificateur du processeur sur lequel il s'exécute. On suppose que sur un processeur s'exécute un seul processus à la fois. Par exemple,* SEND_PROC[ACTION](PROC(0)) *indique l'exécution de la primitive send modélisée en LOTOS par le processus* SEND_PROC[ACTION] *sur le processeur* P_0.

3.1.2 La mémoire

Nous avons modélisé la mémoire par le type abstrait **Memory**. Ce dernier est implémenté au moyen d'un tableau de structures en C sous le nom ADT_MEMORY.

Chaque élément du tableau mémoire correspond à une variable précise de l'algorithme MPI. Un élément dans le tableau mémoire est une structure de données à deux champs. Le premier champ représente une modélisation de la nature distribuée de la mémoire dans l'architecture matérielle, il comporte l'identificateur du processeur qui fait référence à la mémoire locale de ce dernier (le processeur). Le second champ comporte le contenu de la variable. Par exemple, soit une variable v décrite dans le tableau mémoire par le couple $(0,3)$, ceci signifie que v est déclarée dans la mémoire locale du processeur P_0 et que son contenu est égal à 3. Nous supposons que les variables partagées sont déclarées dans la mémoire locale du processeur P_0 par défaut.

Dans la construction suivante on déclare en LOTOS le type **Memory** obtenu par la combinaison des types **Address** et **Memory_Value** que nous décrivons par la suite, et du type **ID_Action** qui indique la nature de l'accès : *Load*, *Store*, *Test_and_set* ou *Fetch_and_decrement* (voir la déclaration en LOTOS dans la section 2.1.1).

```
type Memory is Address, Memory_Value, ID_Action
 sorts Memory   (*! implementedby ADT_MEMORY external *)
 opns
   Init_Memory (*! implementedby ADT_INIT_MEMORY constructor external *)
                : -> Memory
   UpDate_Memory (*! implementedby ADT_UPDATE_MEMORY external *)
                 : Memory, Address, ID_Action, Memory_Value -> Memory
   Load_Memory (*! implementedby ADT_LOAD_MEMORY external *)
                : Memory, Address -> Memory_Value
endtype
```

Nous avons associé au type **Memory** les fonctions *Init_Memory*, *UpDate_Memory*, *Load_Memory* qui permettent l'initialisation de la mémoire (selon la fonction d'initialisation décrite dans l'algorithme MPI), la mise à jour et la lecture du contenu des variables respectivement.

Le fragment de code C suivant décrit l'implémentation du tableau mémoire (*lignes 1-7*) ainsi que la fonction d'initialisation (*lignes 8-29*) de la primitive barrière *centralized*.

```
1. typedef struct adt_struct_packet {
2. unsigned char ADT_ID_PROC;
3. unsigned char ADT_VAL;
4. } ADT_PACKET;

5. struct adt_struct_memory {
6. ADT_PACKET  ADT_TAB[ADT_SIZE_MEMORY];
7. } ADT_MEMORY;

8.  ADT_MEMORY ADT_INIT_MEMORY()
9.  {
10.   ADT_MEMORY  ADT_M;
11.   ADT_ID_PROCESSOR  ADT_P;
12.   ADT_ADDRESS  ADT_A = 0;

13.   memset ((void *) (&ADT_M), 0, sizeof (ADT_MEMORY));

14.   /* initialisation de  COUNT  */
15.   ADT_MEM.ADT_TAB[ADR_A].ADT_ID_PROC = 0;
16.   ADT_MEM.ADT_TAB[ADR_A].ADT_VAL = ADT_COUNT_VALUE();
17.   ADT_A++;

18.   /* initialisation de  SENSE  */
19.   ADT_MEM.ADT_TAB[ADR_A].ADT_ID_PROC = 0;
20.   ADT_MEM.ADT_TAB[ADR_A].ADT_VAL = ADT_TRUE_VAL();
21.   ADT_A++;

22.   /* initialisation de  LOCAL_SENSE[pi]  */
23.   for (ADT_P = 0; ADT_P < ADT_NB_PROC; ADT_P++) {
24.     ADT_MEM.ADT_TAB[ADR_A].ADT_ID_PROC = ADT_P;
25.     ADT_MEM.ADT_TAB[ADR_A].ADT_VAL = ADT_TRUE_VAL();
26.     ADT_A++;
27.   }
28.   return (ADT_M);
29. }
```

3.1.3 Les caches et le protocole de cohérence de caches

Afin de simplifier la modélisation du protocole de cohérence de caches et d'éviter la vérification de la cohérence des données en caches et en mémoire dans le modèle, nous avons considéré uniquement l'état des données en caches (sans les données elles-mêmes). Cette abstraction est jugée suffisante pour l'étude de notre système, étant donné que l'objectif primordial de notre travail est focalisé sur l'évaluation des performances et dans une moindre mesure sur la vérification de la correction du fonctionnement du protocole de cohérence de caches.

Nous avons modélisé les caches de tous les processeurs du systèmes par le type abstrait **Cache** auquel nous avons associé l'opération *Up-Date_Cache* qui spécifie la mise à jour des caches suivant le protocole de cohérence adapté.

La sorte **Cache** est implémentée au moyen d'une matrice en C sous le nom ADT_CACHE qui spécifie les états des variables en caches. Chaque ligne de la matrice cache correspond à une variable de l'algorithme MPI.

Notons qu'une variable possède un indice unique qui sert à la fois dans le tableau mémoire et la matrice cache comme il est indiqué dans la figure 3.1. Les indices des colonnes de la matrice cache correspondent aux identificateurs des processeurs, par exemple la colonne 0 représente le cache du processeur P_0.

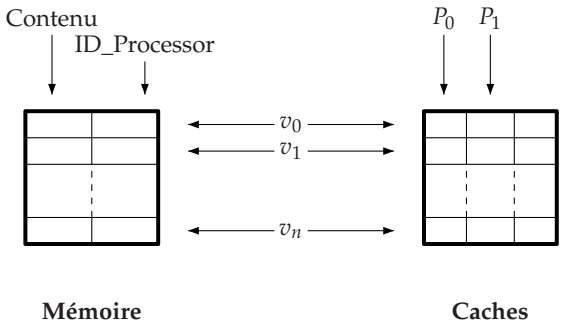


FIG. 3.1 – *Structure de la mémoire et des caches*

Dans la construction suivante on déclare en LOTOS le type **Cache** obtenu par la combinaison des types **Address** et **ID_Action**. L'opération *Init_Cache* permet l'initialisation des caches par l'invalidation de tous ces éléments (toutes les variables dans tous les processeurs sont à l'état **I**).

```
type Cache is Address, ID_Action, ID_Processor
  sorts Cache   (*! implementedby ADT_CACHE external *)
  opns
    Init_Cache   (*! implementedby ADT_INIT_CACHE constructor external *)
                    : -> Cache
    UpDate_Cache   (*! implementedby ADT_UPDATE_CACHE external *)
                       : Cache, ID_Action, Address, ID_Processor -> Cache
endtype
```

L'opération *UpDate_Cache* est implémentée en C par la fonction ADT_UPDATE_CACHE paramétrée par : les caches (en entrée avant mise à jour et en sortie après mise à jour), le type d'accès, l'indice de la variable dans la structure cache et l'identificateur du processus réalisant l'accès. La procédure de mise à jour de cache est basée sur la lecture des règles du changement des états définies dans le tableau 1.2 et 1.3 que nous avons implémenté au moyen de tableaux statiques en C.

Afin de faciliter l'étude comparative des performances du système pour différents protocoles de cohérence de caches, nous considérons le protocole de cohérence de caches comme étant un paramètre à fixer lors de la compilation, ainsi nous l'avons codé dans un fichier ".*t*". Par exemple le fichier du protocole de cohérence de caches *A* défini dans 1.4 est nommé *PCC_A.t*. La construction ci-dessous illustre une partie du contenu du fichier *PCC_A.t*. Chaque ligne dans le tableau ADT_LOAD_PROTOCOL correspond à une règle de changement d'états. Les *colonnes 0* et *1* indiquent

l'état courant du cache du processeur demandeur, et des autres processeurs dans le système respectivement. La *colonne 2* indique que tous les caches (ADT_ALL) ou au moins un cache (ADT_EXIT) est dans l'état mentionné dans la *colonne 1*. Les *colonnes 3* et *4* désignent les nouveaux états de caches du processeur demandeur et d'autres processeurs dans le système respectivement et la *colonne 5* indique le type de transfert engendré par l'accès.

```
#define ADT_ALL 0
#define ADT_EXIT 1
#define ADT_NB_LINE_LOAD_PROTOCOL 7

#define ADT_CURRENT_STATE_REQ_INDEX 0          /* column 0 */
#define ADT_CURRENT_STATE_OTHER_INDEX 1        /* column 1 */
#define ADT_NB_OTHER_INDEX 2                   /* column 2 */
#define ADT_NEXT_STATE_REQ_INDEX 3             /* column 3 */
#define ADT_NEXT_STATE_OTHER_INDEX 4           /* column 4 */
#define ADT_TRANSFER_TYPE_INDEX 5              /* column 5 */

unsigned char ADT_LOAD_PROTOCOL[ADT_NB_LINE_LOAD_PROTOCOL][6] = {
/*RL1*/ { ADT_I, ADT_I, ADT_ALL, ADT_E, ADT_I, ADT_MEMORY_TRANSFER},
/*RL2*/ { ADT_I, ADT_S, ADT_EXIST, ADT_S, ADT_S, ADT_MEMORY_TRANSFER},
/*RL3*/ { ADT_I, ADT_E, ADT_EXIST, ADT_S, ADT_S, ADT_MEMORY_INV_TRANSFER},
/*RL4*/ { ADT_I, ADT_M, ADT_EXIST, ADT_E, ADT_I, ADT_CACHE_TRANSFER},
/*RL6*/ { ADT_S, ADT_S, ADT_EXIST, ADT_S, ADT_I, ADT_INTERNAL},
/*RL6*/ { ADT_E, ADT_I, ADT_ALL, ADT_E, ADT_I, ADT_INTERNAL},
/*RL6*/ { ADT_M, ADT_I, ADT_ALL, ADT_M, ADT_I, ADT_INTERNAL}};
```

3.1.4 La topologie du système

Nous définissons la topologie du système comme étant une relation de distance entre les processeurs, comme illustré dans la figure 3.2.

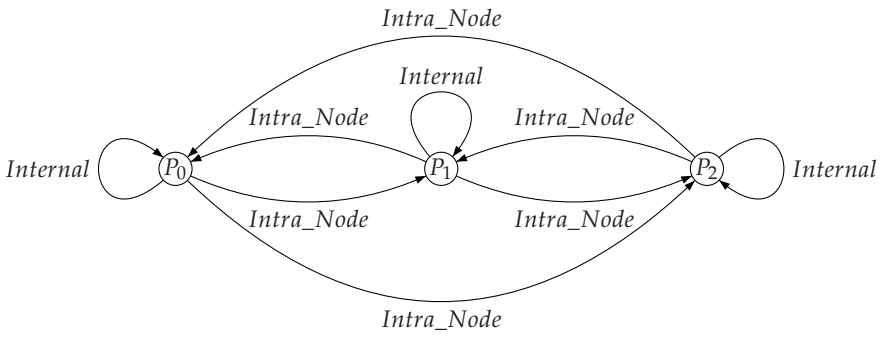


FIG. 3.2 – *Topologie du système : relation de distance entre les processeurs*

La topologie du système est utilisée implicitement dans la spécification LOTOS lors de l'évaluation des latences d'accès, par conséquent, nous l'avons implémentée directement au moyen d'une matrice en C sans la modéliser en un type abstrait en LOTOS.

Nous avons codé en C chaque topologie dans un fichier ".*t*" (on indique généralement dans le nom du fichier le nombre de processeurs et l'identificateur de la topologie). Par exemple le fichier topologie de la figure 3.2 nommé par *topology_nb*3_0.*t* correspond à la $topology_0$ ou tous les processeurs se trouvent dans le même nœud.

```
1. #define ADT_ND_PROC  3
2. #define ADT_INTERNAL  0
3. #define ADT_INTRA_NODE  1
4. #define ADT_INTER_NODE  2
5. #define ADT_INTER_MODULE  3
6. ADT_TOPOLOGY_SYSTEM[ADT_NB_PROC][ADT_NB_PROC]={
7. /*          P0              P1              P2    */
8. /*P0*/ {ADT_INTERNAL,    ADT_INTRA_NODE,  ADT_INTRA_NODE },
9. /*P1*/ {ADT_INTRA_NODE,  ADT_INTERNAL,    ADT_INTRA_NODE},
10. /*P2*/ {ADT_INTRA_NODE,  ADT_INTRA_NODE,  ADT_INTERNAL }};
```

Dans la *ligne 1* on déclare le nombre de processeurs et dans les *lignes 2-5* on déclare les différentes distances possibles entre les processeurs.

L'instanciation de la topologie se fait au niveau de la compilation de la spécification LOTOS en précisant le nombre de processeurs ainsi que l'identificateur de la topologie. L'exemple 3.1 décrit la compilation de la spécification LOTOS de la barrière *centralized* entre 3 processus dans la $topology_0$ (*topology_nb*3_0.*t*).

Exemple 3.1

```
#> caesar.adt -cc "-DNB_PROC=3 -DTOPOLOGY=0" Centralized.lotos
#> caesar -cc "-DNB_PROC=3 -DTOPOLOGY=0" Centralized.lotos
```

A l'aide d'un fichier bibliothèque *topology.t* regroupant toutes les topologies que nous avons définies, et avec la précision du nombre de processeurs ainsi que l'identificateur de la topologie, l'instanciation de la topologie désirée se fait correctement lors de la compilation. Ci-dessous une partie du code C du fichier *topology.t* :

```
#if (ADT_NB_PROC == 3) && (ADT_TOPOLOGY == 0)
  #include "topology_nb3_0.t"
#elif (ADT_NB_PROC == 3) && (ADT_TOPOLOGY == 1)
  #include "topology_nb3_1.t"
...
#endif
```

3.1.5 Les latences d'accès

Nous avons modélisé les différentes latences possibles d'un accès par le type énuméré **Latency_Value** décrit ci-dessous :

```
type Latency_Value is Natural
  sorts Latency_Value
  opns
    C_INT   (*! constructor *),
    M_FSB_1   (*! constructor *),
    M_FSB_2   (*! constructor *),
    M_FSB_3   (*! constructor *),
    M_SP   (*! constructor *),
    M_XSP   (*! constructor *),
    C_FSB   (*! constructor *),
    C_SP   (*! constructor *),
    C_XSP   (*! constructor *),
    NIL_LATENCY   (*! constructor *) : -> Latency_Value
  eqns
    forall A, B : Latency_Value
    ofsort Bool
      A == A = true;
      A == B = false;
      A <> B = not (A == B);
endtype
```

Chaque constructeur spécifie la latence d'un accès qui dépend de la distance entre la source et la destination de la donnée : *Internal Intra_Node*, *Inter_Node*, *Inter_Module*, et le type de transfert : interne, mémoire, cache, mémoire avec consultation (et / ou invalidation) (voir tableau 1.6).

3.1.6 Gestion des accès

Nous avons spécifié la gestion de la mémoire et les caches par un processus en LOTOS que nous avons appelé TRANSFER. Ce dernier joue le rôle d'un serveur de données et d'un contrôleur de caches à la fois : il satisfait les demandes d'accès aux variables et assure la cohérence des caches par les mises à jour nécessaires après chaque accès.

Le processus TRANSFER possède un comportement réactif (fonctionnalité **noexit**). Il est paramétré par la porte de communication ACTION et une variable de type **Memory** et une autre de type **Cache**.

Une demande d'accès à une variable s'effectue par un rendez-vous sur la porte ACTION. Lors de la synchronisation sur ACTION, le processus TRANSFER reçoit toutes les informations nécessaires pour la mise à jour de la mémoire et des caches : l'identificateur du processeur *P*, l'indice de la variable en mémoire *adr*, l'opération d'accès *op* et la valeur *value* qui représente la nouvelle valeur de la variable *adr* dans le cas d'une écriture, ou son contenu dans le cas d'une lecture (et dans le cas des autres opérations d'accès elle sert pour la vérification) et renvoie sur le même rendez-vous la latence de l'accès *latency*.

Le processus TRANSFER se rappelle récursivement avec les nouvelles valeurs de la mémoire et les caches évaluées par les opérations *UpDate_Memory* et *UpDate_Cache* respectivement.

```
process   TRANSFER [ACTION] (M :Memory, C :Cache) : noexit :=
 ACTION   ? P :ID_Processor   ? adr :Address   ? op :ID_Action
          ? val :Memory_Value   ? latency :Latency_Value
   [
    (((op == Load) and (val == Load_Memory(M, adr))) or
     ((op == Fetch_and_decrement) and
      (val == Load_Memory(UpDate_Memory(M, adr, op, val), adr))) or
     (op == Store)
    ) and
    (latency == Evaluation_Latency(M, C, op, adr, p))
   ];
  TRANSFER [ACTION] (UpDate_Memory(M, adr, op, val),
                     UpDate_Cache(C, op, adr, P))
endproc
```

Remarque 3.2 *Pour assurer la cohérence de la mémoire et les caches, toutes les demandes d'accès ne doivent pas être traitées indépendamment les unes des autres, de ce fait le processus* TRANSFER *est un processus* ***central*** *dans la spécification LOTOS. Cependant, il est possible de définir un processus* TRANSFER *pour chaque variable, avec une mémoire représentant son contenu et des caches représentant son état dans tous les caches des processeurs, mais il n'est possible de définir un processus* TRANSFER *par processus P_i que pour des variables privées.*

3.2 Modélisation de l'environnement logiciel

3.2.1 Structures de données

En LOTOS l'échange et les mises à jour des données se font par le biais des rendez-vous sur les portes de synchronisation, en précisant le nom de la variable ainsi que son contenu sur des offres différentes. Cette distinction entre le nom et le contenu des variables sur les portes de synchronisation nous a mené à la définition de deux types abstraits en LOTOS : le type **Address** pour le traitement du nom des variables et le type **Memory_value** pour le traitement du contenu des variables.

• Les variables de l'algorithme MPI

Chaque algorithme MPI possède ses propres opérations dans le type **Address** qui correspondent aux noms de ses variables.

Dans la construction suivante on déclare le type **Address** de l'algorithme barrière *centralized*, avec *count*, *sense* et *local_sense* correspondant aux noms de ses variables.

```
type Address is ID_Processor
 sorts Address (*! implementedby ADT_ADDRESS
                   printedby ADT_PRINT_ADDRESS external *)
 opns
   count   (*! implementedby ADT_COUNT external *) : -> Address
   sense   (*! implementedby ADT_SENSE external *) : -> Address
   local_sense   (*! implementedby ADT_LOCAL_SENSE external *)
                 : ID_Processor -> Address
endtype
```

On distingue les variables partagées des variables privées par la précision de l'identificateur du processeur dans ces dernières. Par exemple, *local_sense(Proc(0))* correspond à la variable privée du processus P_0.

La partie du code C ci-dessous illustre l'implémentation en C du type **Address** de l'algorithme barrière *centralized* :

- *Ligne 1* : indique l'implémentation de ADT_ADDRESS en tant que caractère non signé (pour un codage optimisé sur un seul octet) qui fait référence aux indices des variables dans le tableau mémoire.
- *Lignes 2-4* : les constantes ADT_COUNT_RAW, ADT_SENSE_RAW et ADT_LOCAL_SENSE_RAW correspondent aux indices des variables *count*, *sense* et *local_sense* respectivement, dans le tableau mémoire.
- *Lignes 5-11* : les opérations LOTOS *count*, *sense* et *local_sense* sont implémentées en C par les fonctions ADT_COUNT, ADT_SENSE et ADT_LOCAL_SENSE respectivement. Elles permettent de retourner l'indice des variables dans le tableau mémoire. Dans le cas des variables privées, la précision de l'identificateur du processeur est indispensable. Par exemple, l'indice de la variable privée *local_sense* du processeur P_2 est égal à la somme de la constante ADT_LOCAL_SENSE_RAW et l'identificateur du processeur P_2, c'est à dire ADT_LOCAL_SENSE_RAW + 2.
- *Lignes 12-29* : ADR_PRINT_ADDRESS est une fonction d'impression, elle permet un affichage non ambigüe des variables sur les portes de synchronisation et facilite par conséquent l'interprétation des transitions. Par exemple, à partir du rendez-vous "**ACTION** ! local_sense(Proc(0)) ;" on peut générer deux transitions différentes :
  (a) "**ACTION** ! LOCAL_SENSE_P0" avec l'utilisation de la fonction d'impression ADR_PRINT_ADDRESS.
  (b) "**ACTION** !2" sans l'utilisation d'un fonction d'impression. Avec 2 est l'indice de la variable privée *local_sense* du processeur P_0 dans le tableau mémoire.

```
1. typedef unsigned char ADT_ADDRESS;
2. #define ADT_COUNT_RAW  0
3. #define  ADT_SENSE_RAW  ADT_COUNT_RAW + 1
4. #define ADT_LOCAL_SENSE_RAW  ADT_SENSE_RAW + 1

5. ADT_ADDRESS  ADT_COUNT() {
6.    return (ADT_ADT_COUNT_RAW); }

7. ADT_ADDRESS  ADT_SENSE() {
8.    return (ADT_SENSE_RAW); }

9.  ADT_ADDRESS  ADT_LOCAL_SENSE(ADT_ID_PROC)
10. ADT_ID_PROCESSOR  ADT_ID_PROC; {
11.    return (ADT_ID_PROC + ADT_LOCAL_SENSE_RAW); }

12. void ADT_PRINT_ADDRESS (ADT_F, ADT_A)
13. FILE *ADT_F;
14. ADT_ADDRESS  ADT_A;
15. {
16.    ADT_ID_PROCESSOR  ADT_P = 0;
17.    int exit = 0;
18.    while ((ADT_P < ADT_NB_PROC) && (exit == 0)) {
19.     if (ADT_A == ADT_LOCAL_SENSE(ADT_P)) {
20.      fprintf (ADT_F,"LOCAL_SENSE_P%d",ID_PROC);
21.      exit = 1;
22.      ADT_P = ADT_NB_PROC; }
23.     else ADT_P++;
24.    }
25.    if (exit == 0) {
26.     if (ADT_A == ADT_COUNT_RAW)  fprintf (ADT_F, "COUNT");
27.     else (ADT_A == ADT_SENSE_RAW)  fprintf (ADT_F, "SENSE");
28.    }
29. }
```

• Le contenu des variables de l'algorithme MPI

La plupart des algorithmes MPI que nous avons modélisés utilisent des variables de type booléen, entier naturel ou pointeur. Nous avons uniformisé le traitement de ces types en les regroupant dans un seul type en LOTOS, appelé **Memory_Value**.

Nous définissons principalement dans le type **Memory_Value** les constantes de l'algorithme MPI afin de simplifier leur utilisation au niveau de la spécification LOTOS. Par exemple dans la construction suivante on associe à **Memory_Value** l'opérateur *True_Val* (resp. *False_Val*) qui est une modélisation de la valeur de vérité *true* (resp. $false$). L'opérateur *Lock_Memory* (resp. *UnLock_Memory*) correspond à la constante de verrouillage (resp. déverrouillage) et *Nil_Memory* modélise le pointeur Nil.

Nous avons associé également l'opérateur *Memory_Val* permettant la conversion d'un entier en un **Memory_Value**.

Les opérateurs *Addition*, *Minus*, *Multiplication* et *Division* correspondent à la modélisation des fonctions usuelles d'addition, de soustraction, de multiplication et de division respectivement.

L'opérateur *Less* permet la comparaison entre deux valeurs de type **Memory_Value**. Si a et b deux quantités de type **Memory_Value** et si $a > b$, alors *Less(a,b)* retourne *False_Val* et *Less(b,a)* retourne *True_Val*.

```
type Memory_Value is Natural, ID_Processor, ID_Action, Boolean
 sorts Memory_Value  (*! implementedby ADT_MEMORY_VALUE
                        printedby ADT_PRINT_MEMORY_VALUE external *)
 opns
   True_Val  (*! implementedby ADT_TRUE_VAL external *),
   False_Val  (*! implementedby ADT_FALSE_VAL external *),
   Lock  (*! implementedby ADT_LOCK external *),
   UnLock  (*! implementedby ADT_UNLOCK external *),
   Nil_Memory  (*! implementedby ADT_NIL_MEMORY external *) : -> Memory_Value
   Memory_Val  (*! implementedby ADT_MEMORY_VAL external *) : Nat -> Memory_Value
   Addition  (*! implementedby ADT_ADDITION external *),
   Minus  (*! implementedby ADT_MINUS external *),
   Multiplication  (*! implementedby ADT_MULTIPLICATION external *),
   Division  (*! implementedby ADT_DIVISION external *)
   Less  (*! implementedby ADT_LESS external *)
                : Memory_Value, Memory_Value -> Memory_Value
   _==_ , _<>_ : Memory_Value, Memory_Value -> Bool
 eqns
   forall A, B : Memory_Value
   ofsort Bool
     A == A = true;
     A == B = false;
     A <> B = not (A == B);
endtype
```

Comme le type **Address**, **Memory_Value** est implémenté au moyen de caractère non signé en C par ADT_MEMORY_VALUE. En effet la considération des pointeurs comme des entiers naturels ne contredit pas les hypothèses de modélisation puisque'ils correspondent aux indices des tableaux mémoire et caches.

3.2.2 Structures de contrôle

• Affectation de variables

Une affectation se traduit par une (ou deux) demande(s) d'accès à une (ou deux) variable(s) pour une écriture (précédée d'une lecture). Nous avons modélisé l'instruction d'affectation par des rendez-vous sur la porte ACTION, comme décrit dans l'exemple suivant :

Exemple 3.2

Langage impératif	LOTOS
var1 = 7	**ACTION** ! var1 ! Store ! Memory_Val(7);

L'affectation d'une variable à une autre nécessite deux accès, le premier en lecture et le second en écriture, comme dans l'instruction d'affectation suivante :

Exemple 3.3

Langage impératif	LOTOS
var1 = var2	**ACTION** ! var2 ! Load ? val_var2 ; **ACTION** ! var1 ! Store ! val_var2 ;

Dans l'exemple 3.3, le premier rendez-vous a lieu pour charger le contenu de *var2* dans *val_var2* par un accès en lecture et dans le second rendez-vous le contenu de *var1* est positionné à la valeur *val_var2* suite à un accès en écriture.

• **La structure conditionnelle : "if then else"**

La clause "**if then else** " se traduit en LOTOS par les commandes gardées ([ ] →), ou par des rendez-vous conditionnés par une garde.

Soit la traduction en LOTOS d'une instruction conditionnelle décrite dans l'exemple suivant :

Exemple 3.4

Langage impératif	LOTOS
if var1 == var2 **then** var3 = true ; **else** var3 = false ;	**ACTION** ! var2 ! Load ? val_var2 ; **ACTION** ! var1 ! Load ? val_var1 ; ([val_var1 == val_var2] ->**ACTION** ! var3 ! Store ! True_val ; **exit** [] [val_var1 <> val_var2] ->**ACTION** ! var3 ! Store ! False_val ; **exit**)

D'abord, le contenu des variables *var1* et *var2* est chargé dans *val_var1* et *val_var2* respectivement. Si la condition de la première garde est satisfaite, c'est à dire *val_var1* == *val_var2*, alors le contenu de la variable *var3* est positionné à *True_Val* par un accès en écriture, sinon (la condition de deuxième garde est satisfaite) *var3* reçoit la valeur *False_val* par un accès en écriture également. Ici, les conditions des deux gardes sont mutuellement exclusives, on dit alors que le choix est *déterministe*, sinon on dit qu'il est *non-déterministe*.

Par ailleurs, il est possible de traduire l'instruction précédente par la modélisation des conditions par des gardes sur les rendez-vous, comme suit :

```
ACTION  ! var2  ! Load  ? val_var2;
(
 ACTION  ! var1  ! Load  ! val_var2;
 ACTION  ! var3  ! Store  ! True_val;
 exit
  [ ]
 ACTION  ! var1  ! Load  ! val_var1  [val_var1 <> val_var2];
 ACTION  ! var3  ! Store  ! False_val;
 exit
)
```

Le contenu de la variable *var2* est chargé dans *var_var2*, ensuite un choix déterministe est effectué sur le rendez-vous modélisant la lecture de *var1*. Pour que le rendez-vous du premier choix ait lieu, il faut que la valeur lue soit égale à *val_var2*, car elle est décrite sur la 3^{eme} offre par un envoi (*! val_var2*). Dans le cas contraire, c'est la synchronisation du deuxième choix qui a lieu après la vérification de la satisfaction de la condition de la garde *val_var1 <> val_var2*.

• Les boucles

Nous avons modélisé les boucles au moyen de processus LOTOS de fonctionnalité **exit**. Soit la boucle **while** suivante :

while (var > 0) { var = var - 1 ; }

La traduction en LOTOS de cette boucle est donnée par le processus WHILE_PROC définit ci-dessous :

```
process  WHILE_PROC [ACTION] : exit :=
 ACTION  ! var  ! Load  ? val_var [val_var > Memory_Val(0)];
 ACTION  ! var  ! Store  ! Minus(val_var, Memory_Val(1));
 WHILE_PROC [ACTION]
  []
 ACTION  ! var  ! Load  ! Memory_Val(0);
 exit
endproc
```

Le premier rendez-vous du premier choix a lieu si la condition de la garde est satisfaite : le contenu de la variable *var* est supérieur à 0, suivi d'un rendez-vous pour un accès en écriture pour décrémenter le contenu de *var* de 1 par l'application de l'opération *Minus* (*Minus(val_var, Memory_Val(1))*). Par la suite, le processus WHILE_PROC se rappelle récursivement pour effectuer l'itération suivante.

Le rendez-vous du second choix a lieu si le contenu de *var* est égal à 0, suivi de l'action **exit** qui met fin à l'exécution du processus WHILE_PROC.

• La récursivité

Une fonction est dite récursive si elle a la faculté de s'appeler elle-même. On distingue deux types de récursivité : *terminale* et *non-terminale*. Une

récursivité terminale offre la possibilité d'une transformation au mode itératif, contrairement à la récursivité non-terminal où la transformation nécessite la définition d'une pile pour la sauvegarde des contextes des appels.

La modélisation en LOTOS des fonctions récursives nécessite d'abord leur dé-récursion et par conséquent, l'implémentation de toute la procédure de la sauvegarde du contexte des appels dans le cas d'une récursivité non-terminale.

Nous présentons dans la section 3.3.3, la modélisation en LOTOS d'un cas d'un algorithme de MPI comportant une récursivité non-terminal, il s'agit de l'algorithme barrière *combining tree*.

3.3 Application

Nous présentons dans cette section l'application de notre méthode de modélisation -présentée ci-dessus- sur les primitives *send* et *receive* d'un algorithme de *ping-pong* et deux algorithme de barrière : *tournament* et *combining tree*, en mettant l'accent sur la particularité de chaque algorithme.

3.3.1 Algorithme *ping-pong*

L'algorithme de *ping-pong* consiste en des envois alternés de messages entre processus via les primitives d'envoi *send* et de réception *receive*. Algorithmiquement, le *ping-pong* est une composition parallèle de deux processus :

$$ping\text{-}pong(P_0, P_1) = ping(P_0, P_1) \;|||\; pong(P_1, P_0)$$

Avec :

$$ping(P_0, P_1) = send(P_0, P_1) \;>>\; receive(P_0, P_1)$$
$$pong(P_1, P_0) = receive(P_1, P_0) \;>>\; send(P_1, P_0)$$

Le processus *ping* (resp. *pong*) représente une exécution récursive de la composition séquentielle *send* et *receive* (resp. *receive* et *send*). On suppose que son traitement est réalisé sur le processeur P_0 (resp. P_1).

Nous avons modélisé l'algorithme *ping-pong* basé sur l'utilisation des primitives *send* et *receive* de l'implémentation MPICH[1] de la version 1.2.6 pour le device ch_shmem, dont la description en langage algorithmique des primitives *send* et *receive* est donnée ci-dessous :

Algorithme 2 : La primitive *send*

```
   Data : Pi, Pj process identifer
 1 begin
 2     if Local_Available_Ptr[i] != Null then
 3         Pkt = Local_Available_Ptr[i];
 4     else
 5         if Free_Head_Ptr[i, j] != Null then
 6             Lock (Available_Lock[j]);
 7             Free_Tail_Ptr[i, j] → Next = Available_Ptr[j];
 8             Available_Ptr[j] = Free_Head_Ptr[i, j];
 9             Unlock (Available_Lock[j]);
10             Free_Head_Ptr[i, j] = Null;
11             Free_Tail_Ptr[i, j] = Null;
12         end
13         while true do
14             if Available_Ptr[i] != Null then
15                 Lock (Available_Lock[i]);
16                 Pkt = Available_Ptr[i];
17                 Available_Ptr[i] = Null;
18                 Unlock (Available_Lock[i]);
19                 Break;
20             end
21         end
22     end
23     Local_Available_Ptr[i] = Pkt_Next;
24     Pkt → src = i;
25     Pkt → Next = Null;
26     Lock (Incoming_Lock[j]);
27     if Incoming_Head_Ptr[j] == Null then
28         Incoming_Head_Ptr[j] = Pkt;
29         Incoming_Tail_Ptr[j] = Pkt;
30     else
31         Incoming_Tail_Ptr[j] → Next = Pkt;
32         Incoming_Tail_Ptr[j] = Pkt;
33     end
34     Unlock (Incoming_Lock[j]);
35 end
```

[1]http ://www-unix.mcs.anl.gov/mpi/mpich1/

Algorithme 3 : La primitive *receive*

```
    Data : Pi, Pj process identifer
 1  begin
 2      if Incoming_Head_Ptr[i] == Null then
 3          while Incoming_Head_Ptr[i] == Null do
 4              Wait (1);
 5              if Incoming_Head_Ptr[i] != Null then
 6                  Break;
 7              end
 8              if Free_Head_Ptr[i, j] != Null then
 9                  Lock (Available_Lock[j]);
10                  Free_Tail_Ptr[i, j] → Next= Available_Ptr[j];
11                  Available_Ptr[j]= Free_Head_Ptr[i, j];
12                  Unlock (Available_Lock[j]);
13                  Free_Head_Ptr[i, j] = Null;
14                  Free_Tail_Ptr[i, j] = Null;
15              end
16              Wait (2);
17          end
18      end
19      Lock (Incoming_Lock[i]);
20      Pkt = Incoming_Head_Ptr[i];
21      Incoming_Head_Ptr[i] = Null;
22      Incoming_Tail_Ptr[i] = Null;
23      Unlock (Incoming_Lock[i]);
24      Pkt → src = j;
25      Pkt → Next = Free_Head_Ptr[i, j];
26      Free_Head_Ptr[i, j] = Pkt;
27      if Free_Tail_Ptr[i, j] == Null then
28          Free_Tail_Ptr[i, j] = Free_Head_Ptr[i, j];
29      end
30  end
```

L'instruction *Wait()* (lignes 4 et 16 dans la primitive *receive*) correspond à un traitement autre que les accès aux données en mémoire ou en caches, il peut s'agir par exemple de relâcher le processeur. Nous avons modélisé en LOTOS cette instruction par un rendez-vous sur une porte de communication que nous avons nommé WAIT.

Exemple 3.5 *Le rendez-vous*

WAIT ! P **of** ID_Processor ! 1 **of** Nat ;

est une modélisation de l'instruction Wait(1).

Les messages échangés entre les processus sont constitués de paquets de données, ils comportent des informations sur l'émetteur et le récepteur ainsi que le type et la taille des données. Par souci de simplicité, nous considérons que les messages ne contiennent qu'un seul paquet pour indiquer l'identificateur de l'émetteur. De ce fait, nous utiliserons par la suite le mot *paquet* pour désigner un message. Nous utiliserons également les identificateurs des processeurs pour dénoter les processus exécutés par ces derniers.

Chaque processus possède un espace d'adressage privé protégé vis-à-vis des accès de l'autre processus. Les messages échangés sont définis dans des listes de paquets pour chaque processus. On distingue trois types de listes selon l'état de l'émission ou de la réception du paquet :

- *Disponible* : représente la liste des paquets disponibles prêts à l'émission.
- *Reçu* : représente la liste des paquets reçus prêts à la consommation.
- *Consommé* : représente la liste des paquets consommés prêts a être rendus à l'émetteur.

Comme il s'agit de liste de paquets, on déduit les types de données suivants :

- *Paquet* : il s'agit d'un descripteur qui contient l'identité de l'émetteur, et un pointeur vers le paquet suivant.
- *Pointeur* : chaque liste de paquet possède un pointeur sur sa tête, et éventuellement sur sa fin.
- *Verrou* : pour accéder aux pointeurs des listes définies dans l'espace d'adressage partagé.

Dans la description des primitives *send* et *receive*, la notation $nom\_liste[i]$ indique une liste du processus P_i, et la notation $nom\_liste[i, j]$ indique une liste du processus P_i pour les paquets du processus P_j.

Le tableau 3.1 résume les différentes variables utilisées dans l'algorithme *ping-pong* :

Type liste	Nom liste	Nom variable	Type variable	Espace
Disponible	Local_Available_list	Local_Available_Ptr	pointeur	privé
	Available_list	Available_Ptr	pointeur	partagé
		Available_Lock	verrou	partagé
Reçu	Incoming_List	Incoming_Head_Ptr	pointeur	partagé
		Incoming_Tail_Ptr	pointeur	partagé
		Incoming_Lock	verrou	partagé
Consommé	Free_List	Free_Head_Ptr	pointeur	privé
		Free_Tail_Ptr	pointeur	privé

TAB. 3.1 – *Structures de données de l'algorithme* ping-pong

• Modélisation des structures de données

En suivant la méthodologie de modélisation que nous avons présentée précédemment, le type **Address** pour l'algorithme *ping-pong* définit toutes les variables utilisées par ce dernier en tant qu'opérations référençant leurs indice dans le tableau mémoire.

```
type Address is ID_Processor, Memory_Value
 sorts Address (*! implementedby ADT_ADDRESS
                   printedby ADT_PRINT_ADDRESS external *)
 opns
   Local_Available_Ptr (*! implementedby ADT_LOCAL_AVAILABLE_PTR external *),
   Available_Ptr (*! implementedby ADT_AVAILABLE_PTR external *),
   Available_Lock (*! implementedby ADT_AVAILABLE_LOCK external *) ,
   Incoming_Head_Ptr (*! implementedby ADT_INCOMING_HEAD_PTR external *),
   Incoming_Tail_Ptr (*! implementedby ADT_INCOMING_TAIL_PTR external *),
   Incoming_Lock (*! implementedby ADT_INCOMING_LOCK external *),
   Free_Head_Ptr (*! implementedby ADT_FREE_HEAD_PTR external *),
   Free_Tail_Ptr (*! implementedby ADT_FREE_TAIL_PTR external *),
   Pkt_Tmp (*! implementedby ADT_PKT_TMP external *)
             : ID_Processor -> Address
   Addr (*! implementedby ADT_ADDR external *) : Memory_Value -> Address
   _==_ , _<>_ : Memory_Value , Memory_Value -> Bool
 eqns
   forall A, B : Address
   ofsort Bool
     A == A = true ;
     A == B = false ;
     A <> B = not (A == B) ;
endtype
```

Nous avons implémenté la fonction d'initialisation de la mémoire ADT_INIT_MEMORY en conformité avec les traitements effectués sur les variables dans les primitives *send* et *receive*. Cette fonction correspond à l'opération *Init_Memory* dans le type abstrait **Memory**.

La figure 3.3 illustre le contenu de la mémoire après l'initialisation en indiquant la correspondance entre les indices du tableau et les noms des variables (les indices sont marqués en couleur rouge). Par exemple, la variable privée *Local_Available_Ptr* du processus P_0 (resp. P_1) est d'indice 0 (resp. 11) en mémoire et son contenu est initialisé à 9 (resp. 19).

Les indices 9 et 10 dans la figure 3.3 correspondent aux adresses des paquets de P_0 disponibles pour l'émission. Ici le nombre de paquets est fixé à 2 pour chaque processus. Notons que les deux processus possèdent toujours le même nombre de paquets.

Dans l'étude de l'algorithme de *ping-pong* le nombre de paquets alloués initialement représente une limite dans l'analyse de son comportement, notamment durant la phase de l'évaluation des performances, l'explosion de l'espace d'états se produisant pour un nombre très petit de paquets (> 3) (voir la section 5.3.1)

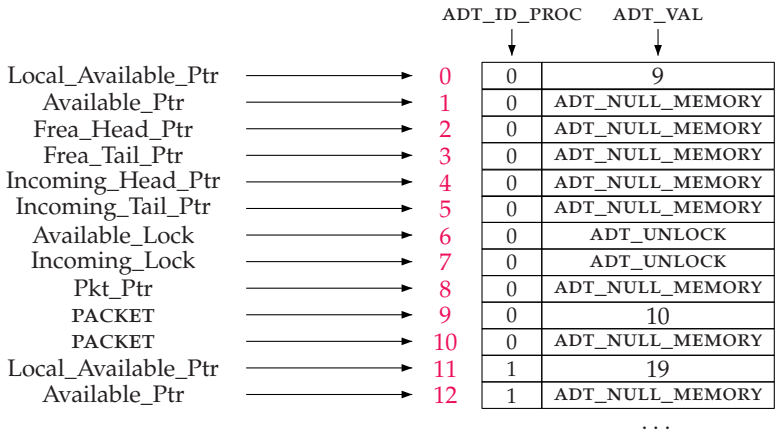


FIG. 3.3 – *L'organisation du tableau mémoire du ping-pong*

• **Gestion des verrous**

La prise de verrou dans les primitives *send* et *receive* est effectuée par l'invocation de la fonction *Lock()*. Elle consiste en la lecture du contenu de la variable verrou, suivie d'une écriture si le contenu lu indique que le verrou est libre, sinon une attente active sous forme d'une boucle (*spin*). Quant à la libération du verrou, elle est effectuée par l'appel à la fonction *Unlock()* qui modifie son contenu.

La modélisation de la gestion des verrous en LOTOS par un processus qui s'appelle récursivement si la condition de la prise du verrou n'est pas vérifiée suite à une simple lecture, s'avère insuffisante en raison de la simultanéité des demandes d'accès. Pour plus de précision nous proposons l'exemple suivant (3.6).

Exemple 3.6 *Soit le processus* TAKE_LOCK_PROC *modélisant la prise de verrou* var_lock *:*

```
process   TAKE_LOCK_PROC [ACTION] (P : ID_Processor) : noexit :=
 ACTION   ! P ! var_lock ! Load ? val_lock :Memory_Value;
 (
   [val_lock == UnLock] ->
            ACTION   ! P  ! var_lock  ! Store ! Lock;
            TAKE_LOCK_PROC [ACTION](P)
      []
   [val_lock <> UnLock] -> TAKE_LOCK_PROC [ACTION](P)
 )
endproc
```

Le premier rendez-vous est une demande de lecture du contenu du verrou var_lock, *suivi d'un choix déterministe basé sur la valeur lue : si elle est égale à* UnLock *alors le verrou est libre, ainsi une demande d'écriture a lieu pour la prise de verrou (positionner son contenu à* Lock*), sinon,* TAKE_LOCK_PROC *se rappelle récursivement pour une redemande de verrou.*

Soit la composition parallèle entre les deux processus P_0 et P_1 pour la prise de verrou var_lock *:*

TAKE_LOCK_PROC [ACTION](p_0) ||| TAKE_LOCK_PROC [ACTION](p_1)

Initialement var_lock *n'est valide qu'en mémoire et son contenu est égal à* UnLock.

Le tableau 3.2 ci-dessous illustre un scénario de conflit dans la prise du verrou var_lock. *À l'instant t_1, le processus P_0 accède à* var_lock *en lecture et d'après son contenu la condition de prise de verrou est bien vérifiée, il procède alors à son écriture à l'instant t_3. Cependant, à l'instant t_2, avant bien que P_0 prenne effectivement le verrou, le processus P_1 réalise à son tour une demande d'accès en lecture de* var_lock *qu'est toujours libre, lui permettant ainsi un accès en écriture à* var_lock *à l'instant t_4.*

La condition de prise de verrou est vérifiée pour les deux processus en même temps, ce qui représente un comportement erroné dans le principe de gestion des verrous.

Instant	En mémoire	P_0			P_1		
		Accès	État	En cache	Accès	État	En cache
t_0	*UnLock*	–	I	–	–	I	–
t_1	*UnLock*	*Load*	E	*UnLock*	–	I	–
t_2	*UnLock*	–	S	*UnLock*	*Load*	S	*UnLock*
t_3	*UnLock*	*Store*	M	*Lock*	–	I	*UnLock*
t_4	*Lock*	–	I	*Lock*	*Store*	M	*Lock*

TAB. 3.2 – *Exemple d'un scénario d'une prise de verrou en conflit*

Pour assurer une bonne gestion des verrous et éviter les conflit, nous avons remplacé les opération d'accès en lecture et en écriture aux variables verrous dans le processus TACK_LOCK_PROC par un accès lecture-écriture automatique en utilisant l'opération *Test_and_Set*, dont le principe de son fonctionnement est le suivant :

Sur le même rendez-vous d'accès à la variable verrou :

(a) si le verrou est libre, c'est à dire que son contenu est égale à *UnLock*, alors une prise de verrou à lieu par la mise à jour de son contenu à *Lock*,

(b) sinon, il s'agit d'une simple lecture.

Dans la nouvelle version du processus TACK_LOCK_PROC illustré ci-dessous, le premier rendez-vous a lieu si la variable verrou est libre, suivi de l'action *exit* pour mettre fin au processus puisque le verrou est pris par la modification de son contenu par l'opération *Test_and_Set* (prise de verrou implicite). Le second rendez-vous correspond à une simple lecture, car le verrou est pris par un autre processus, suivi d'un appel récursif du processus TACK_LOCK_PROC.

```
process   TAKE_LOCK_PROC [ACTION] (P : ID_Processor) : noexit :=
 ACTION   ! P   ! var_lock   ! Test_and_Set   ! Unlock ;
 exit
   []
 ACTION   ! P   ! var_lock   ! Test_and_Set   ! Lock ;
 TAKE_LOCK_PROC [ACTION](p)
endproc
```

• Spécification LOTOS

La spécification LOTOS de l'algorithme *ping-pong* est composition des processus suivants :

- SEND : modélisant le comportement de la primitive *send*.
- RECEIVE : modélisant le comportement de la primitive *receive*.
- PING : modélisant le comportement infini de la séquence <*send* ; *receive*>.
- PONG : modélisant le comportement infini de la séquence <*receive* ; *send*>.
- TRANSFER : modélisant la gestion de la mémoire et les caches.

La figure 3.4 est une représentation graphique de la spécification LOTOS du *ping-pong*, mettant en évidence les synchronisations entre les différents processus. Le processus central TRANSFER est paramétré par la mémoire et les caches, et les processus SEND, RECEIVE, PING et PONG sont paramétrés par les identificateurs des processus participant à l'exécution du *ping-pong* : P_0 et P_1.

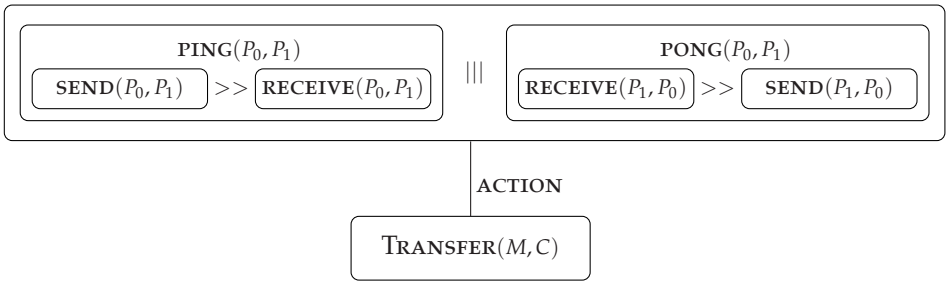


FIG. 3.4 – *Spécification LOTOS du ping-pong*

• Génération de l'espace d'états

Le tableau 3.3 indique la taille du graphe BCG représentant le comportement fonctionnel du *ping-pong* pour un nombre de paquets allant de 1 à 3.

Seule la variation du protocole de cohérence de caches impacte la taille de l'espace d'états. Le protocole de caches *B* engendre des graphes BCG plus petits pour un nombre de paquet > 1, que ceux générés en utilisant le protocole de caches *A*. Une différence de 210 états dans le cas de 2 paquets et de 18.708 états dans le cas de 3 paquets.

L'architecture du système ainsi que la topologie n'ont aucun impact sur la taille de l'espace d'états. Pour un nombre de paquet donné et un

protocole de caches donné, nous obtenons des graphes BCG de tailles identiques quelque soit l'architecture (FAME, MESCA) et la topologie ($topology_0$, $topology_1$, $topology_2$).

Paquets	Protocole *A*		Protocole *B*	
	États	Transitions	États	Transitions
1	1.732	3.464	1.737	3.474
2	23.571	47.142	23.355	46.710
3	1.733.055	3.466.110	1.714.347	3.428.694

TAB. 3.3 – *La taille de l'espace d'états du comportement fonctionnel de l'algorithme de* ping-pong

La croissance rapide de la taille des graphes provoque si vite une explosion d'espace d'états malgré la variation très modeste du nombre de paquets. Par exemple, la taille du graphe pour 3 paquets est 1000 fois plus grand que celui pour un seul paquet : la différence est incontestable.

3.3.2 Algorithme barrière *tournament*

La barrière *tournament* a été proposée par Hensgen, Finkel et Manber dans [17]. Son principe de fonctionnement est similaire au jeu du tournoi : deux processus jouent les uns contre les autres à chaque tour (*round*). Le processus gagnant est connu d'avance et attend l'arrivée des processus perdants. Les gagnants jouent les uns contre les autres au tour suivant, comme illustré dans la figure 3.5. Le gagnant final devient *champion* et informe tous les processus perdants de la fin de la barrière.

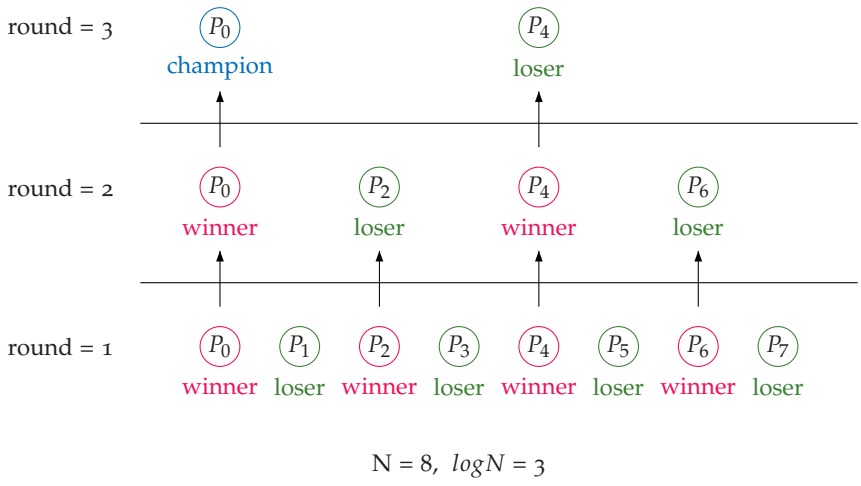


FIG. 3.5 – *Le jeu de tournoi dans la barrière tournament*

L'exécution complète de *tournament* nécessite $logN$ tours, avec N le nombre de processus participant à l'exécution de la barrière.

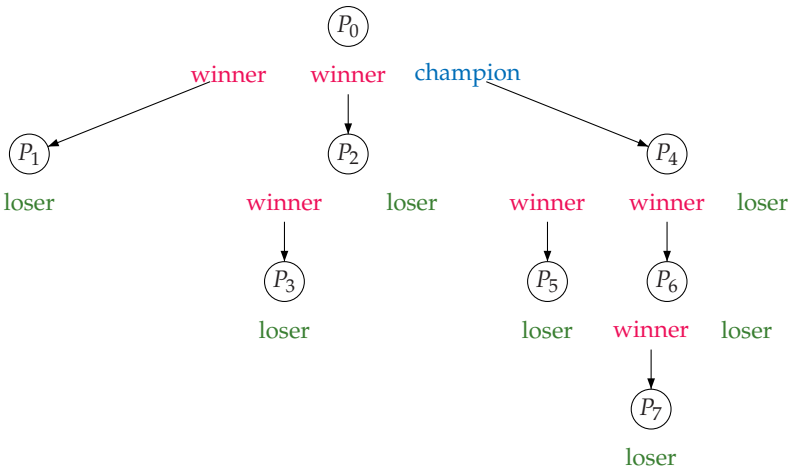


FIG. 3.6 – *La propagation de l'information "*fin de barrière*" entre les processus*

Algorithme 4 : La barrière *tournament*

```
Data :
  type round_t : record
        role : (winner, loser, champion, bye, dropout)
        opponent :^boolean
        flag : boolean
    shared rounds : array [0 .. N-1][0 .. LogN] of rounds
processor private sense : boolean := true
processor private vpid : integer        /* a unique virtual processor index */
  begin
    /* initially                                                              */
    /* rounds[i][k].flag = false for all i, k                                 */
    /* rounds[i][k].role ..                                                   */
    /* winner if k>0, i mod 2^k=0, i+2^(k-1)<N and 2^k<N                      */
    /* bye if k>0, i mod 2^k=0 and i+2^(k-1)>=N                               */
    /* loser if k>0 and i mod 2^k=2^(k-1)                                     */
    /* champion if k>0, i=0 and 2^k>=N                                        */
    /* dropout if k=0                                                         */
    /* unused otherwise; value immaterial                                     */
    /* rounds[i][k].opponent points to                                        */
    /* rounds[i-2^(k-1)][k].flag if rounds[i][k].role = loser    */
    /* rounds[i+2^(k-1)][k].flag if rounds[i][k].role = winner or
    champion                                                                  */
    /* unused otherwise; value immaterial                                     */
    procedure tournament_barrier
      round : integer := 1
        loop                                                  /* arrival */
          case rounds[vpid][round].role of
            loser : rounds[vpid][round].opponent^:= sense
                    repeat until rounds[vpid][round].flag = sense
                    exit loop
            winner : repeat until rounds[vpid][round].flag = sense
            bye :                                          /* do nothing */
            champion : repeat until rounds[vpid][round].flag = sense
                       rounds[vpid][round].opponent^:= sense
                       exit loop
            dropout :                                      /* impossible */
      round := round + 1
        loop                                                   /* wakeup */
      round := round - 1
          case rounds[vpid][round].role of
            loser :                                        /* impossible */
            winner : rounds[vpid][round].opponent^:= sense
            bye :                                          /* do nothing */
            champion :                                     /* impossible */
            dropout : exit loop
      sense := not sense
  end
```

Nous avons modélisé la primitive de barrière *tournament* proposée dans [68]. La particularité dans cette primitive réside dans son principe de mise en fin de l'exécution d'un cycle de barrière. En effet, les auteurs proposent une technique basée sur la propagation de l'information "*fin de barrière*" du processus *père* vers les processus *fils* suivant une organisation sous forme d'un arbre binomial d'ordre $logN$ (voir l'annexe A.1).

L'interprétation en terme de jeu de tournoi est la suivante : le processus gagnant informe les processus perdants avec lesquels il joue de la fin de la compétition, qui correspondent à ses fils dans l'arbre binomial, comme illustré dans la figure 3.6 : P_0 informe P_1, P_2 et P_4, le processus P_2 informe P_3, P_4 informe les processus P_5, P_6 et, le processus P_6 informe P_7.

L'exécution de l'algorithme de barrière *tournament* est répartie en 3 phases principales :

1. La phase d'*initialisation* : exécutée uniquement au premier lancement de la barrière. Sa mission est l'attribution des rôles (*winner*, *loser*, *champion*, ...) aux différents processus en fonction de leurs identificateurs.
2. La phase "*arrivée à la barrière*" (*arrival*) : décrit l'arrivé d'un processus à la barrière et les mises à jour de variables en fonction de son rôle.
3. La phase "*fin de la barrière*" (*wakeup*) : décrit la fin du cycle de barrière. Les processus gagnant mettent fin à l'attente active des processus par la mise à jour de variables.

Chaque processus participant à l'exécution de la barrière *tournament* possède 4 variables :

- *sense* : est une variable privée de type booléen. Son contenu peut être considéré comme un pseudo-identifiant du cycle de la barrière en cours. À l'arrivée à la barrière, tous les processus doivent avoir la même valeur de *sense* et à la sortie ils inversent son contenu de telle sorte que deux exécutions consécutives de *tournament* ne doivent pas avoir la même valeur de *sense*. Ainsi, s'il existe un processus possédant une valeur différente cela signifie qu'il n'a pas encore quitté l'ancien cycle de barrière et que les autres processus ont entamé un nouveau cycle ou inversement, dans les deux cas il s'agit d'une erreur fatale dans l'exécution de la barrière.
- *role* : est un tableau de $logN$ éléments. Il prend ses valeur depuis l'ensemble des rôles : *winner*, *loser*, *champion*, *bye*, *dropout*. Les indices du tableau correspondent aux différents tours possibles (*round*). Chaque élément dans le tableau correspond au rôle du processus dans les différents tours.
- *flag* : est un tableau de $logN$ booléens. Chaque élément indique l'arrivée d'un processus fils à la barrière. Par exemple dans le cas du processus P_0 le premier élément de son tableau *flag* n'est pas exploité, le deuxième décrit l'état d'arrivé du processus P_1, le troisième celui de P_2 et le dernier celui de P_4.
- *opponent* : est un tableau de $logN$ pointeurs vers les éléments de *flag*. Par exemple pour le processus P_0 le premier élément n'est pas

utilisé, le deuxième pointe vers le deuxième élément de *flag* de P_1, le troisième vers le troisième élément de *flag* de P_2 et le dernier élément vers le quatrième élément de P_4.

• Modélisation des structures de données

Dans l'algorithme de *tournament*, les variables partagées de chaque processus sont regroupées dans une seule structure nommée *round_t* et par la suite tous les *round_t* de tous les processus seront regroupées à leur tour dans tableau nommé *rounds*. Par souci de simplicité et d'efficacité nous avons cassé ce regroupant en considérant l'indépendance des variables.

La construction suivante décrit la modélisation en LOTOS des variables de l'algorithme barrière *tournament* :

```
type Address is Natural, ID_Processor, Memory_Value
 sorts Address  (*! implementedby ADT_ADDRESS
                   printedby ADT_PRINT_ADDRESS external *)
 opns
   sense  (*! implementedby ADT_SENSE external *) : ID_Processor -> Address
   opponent  (*! implementedby ADT_OPPONENT external *),
   role  (*! implementedby ADT_ROLE external *) ,
   flag  (*! implementedby ADT_FLAG external *),
          : ID_Processor, Memory_Value -> Address
   Addr  (*! implementedby ADT_ADDR external *) : Memory_Value -> Address
   _==_ , _<>_ : Memory_Value , Memory_Value -> Bool
 eqns
   forall A, B : Address
   ofsort Bool
     A == A = true ;
     A == B = false ;
     A <> B = not (A == B) ;
endtype
```

L'appel à un élément des tableaux *role* et *flag* s'effectue par la précision de l'identificateur du processeur possédant la variable ainsi que le numéro du tour (*rond*) qui est de type **Memory_Value**. Par exemple *role(Proc(0), Memory_Val(1))* indique le rôle du processus P_0 au premier tour (qui est égal à *winner* d'après la procédure d'initialisation). Suivant la procédure d'initialisation de l'algorithme *tournament*, nous avons défini une fonction nommée ADT_INIT_ROLE. Cette dernière prend en entrée l'identificateur du processus et un numéro de tour et, retourne en sortie le rôle correspondant.

```
ADT_MEMORY_VALUE  ADT_INIT_ROLE (ADT_ID_PROC, ADT_ROUND)
ADT_ID_PROCESSOR  ADT_ID_PROC;
ADT_MEMORY_VALUE  ADT_ROUND;
{
 int pw_2k, pw_2k_1, i_mod_2k;
 if (ADT_ROUND == 0)  return (ADT_DROPOUT());
 else {
       pw_2k = pw(2, ADT_ROUND);
       pw_2k_1 = pw(2, ( ADT_ROUND-1));
       i_mod_2k = ( ADT_ID_PROC) % (pw_2k);
       if ((i_mod_2k == 0) && (ADT_ID_PROC + pw_2k_1) && (pw_2k < ADT_NB_PROC))
         return (ADT_WINNER());
       else if ((i_mod_2k == 0) && ((ADT_ID_PROC + pw_2k_1) >= ADT_NB_PROC))
         return (ADT_BYE());
       else if (i_mod_2k == pw_2k_1 ) return (adt_loser());
       else if ((ADT_ID_PROC == 0) && (pw_2k >= ADT_NB_PROC)) return (ADT_CHAMPION());
       else return (ADT_UNUSED());
       }
}
```

La tableau mémoire comporte uniquement le contenu des variables *sense*, *role* et *flag*. Pour des raisons de simplicité et d'économie dans taille du tableau mémoire, on considère que les variables *opponent* et *flag* possèdent les même indices dans le tableau mémoire. Ceci par le fait que la variable *opponent* est un pointeur vers *flag*. Nous avons implémenté la variable *opponent* au moyen de la fonction ADT_OPPONENT qui prend en entrée l'identificateur du processus et le numéro du tour et retourne l'indice de la variable *flag* dans le tableau mémoire (suivant la fonction d'initialisation de l'algorithme) :

```
ADT_ADDRESS  ADT_FLAG (ADT_ID_PROC, INDICE_K)
ADT_ID_PROCESSOR  ADT_ID_PROC;
ADT_ADDRESS  INDICE_K;
  { return ((ADT_ID_PROC * (ADT_NB_VAR)) + ADT_FLAG_RAW + INDICE_K); }

ADT_ADDRESS  ADT_OPPONENT(ADT_ID_PROC, ADT_K)
ADT_ID_PROCESSOR  ADT_ID_PROC;
ADT_MEMORY_VALUE  adt_k;
{
 int pow_2k_1;
 ADT_MEMORY_VALUE  ADT_ROLE = ADT_INIT_ROLE (ADT_ID_PROC, adt_k);
 pow_2k_1 = pow(2, (adt_k-1));
 if (ADT_ROLE == ADT_LOSER()) return (ADT_FLAG((ADT_ID_PROC - pow_2k_1), adt_k));
 else if ((ADT_ROLE == ADT_WINNER()) || (ADT_ROLE == ADT_CHAMPION()))
   return (ADT_FLAG((ADT_ID_PROC + pow_2k_1), adt_k));
 else return (ADT_FLAG(ADT_ID_PROC, 0));
}
```

• **Spécification LOTOS**

On spécifie en LOTOS le comportement d'un algorithme de barrière entre n processus par une composition parallèle entre eux et une synchronisation avec le processus TRANSFER sur la porte ACTION comme suit :

BARRIER [ACTION, ERROR](P_0) ||| ... ||| BARRIER [ACTION, ERROR](P_{n-1})
|[ACTION]|
TRANSFER [ACTION] (Init_Memory, Init_Cache)

Le processus BARRIER spécifie un comportement récursif de fonctionnalité *noexit*. Il est paramétré par deux portes : ACTION qui assure les demandes d'accès aux variables et ERROR qui spécifie des erreurs dans l'exécution de l'algorithme.

```
process  BARRIER [ACTION, ERROR] (vpid : ID_Processor) : noexit :=
 ARRIVAL_LOOP [ACTION, ERROR](vpid, Memory_Val(1))
 >>
 ACTION  ! vpid ! sense(vpid) ! Load ? sense_val :Memory_Value
         ? latence :Latency_Value ;
 ACTION  ! vpid ! sense(vpid) ! Store ! Inv_Value(sense_val)
         ? latence :Latency_Value ;
 BARRIER [ACTION, ERROR](vpid)

 where
 ...
endproc
```

Le processus ARRIVAL_LOOP spécifie le comportement de la phase "*d'arrivée à la barrière*", il est paramétré par les deux porte ACTION et ERROR et par l'identificateur du processeur et la contenu de la variable *round*, qui initialement vaut *Memory_Val(1)*. Le processus WAKEUP_LOOP spécifie le comportement de la phase "*fin de la barrière*".

A la fin de l'exécution du processus ARRIVAL_LOOP deux rendez-vous sur la porte ACTION auront lieu, le premier modélise un accès en lecture à la variable *sense* et le second un accès en écriture inversant le contenu de *sense*.

Le comportement du processus ARRIVAL_LOOP spécifie 5 choix déterministes en fonction du rôle :

- Le premier choix a lieu quand le processus *vpid* a pour rôle *loser* au tour *round*. Dans ce cas le processus informe son arrivé à la barrière par un rendez-vous pour un accès en lecture de *sense*, ensuite un rendez-vous en écriture en positionnant le contenu de son *opponent* à la valeur de *sense* précédemment lue. Par la suite il se lance dans une attente active de l'arrivé de tous les processus à la barrière en exécutant le processus LOTOS SPIN. L'instruction *exit loop* correspond un saut vers l'exécution de la boucle *wakeup* qui est modélisée par un appel au processus WAKEUP_LOOP.
- Le second choix a lieu quand le processus *vpid* a pour rôle *winner* au tour *round*. Dans ce cas, il se lance dans une attente active du

processus qui pointe vers son *flag(vpid, round)*, par la suite il appelle le processus ARRIVAL_LOOP pour le tour suivant *Addition(vpid, Memory_Val(1))*.

- Le troisième choix a lieu quand le processus *vpid* a pour rôle *bye* au tour *round*, dans ce cas il n'effectue aucun traitement sur les variable et fait appel au processus ARRIVAL_LOOP pour le tour suivant.
- Le quatrième choix a lieu quand le rôle du processus *vpid* est *champion* où il effectue un traitement similaire à celui qui à pour rôle *loser*.
- Le cinquième choix a lieu quand le rôle du processus pour le tour *round* est *dropout*, ce qui représente une erreur dans la phase d'initialisation et l'attribution des rôles aux processus à chaque tour, car un processus à pour rôle *dropout* avant le début de la compétition du tournoi c'est-à-dire quand le tour est égale à 0. Nous signalons cette erreur par un rendez-vous sur la porte ERROR en indiquant l'identificateur du processus pour faciliter la correction.

```
process  ARRIVAL_LOOP [ACTION, ERROR](vpid :ID_Processor, round :Memory_Value)
         : noexit :=

 ACTION  ! vpid ! role(vpid, round) ! Load  ! loser  ? latence ;
 ACTION  ! vpid ! sense(vpid) ! Load  ? sense_val  ? latence ;
 ACTION  ! vpid ! opponent(vpid, round) ! Store !sense_val ?latence ;
 SPIN [ACTION](vpid, round, sense_val) »
 WAKEUP_LOOP [ACTION,ERROR](vpid, Minus(round, Memory_Val(1)))
  []
 ACTION  ! vpid ! role(vpid, round) ! Load ! winner ? latence ;
 ACTION  ! vpid ! sense(vpid) ! Load ? sense_val ? latence ;
 SPIN [ACTION](vpid, round, sense_val) >>
 ARRIVAL_LOOP [ACTION, ERROR](vpid, Addition(round, Memory_Val(1)))
  []
 ACTION  ! vpid ! role(vpid, round) ! Load ! bye ? latence ;
 ARRIVAL_LOOP [ACTION, ERROR](vpid, Addition(round, Memory_Val(1)))
  []
 ACTION  ! vpid ! role(vpid, round) ! Load ! champion ? latence ;
 ACTION  ! vpid ! sense(vpid) ! Load ? sense_val ? latence ;
 SPIN [ACTION](vpid, round, sense_val) >>
 ACTION  ! vpid ! sense(vpid) ! Load ? sense_val ? latence ;
 ACTION  ! vpid ! opponent(vpid,round) ! Store ! sense_val ? latence ;
 WAKEUP_LOOP [ACTION, ERROR](vpid, Minus(round, Memory_Val(1)))
  []
 ACTION  ! vpid ! role(vpid, round) ! Load ! dropout ? latence ;
 ERROR  ! vpid ;
 exit
endproc
```

• Génération de l'espace d'états

Le tableau 3.4 présente la taille de l'espace d'états du graphe BCG spécifiant le comportement fonctionnel de l'algorithmde barrière *tournament*, en fonction du nombre de processus participant (de 2 à 7) à son exécution.

Seul le nombre de processus participant à la barrière impact la taille de l'espace d'états. En effet la taille des graphes BCG augmente en exponentielle en fonction du nombre de processus . Une explosion d'espace d'états à lieu pour un nombre de processus ≥ 8

Processus	États	Transitions
2	140	280
3	1.136	3.408
4	6.582	26.328
5	10.720	53.600
6	30.208	181.248
7	187.072	1.309.504

TAB. 3.4 – *La taille de l'espace d'états du comportement fonctionnel de l'algorithme de barrière* tournament

3.3.3 Algorithme barrière *combining tree*

La barrière *combining tree* a été introduite par Yew, Tzeng et Lawrie [95] dans le but de réduire le problème de contention dans la barrière *centralized*. Elle consiste en une organisation décentralisée des processus sous forme d'un arbre combinatoire. Chaque processus dans l'arbre possède une variable (*count*) indiquant le nombre de ses fils y compris lui-même, comme indiqué dans la figure 3.7.

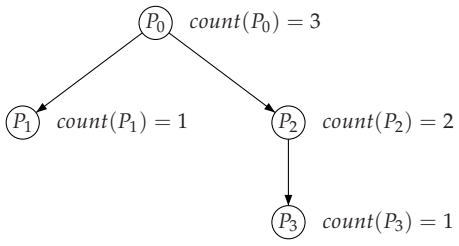


FIG. 3.7 – *Organisation des processus en arbre combinatoire dans la barrière* combining tree

Remarque 3.3 *On distingue dans l'arbre combinatoire deux types de processus père : direct et indirect. Par exemple dans la figure 3.7, le processus P_0 est un processus père direct de P_1 et P_2 et un père indirect de P_3 (P_2 est un processus père direct de P_3).*

Nous avons modélisé la version optimisée de la barrière *combining tree* proposée dans [68]. Pour des raisons de clarté, nous présentons l'algorithme avec la précision des identificateurs des processeurs.

Algorithme 5 : La barrière *combining tree*

```
Data :
  type node : record
        k : integer                                  /* fan in of this node */
        count : integer                              /* initialized to k */
        lock_sense : boolean                          /* initially false */
        parent :^node              /* pointer to parent node nil if root */
   shared nodes : array [0 .. N-1] of node
   processor private sense : boolean := true
   processor private mynode :^node          /* my group's leaf in the tree */
begin
    procedure combining
      combining_barrier_aux (Pi, Pi)                 /* join the barrier */
      sense := not sense                             /* for next barrier */
    procedure combining_barrier_aux (Pi :^node, Pj :^node)
      with Pi^do
        if (fetch_and_decrement (count(Pj)) = 0)
          if (parent(Pj) != nil)
            combining_barrier_aux (Pi, parent(Pj))
          endif
          count(Pj) := k(Pj)                    /* prepare for next barrier */
          lock_sense(Pj) := not lock_sense(Pj)  /* release waiting processors */
        endif
        repeat until lock_sense(Pj) = sense(Pi)
end
```

Quand un processus P_i arrive à la barrière, il décrémente la variable $count(P_i)$ pour informer ses processus fils de son arrivée, par la suite :

- Si le contenu de $count(P_i)$ résultant est égal à 0 et que $i \neq 0$ (P_i n'est pas le processus racine), alors il informe son processus père de son arrivée en exécutant la procédure *combining_barrier_aux* avec les variables du processus père de P_i : *combining_barrier_aux(Pi, parent(Pj))*. Le processus *Parent(Pj)* peut être le processus père direct ou l'un des processus père indirect de P_i en fonction du niveau d'appel de la procédure *combining_barrier_aux*.
- Si le contenu de $count(P_i)$ résultant est différent de 0, alors P_i se met en attente active sur la variable *lock_sense* du processus de la *combining_barrier_aux* en cours (P_j).
- Si le contenu de $count(P_i)$ résultant est égale à 0 et $i = 0$, alors il y a une réinitialisation des variables *count* et *lock_sense* de tous les processus qui ont été appelés par l'exécution de la procédure *combining_barrier_aux* dans P_i.

La difficulté dans la modélisation de cet algorithme réside dans la récursivité non-terminale de la fonction *combining_barrier_aux* à cause des instructions de mise à jour des variables *count* et *lock_sense* à la fin de chaque appel. Par conséquent nous avons réécrit la fonction *combining_barrier_aux* en ajoutant un mécanisme de sauvegarde des appels récursifs, basé sur l'exploitation de l'organisation des processus d'un arbre combinatoire pour déduire la trace des appels. Ceci évite la modélisation d'une pile de contextes et n'introduit pas des accès mémoire supplémentaires.

```
1.  procedure new_combining_barrier_aux(Pi :=^node, Pj :=^node)
2.   with   Pi^  do
3.    count = fetch_and_decrement (count(Pj))
4.    while (count = 0) and (p arent(Pj) != nil)
6.     do
7.      Pj = parent(Pj)
8.      count = fetch_and_decrement (count(Pj))
9.    endwhile
10.    if (count = 0)
11.     count(Pj) := k(Pj)
12.     lock_sense(Pj) := not lock_sense(Pj)
13.    endif
14.    repeat until  lock_sense(Pj) = sense(Pi)
15.    Pk = child (Pj, Pi)
16.    while (Pk != nil)
17.     do
18.      count(Pk) := k(Pk)
19.      lock_sense(Pk) := not lock_sense(Pj)
20.      repeat until  lock_sense(Pk) = sense(Pi)
21.      Pk = child (Pk, Pi)
22.    endwhile
```

Le principe du fonctionnement de la nouvelle procédure *new_combining_barrier_aux* est le suivant :

- *lignes 3-9* : le processus P_i informe de son arrivée à la barrière en décrémentant les variables *count* de ses processus père (direct et indirect).
- *lignes 10-13* : le processus P_i met à jour les variables *count* et *lock_sense* du processus racine P_0.
- *ligne 14* : cette instruction est exécutée si la sortie de l'exécution de P_i de la boucle des *lignes 4-9* est due à une variable *count* d'un processus P_j différente de 0, ce qui signifie que P_j n'est pas encore arrivée à la barrière, alors P_i se met en attente.
- *ligne 15-22* : dans ces instructions P_i informe de la fin de barrière les processus qu'il a déjà informé de son arrivée à la barrière (ces processus père direct et indirect). La fonction *child(Pj, Pi)* retourne le processus père direct ou indirecte du P_i en partant du processus P_j.

Exemple 3.7 *Avec l'organisation d'arbre combinatoire de la figure 3.7, on suppose que le processus P_3 est le* dernier *processus arrivant à la barrière. On peut distinguer deux phases dans l'exécution de P_3 de* combining_barrier_aux *:*

1. *Arrivée à la barrière : P_3 décrémente la variable* count *des processus P_3, P_2 et P_0 dans cet ordre : le contenu de toutes les variables* count *après la décrémentation est égal à* 0 *(*lignes 3-9*).*
2. *Fin de la barrière : P_3 informe d'abord le processus P_0 de la fin de la barrière par la mise à jour de ses variables* cout *et* lock_sense *(*lignes 10-13*), par la suite il réalise le même traitement avec les variables de P_2 et ensuite celles de P_3.*

Nous avons implémenté l'arbre combinatoire en C par un tableau d'entiers à deux colonnes : la première indique l'identificateur du processus parent et la seconde la valeur de la variable *count*. Dans le fragment du code C ci-dessous on décrit l'implémentation de l'arbre combinatoire de la figure 3.7 :

```
#define ADT_NB_PROC  4
unsigned char  ADR_TREE[ADT_NB_PROC][2] {
  /* ID_PARENT,  COUNT   */
/*P0 */ {ADT_NB_PROC, 3 },
/*P1 */ { 0,          1 },
/*P2 */ { 0,          2 },
/*P3 */ { 2,          1 }};
```

Le processus LOTOS COMBINING_BARRIER_AUX décrit ci-dessous spécifie la fonction *new_combining_barrier_aux* par un comportement de fonctionnalité *exit*, avec le processus WHILE_COUNT_PARENT, WHILE_UPDATE et SPIN qui traduisent les instructions des *lignes 3-14*, *lignes 15-22* et de la *ligne 14* respectivement.

• Génération de l'espace d'états

Dans l'algorithme de *combining tree* il n'y a aucune indication sur la structure de l'arbre (la profondeur, le nombre de fils, ..., etc). En effet le nombre d'arbres possibles devient important en augmentant le nombre de processus, par conséquent nous avons considéré que 3 arbres au maximum. La figure 3.8 illustre les arbres combinatoires que nous avons utilisés.

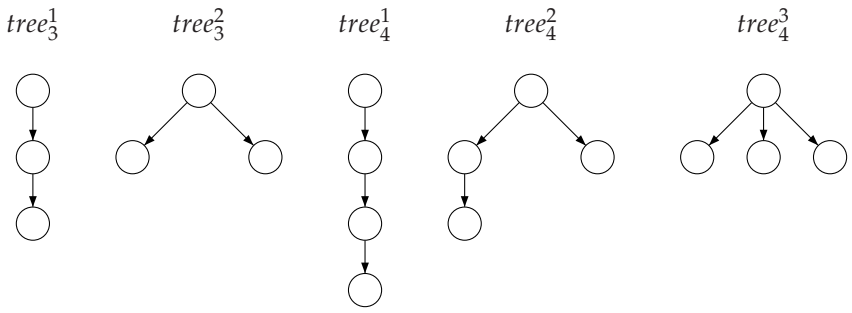


FIG. 3.8 – *Organisation des processus en arbre combinatoire pour* 3 *et* 4 *processus*

La taille des graphes BCG spécifiant le comportement fonctionnel de la barrière *combining tree* est sensible non seulement au nombre de processus participant à son exécution mais notamment à l'organisation décentralisée des processus sous forme d'arbre combinatoire ainsi que le protocole de cohérence de caches.

On constate, que d'après le table 5.13, la taille des graphes BCG diminue quand le nombre de fils du processus racine de l'arbre (P_0) augmente. Par exemple, pour 4 processus, la taille de graphe BCG quand le nombre de fils de la racine est égal à 1 ($tree_4^1$) est deux fois plus grand que quand le nombre de fils est égal à 3 ($tree_4^3$).

On constate également que le protocole de cohérence de cache *B* engendre des graphes nettement plus petits à ceux engendrés par le protocole *A*.

Conclusion du chapitre

Dans ce chapitre, nous avons proposé une méthodologie pour la modélisation en langage LOTOS l'environnement matériel et logiciel de notre système.

Pour une modélisation efficace et suffisante, nous avons effectué quelques abstractions de certains détails liés à l'implémentation physique. Par exemple, on considère que la mémoire est d'une structure centralisée et que les caches comme entités indépendantes des processeurs et ne comportent que l'état des données en caches (M, E, S, I).

Les exemples d'application des trois algorithmes MPI (*ping-pong*, la barrière *tournament* et la barrière *combining tree*) montrent que chaque algorithme possède une spécification qui lui est propre et, qui nécessite par fois l'ajout de nouvelles structures de données, comme la définition de la structure implant l'arbre combinatoire dans le cas de la barrière *combining tree*.

Le nombre de processeurs participants à l'exécution de l'algorithme MPI n'est pas le seul facteur impactant la taille de l'espace d'états du comportement fonctionnel de ces algorithmes, on note également l'impact de la topologie d'interconnexion ainsi que le protocole de cohérence de caches utilisé.

```
process  COMBINING_BARRIER_AUX [ACTION] (Pi :ID_Processor) : exit :=
WHILE_COUNT_PARENT [ACTION] (Pi, Pi)
where

process  WHILE_COUNT_PARENT [ACTION] (Pi :ID_Processor, Pj :ID_Processor) : exit :=
 (
  ACTION  ! Pi ! count(Pj) ! Fetch_and_Decrement ! Memory_Val(0) ? latence ;
  [Parent(Pj) <> Null_Proc]-> WHILE_COUNT_PARENT [ACTION] (Pi, Parent(Pj))
   []
  [Parent(Pj) == Null_Proc]->
   ACTION  ! Pi ! count(Pj) ! Store ! K_count(Pj) ? latence ;
   ACTION  ! Pi ! lock_sense(Pj) ! Load ? lock_sense_val ? latence ;
   ACTION  ! Pi ! lock_sense(Pj) ! Store ! Inv_Value(lock_sense_val) ? latence ;
   ACTION  ! Pi ! sense(Pi) ! Load ? sense_val ? latence ;
   SPIN [ACTION] (Pi, Pj, sense_val) >>
   UPDATE_PROC [ACTION] (Child(Pj, Pi), Pi)
 )
  []
 (
  ACTION  ! Pi ! count(Pj) ! Fetch_and_Decrement ? count_val ? latence
            [count_val <> Memory_Val(0)] ;
  ACTION  ! Pi ! sense(Pi) ! Load ? sense_val ? latence ;
  SPIN [ACTION] (Pi, Pj, sense_val) >>
  UPDATE_PROC [ACTION] (Child(Pj, Pi), Pi)
 )
endproc

process  WHILE_UPDATE [ACTION] (Pi :ID_Processor, Pk :ID_Processor) : exit :=
[Pk <> Null_Proc]->
  ACTION  ! Pi ! count(Pk) ! Store ! K_count(Pk) ? latence ;
  ACTION  ! Pi ! lock_sense(Pk) ! Load ? lock_sense_val ? latence ;
  ACTION  ! Pi ! lock_sense(Pk) ! Store ! Inv_Value(lock_sense_val) ? latence ;
  ACTION  ! Pi ! sense(Pi) ! Load ? sense_val ? latence ;
  SPIN [ACTION] (Pi, Pk, sense_val) >>
  WHILE_UPDATE [ACTION] (Pi, Child(Pk, Pi))
  []
[Pk == Null_Proc]-> exit
endproc
```

Processus	ID_Tree	Protocole *A*		Protocole *B*	
		États	Transitions	États	Transitions
2	1	687	1.374	423	846
3	1	19.386	58.158	6.924	20.772
	2	13.567	40.701	7.199	21.597
4	1	498.896	1.995.584	103.018	412.072
	2	385.643	1.542.572	123.416	493.664
	3	231.606	926.424	113.659	454.636

TAB. 3.5 – *La taille de l'espace d'états du comportement fonctionnel de l'algorithme de barrière* combining tree

La vérification formelle

4

Sommaire

Ce chapitre peut être considéré comme une validation du travail réalisé dans le chapitre 3. Après l'obtention de la spécification LOTOS modélisant notre système et avant d'entamer la phase d'évaluation des performances, il est indispensable de vérifier que les aspects logiciels et matériels ont été correctement modélisés.

La vérification est un processus de démonstration de la correction fonctionnelle d'un système. Il existe plusieurs méthodes de vérification, les principales étant le *test*, la *preuve de théorèmes* et le *model-checking* (littéralement : "*vérification de modèle*").

Le test permet de découvrir de nombreuses erreurs, mais il ne peut pas être exhaustif et n'apporte donc que des réponses partielles. Dans la preuve de théorème, le système est considéré comme un ensemble d'axiomes qui servent à prouver des propriétés, c'est une méthode qui permet en principe de répondre à toutes les questions de vérification qui se posent en pratique, mais sa mise en œuvre est souvent lourde et compliquée. Le model-checking [11] est en quelque sorte intermédiaire : il s'agit d'une méthode exhaustive et en grande partie automatique.

Pour vérifier automatiquement un système par la méthode du model-checking, il est nécessaire de :

- Construire une modélisation formelle du système au moyen d'un *langage de spécification de système* de haut niveau ayant une sémantique opérationnelle bien définie, comme le langage LOTOS. La description du système est ensuite traduite automatiquement vers un modèle sous-jacent, qui est souvent un STE (*Système de Transitions Etiquetées*).

- Spécifier les propriétés à vérifier au moyen d'un *langage de spécification de propriétés*, comme les logiques temporelles, qui sont des formalismes bien adaptés pour spécifier les propriétés de bon fonctionnement du système.
- Enfin, il faut disposer d'un algorithme capable de déterminer si le système satisfait ou non les propriétés spécifiées. Cet algorithme est incarné dans un *model-checker* (un outil pour le model-checking), permettant de fournir un *diagnostic* d'erreur complétant utilement la vérification d'une propriété qui n'est pas satisfaite. Nous utilisons le model-checker EVALUATOR [66] de la boîte à outils CADP.

La logique temporelle a été introduite pour la première fois par Pnueli en 1977 [79] pour l'étude du comportement des programmes parallèles. C'est une forme de logique dont les énoncés et les raisonnements font intervenir la notion d'ordonnancement dans le temps. L'idée de son utilisation pour qualifier le comportement des programmes est assez naturelle, elle vient de sa manipulation des propositions dont la vérité évolue dans le temps : l'état d'un système change au cours de l'exécution du programme et, en conséquence, les propriétés de l'état changent également.

Remarque 4.1 *La notion du temps dans la logique temporelle, n'est pas explicitement utilisée comme une durée absolue. Par exemple, dans la propriété :*"chaque fois que j'observe Q, j'ai observé P avant"*, il peut a priori se passer une seconde ou une journée entre l'observation de* P *et l'observation de* Q*. Les logiques manipulant explicitement le temps sont dites des* logiques temporisées *[39].*

Les descriptions de propriétés en logiques temporelles présentent deux qualités importantes [63] : elles sont *abstraites* pour leur indépendance des détails de l'implémentation de l'application et elles sont *modulaires*, parce que la modification, l'ajout ou la suppression d'une propriété n'impacte pas la validité des autres propriétés.

Le choix d'une logique temporelle parmi la multitude des logiques temporelles existantes dans la littérature (une synthèse sur les plus représentatives peut être trouvée dans [65]), doit prendre en considération plusieurs aspects :

- *L'expressivité* : la capacité de la logique à exprimer les classes de propriétés intéressantes, telles que la sûreté, la vivacité, ou l'équité.
- *La complexité d'évaluation* : la complexité des algorithmes permettant de vérifier qu'un modèle satisfait une propriété.
- *La facilité d'utilisation* : la capacité d'exprimer les propriétés de manière concise et naturelle.

L'optimisation de l'un ou l'autre de ces aspects ne pouvant généralement se faire qu'au détriment des autres, le choix doit passer par un compromis judicieux, par exemple si l'efficacité d'évaluation est l'aspect le plus important, alors l'expressivité de la logique devra être limitée.

Nous présentons dans ce chapitre le μ-calcul régulier : la logique temporelle du model-checker EVALUATOR. Nous mettons l'accent par la suite sur la description et la vérification des propriétés de bon fonctionnement de notre système.

4.1 Mu-calcul régulier

Le μ-calcul régulier [66] est la logique temporelle utilisée dans EVALUATOR, elle fait partie des logiques temporelles avec opérateurs de point fixe, connues pour leur expressivité et utilisées pour la description de propriétés arborescentes sur les STEs.

Le μ-calcul régulier remplace les formules sur les actions de μ-calcul modal [56] avec les formules régulières de PDL [18]. Bien que cette extension n'augmente pas l'expressivité de la logique par rapport au μ-calcul modal standard, la description des propriétés en μ-calcul régulier est rendue plus simple et conviviale grâce à l'utilisation des opérateurs réguliers à la place des opérateurs de point fixe.

Nous présentons dans ce qui suit la syntaxe et la sémantique du μ-calcul régulier, pour cela nous considérons un STE M donné par la définition 4.1 décrivant tous les comportements possibles du système.

Définition 4.1 *Un système de transitions étiquetées est un quadruplet $M = (S, A, T, s_0)$, où :*

- *S est un ensemble fini d'états*
- *A est un ensemble fini d'actions*
- *$T \subseteq S \times A \times S$ est un ensemble de transitions. Une transition $(s_1, a, s_2) \in T$, notée également $s_1 \xrightarrow{a} s_2$, signifie que le système peut passer de l'état s_1 à l'état s_2 en exécutant l'action a.*
- *$s_0 \in S$ est l'état initial.*

Remarque 4.2 *Un système de transitions étiquetées peut être vu comme un graphe orienté, muni d'un état initial et dont les arcs sont étiquetés par des actions. Par conséquent, l'ensemble du vocabulaire défini pour les graphes reste valide, en particulier les notions usuelles de chemin et circuits.*

4.1.1 La syntaxe

Le μ-calcul régulier contient trois types d'entités : les formules sur actions (notées α), les formules régulières (notées β) et les formules sur états (notées φ).

• Les formules sur actions sont construites au-dessus du vocabulaire des actions $a \in A$ au moyen des opérateurs booléens standard.

$$\alpha ::= a \mid \neg\alpha_1 \mid \alpha_1 \wedge \alpha_2$$

Les opérateurs booléens dérivés sont définis comme suit :

$$F = a \wedge \neg a, \quad T = \neg F, \quad \alpha_1 \vee \alpha_2 = \neg(\neg\alpha_1 \wedge \neg\alpha_2)$$

• Les formules régulières sont construites à partir des formules sur actions et des opérateurs des expressions régulières standard, comme la concaténation (.), le choix (|) et la fermeture transitive et réflexive (*). L'opérateur de séquence vide (ε) et l'opérateur de répétition une ou plusieurs fois (β^+) sont définis respectivement par : $\varepsilon = F^*$ et $\beta^+ = \beta.\beta^*$.

$$\beta ::= \alpha \mid \beta_1.\beta_2 \mid \beta_1|\beta_2 \mid \beta^*$$

• Les formules sur états sont construites sur des variables propositionnelles $Y \in \mathrm{Y}$, au moyen des opérateurs booléens standard, des modalités de possibilité ($\langle\beta\rangle\varphi$) et de nécessité ($[\beta]\varphi$) et des opérateurs minimal ($\mu Y.\varphi$) et maximal ($\nu Y.\varphi$) de point fixe.

$$\varphi ::= \mathrm{F} \mid \mathrm{T} \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \langle\beta\rangle\varphi \mid [\beta]\varphi \mid Y \mid \mu Y.\varphi \mid \nu Y.\varphi$$

4.1.2 La sémantique

• La sémantique d'une formule α sur le STE M est définie par l'interprétation $\|\alpha\| \subseteq A$, qui dénote le sous-ensemble d'actions de A satisfaisant α. Les opérateurs booléens ont la sémantique habituelle définie en termes d'opérations ensemblistes.

$\|a\|$	$=$	$\{a\}$
$\|\neg\alpha\|$	$=$	$A \backslash \|\alpha\|$
$\|\alpha_1 \wedge \alpha_2\|$	$=$	$\|\alpha_1\| \cap \|\alpha_2\|$

• La sémantique d'une formule β est définie par l'interprétation $\|\beta\| \subset S \times S$, qui dénote les couples d'états qui sont aux extrémités des séquences de transitions satisfaisant β. Les formules régulières atomiques α dénotent les séquences comportant une seule transition étiquetée par une action satisfaisant α. Les formules $\beta_1.\beta_2$, $\beta_1|\beta_2$, et β^* dénotent respectivement la concaténation, le choix et la répétition (zéro ou plusieurs fois) de séquences de transitions satisfaisant β_1, β_2 et β ($\circ$, $\cup$ et * dénotent respectivement la composition, l'union et la fermeture transitive et réflexive d'une relation).

$\|\alpha\|$	$=$	$\{(s_1, s_2) \in S \times S \mid \exists a \in \|\alpha\|.s_1 \xrightarrow{a} s_2\}$
$\|\beta_1.\beta_2\|$	$=$	$\|\beta_1\| \circ \|\beta_2\|$
$\|\beta_1\|\beta_2\|$	$=$	$\|\beta_1\| \cup \|\beta_2\|$
$\|\beta^*\|$	$=$	$\|\beta\|^*$

• La sémantique d'une formule φ est définie par l'interprétation $\|\varphi\|\rho \subseteq S$ qui dénote un sous ensemble d'états de S, avec $\rho : Y \to 2^S$ un environnement associant des sous-ensembles d'états à toutes les variables propositionnelles libres dans φ ($\emptyset$ dénote que : $(\rho_1 \emptyset \rho_2)(Y)$ est égal à $\rho_2(Y)$ si ρ_2 affecte Y et $\rho_1(Y)$ sinon). La modalité $\langle\beta\rangle\varphi$ dénote les états ayant au moins une séquence de transitions successeur satisfaisant β et menant à un état satisfaisant φ La modalité $[\beta]\varphi$ dénote les états dont toutes les séquences de transitions successeurs satisfaisant β mènent à des états satisfaisant φ. Les deux opérateurs modaux sont duaux : $[\beta]\varphi = \neg\langle\beta\rangle\neg\varphi$, et monotones : si $\varphi_1 \Rightarrow \varphi_2$ alors $\langle\beta\rangle\varphi_1 \Rightarrow \langle\beta\rangle\varphi_2$ et $[\beta]\varphi_1 \Rightarrow [\beta]\varphi_2$. Les opérateurs $\mu Y.\varphi$ et $\nu Y.\varphi$ dénotent respectivement la plus petite et la plus grande solution de l'équation $Y = \varphi$ dans un environnement ρ.

$$
\begin{array}{lcl}
\|F\|\rho & = & \emptyset \\
\|T\|\rho & = & S \\
\|\varphi_1 \wedge \varphi_2\|\rho & = & \|\varphi_1\|\rho \cap \|\varphi_2\|\rho \\
\|\varphi_1 \vee \varphi_2\|\rho & = & \|\varphi_1\|\rho \cup \|\varphi_2\|\rho \\
\|\langle\beta\rangle\varphi\|\rho & = & \{s_1 \in S \mid \exists s_2 \in S.(s_1, s_2) \in \|\beta\| \wedge s_2 \in \|\varphi\|\rho\} \\
\|[\beta]\varphi\|\rho & = & \{s_1 \in S \mid \forall s_2 \in S.(s_1, s_2) \in \|\beta\| \Rightarrow s_2 \in \|\varphi\|\rho\} \\
\|Y\|\rho & = & \rho(Y) \\
\|\mu Y.\varphi\|\rho & = & \bigcap\{S' \subseteq S \mid \Phi_\rho(S') \subseteq S'\} \\
\|\nu Y.\varphi\|\rho & = & \bigcup\{S' \subseteq S \mid S' \subseteq \Phi_\rho(S')\} \\
 & & Avec : \Phi_\rho : 2^S \rightarrow 2^S, \Phi_\rho(S') = \|\varphi\|(\rho\ \emptyset[S'/Y])\}
\end{array}
$$

4.2 Spécification des propriétés

La validation du modèle LOTOS de notre système se traduit par la vérification de la bonne modélisation des aspects matériels et logiciels :

- La vérification des aspects matériels concerne le protocole de cohérence de caches (les états des caches, le type de transfert) et la latence d'accès en fonction des niveaux des transfert dans la topologie considérée.
- La vérification des aspects logiciels concerne la correction de l'exécution de l'algorithme MPI : la gestion des verrous, les contraintes d'accès aux variables (partagées et privées) et l'ordonnancement des accès aux variables.

Un rendez-vous sur la porte ACTION modélise un accès en lecture, en écriture ou en lecture-écriture atomique à une variable de l'algorithme MPI :

```
ACTION  ? P :ID_Processor (* Identificateur du processus demandeur*)
        ? adr :Address (* Nom de la variable *)
        ? op :ID_Action (* Type d'accès : Load, Store, ...*)
        ? val :Memory_Value (* La valeur lue ou écrite *)
        ? latency :Latency_Value ; (* La latence de l'accès *)
```

Les informations décrites sur les offres de la porte ACTION s'avèrent suffisantes pour la réalisation d'un accès et l'évaluation de sa latence, en revanche nous n'avons aucune information sur les nouveaux états des caches et sur le transfert (type et niveau) pour s'assurer que l'accès respecte bien les règles du protocole de cohérence de caches et que la latence a été correctement évaluée dans la topologie considérée. Par conséquent, la spécification de ces informations en LOTOS est jugée indispensable pour la vérification de la correction de la modélisation des aspects matériels.

Nous présentons dans ce qui suit les nouveaux types algébriques en LOTOS que nous avons modélisés pour mener à bien la vérification des

aspects matériels. Il s'agit d'un type spécifiant les états d'une ligne de caches, un type spécifiant les différents types possibles d'un transfert et un type spécifiant les différents niveaux possibles d'un transfert.

• **Les états des lignes de caches** sont modélisés en LOTOS par le type **Cache_Line**, implémenté en C par ADT_CACHE_LINE. Suivant le principe de modélisation des caches que nous avons présenté dans la section 3.1, une ligne de cache dénote l'état d'une variable dans le cache de tous les processeurs du système.

```
type Cache_Line is Cache, Address
 sorts Cache_Line   (*! implementedby ADT_CACHE_LINE
                        printedby ADT_PRINT_CACHE_LINE external *)
 opns
   State_Cache_Line (*! implementedby ADT_STATE_CACHE_LINE external *)
                        : Cache, Address -> Cache_Line
endtype
```

L'opération *State_Cache_Line* prend en entrée les caches et l'indice d'une variable et retourne en sortie son état dans le cache de tous les processeurs du système. Par exemple, l'état de ligne de cache de la variable *count* de l'algorithme barrière *centralized* entre 3 processus, après initialisations de caches et avant tout accès est donné par : (P0, I)(P1, I)(P2, I).

Remarque 4.3 *Pour des raisons de précision et de clarté, nous avons préféré la définition du nouveau type **Cache_Line** à l'exploitation du type **Cache**, pour les raisons suivantes :*

1. *La taille de la structure caches est fixe (dépendant du nombre de variables et de processus), par conséquent on ne peut pas définir une opération dans le type **Cache** qui retourne une ligne de caches dans un type **Cache**.*
2. *La fonction d'impression du type **Cache** (générée par le compilateur CAESAR ou définie par l'utilisateur) permet l'affichage du contenu de toute la structure cache, en effet, c'est une fonction non-paramétrable.*

• **Le type du transfert** est modélisé par le type **Transfer_Type** en LOTOS. Il spécifie l'endroit où la donnée est chargée : *Internal*, *Memory*, *Cache* et *MemoryInv*.

```
type Transfer_Type is Natural
 sorts Transfer_Type
 opns
   Internal   (*! constructor *),
   Cache   (*! constructor *),
   Memory   (*! constructor *),
   MemoryInv   (*! constructor *) : -> Transfer_Type
endtype
```

• **Le niveau de transfert** est modélisé par le type **Transfer_Level** en LOTOS. Il spécifie les 4 niveaux des structures hiérarchiques des architectures FAME et MESCA : *Internal*, *Intra_Node*, *Inter_Node* et *Inter_Module*.

```
type Transfer_Level is Natural
 sorts Transfer_Level
 opns
   Internal   (*! constructor *),
   Intra_Node   (*! constructor *),
   Inter_Node   (*! constructor *),
   Inter_Module   (*! constructor *) : -> Transfer_Level
endtype
```

• **Les opérations d'évaluation** des paramètres d'un accès : le type de transfert, le niveau de transfert, l'identificateur du fournisseur de la donnée (cache ou mémoire), le niveau de consultation et/ou invalidation et la latence de l'accès, ont été regroupés dans un seul type **Evaluation**.

```
type Evaluation is Memory, Cache, ID_Action, ID_Processor, Address,
                   Transfer_Type, Transfer_Level, Latency_Value
 sorts Evaluation (*! implementedby ADT_EVALUATION external *)
 opns
   Evaluation_Latency
         (*! implementedby ADT_EVALUATION_LATENCY external *)
            : Memory, Cache, ID_Action, Address, ID_Processor -> Latency_Value
   Evaluation_Transfer_Type
         (*! implementedby ADT_EVALUATION_TRANSFER_TYPE external *)
            : Memory, Cache, ID_Action, Address, ID_Processor -> Transfer_Type
   Evaluation_ID_Provider
         (*! implementedby ADT_EVALUATION_ID_PROVIDER external *)
            : Memory, Cache, ID_Action, Address, ID_Processor -> ID_Processor
   Evaluation_Transfer_Level
         (*! implementedby ADT_EVALUATION_TRANSFER_LEVEL external *)
            : Memory, Cache, ID_Action, Address, ID_Processor -> Transfer_Level
   Evaluation_Inv_Level
         (*! implementedby ADT_EVALUATION_INV_LEVEL external *)
            : Memory, Cache, ID_Action, Address, ID_Processor -> Transfer_Level
endtype
```

Question ? *Comment intégrer ces nouveaux types dans la spécification LOTOS ?*

Nous avons analysé deux solutions possibles :

Solution 1 : définition de nouvelles offres sur la porte ACTION.

Solution 2 : définition d'une nouvelle porte VERIF dans le processus TRANSFER dont le rendez-vous a lieu après une synchronisation sur la porte ACTION.

La première solution révèle une certaine lourdeur dans sa mise en œuvre, en dépit de la taille de l'espace d'états préservé (à la différence

des transitions générées, dont les étiquettes augmentent en taille) : d'une part, il faut ajouter les offres sur tous les rendez-vous de la porte ACTION dans tous les processus LOTOS participant aux synchronisations sur cette porte et, d'autre part, ces informations ne sont utiles que dans la phase de vérification. Nous avons dès lors opté pour la seconde solution et pour plus de généricité, nous avons défini un nouveau processus LOTOS TRANSFER_VERIF accomplissant les mêmes fonctionnalités du processus TRANSFER mais avec la précision de tous les paramètres d'accès sur la porte VERIF. Dans la spécification LOTOS nous avons le choix entre l'utilisation du processus TRANSFER pour une étude des performances ou TRANSFER_VERIF pour la vérification.

```
process   TRANSFER_VERIF [ACTION, VERIF] (M :Memory, C :Cache) : noexit :=
 ACTION   ? P :ID_Processor   ? adr :Address   ? op :ID_Action
          ? val :Memory_Value   ? latency :Latency_Value
  [
   (((op == Load) and (val == Load_Memory(M, adr))) or
    ((op == Fetch_and_decrement) and
     (val == Load_Memory(UpDate_Memory(M, adr, op, val), adr))) or
    (op == Store)
   ) and
   (latency == Evaluation_Latency(M, C, op, adr, p))
  ];
 VERIF   ! p of ID_Processor                                              (*1*)
       ! op of ID_Action                                                  (*2*)
       ! adr of Address                                                   (*3*)
       ! val of Memory_Value                                              (*4*)
       ! State_Cache_Line(C,adr) of Cache_Line                            (*5*)
       ! State_Cache_Line(UpDate_Cache(C,op,adr,p),adr) of Cache_Line     (*6*)
       ! Evaluation_Transfer_Type(M,C,op,adr,p) of Transfer_Type          (*7*)
       ! Evaluation_ID_Provider(M,C,op,adr,p) of ID_Processor             (*8*)
       ! Evaluation_Transfer_Level(M,C,op,adr,p) of Transfer_Level        (*9*)
       ! Evaluation_Invalidation_Level(M,C,op,adr,p) of Transfer_Level   (*10*)
       ! latency of Latency_Value;                                        (*11*)

  TRANSFER_VERIF [ACTION, VERIF] (UpDate_Memory(M, adr, op, val),
                                  UpDate_Cache(C, op, adr, P))
endproc
```

Le processus TRANSFER_VERIF dénote un comportement de fonctionnalité *noexit*. Il est paramétré par deux portes de communication : ACTION et VERIF et par variables, une de type **Memory** et une autre de type **Cache**. Le rendez-vous sur la porte ACTION est identique à celui dans le processus TRANSFER. Le rendez-vous sur la porte VERIF reprend les informations obtenues dans le rendez-vous sur ACTION dans les offres 1, 2, 3, 4 et 11 et, à l'aide des fonctions d'évaluation des paramètres d'un accès, il spécifie les informations concernant : l'état de la ligne des caches de la variable *adr* avant (*offre 5*) et après (*offre 6*) l'accès, le type de transfert (*offre 7*), l'identificateur du processeur fournisseur de la donnée (*offre 8*), le niveau de transfert (*offre 9*) et le niveau d'invalidation (*offre 10*).

4.2.1 Définition des prédicats de base

EVALUATOR permet la définition et l'utilisation de macros pour spécifier des opérateurs temporels paramétrés par des formules sur actions et / ou sur états, ce qui offre la possibilité de construire des bibliothèques réutilisables.

Nous avons défini une bibliothèque "*macros.mcl*" contenant des définitions de prédicats de base sur les transitions du STE. En effet, comme toutes les informations dont nous avons besoin pour la vérification sont fournies par les rendez-vous sur la porte VERIF, nous avons défini une macro de *base* nommée *Action_Verif*, paramétrée par les différentes offres décrites dans le rendez-vous sur VERIF.

Ci-dessous la définition de la macro *Action_Verif* pour un nombre de processus égal à 2.

```
macro Action_Verif ( id_proc, action, adr, id_proc_adr, val, state_after_0, state_after_1,
                     state_before_0, state_before_1, transfer_type, id_provider,
                     transfer_level, inv_level, latency) =

  'VERIF  ! P' # id_proc #
          '!' # action #
          '!' # adr #"# id_proc_adr #
          '!' # val #
          '!(P0, ' # state_after_0 # ')(P1, ' # state_after_1 # ')'
          '!(P0,' # state_before_0 # ')(P1,' # state_before_1 # ')'
          '!' # transfer_type # '_TRANSFER'
          '!' # id_provider #
          '!' # transfer_level #
          '!' # inv_level #
          '!' # latency #
end_macro
```

Les chaînes de caractères entourées de guillemets simples (') dénotent des expressions régulières, qui sont concaténées au moyen de l'opérateur '#' afin de former une expression régulière filtrant l'action complète effectuée sur la porte VERIF.

La bibliothèque *macro.mcl* est incluse en tête de chaque fichier constituant des propriétés à vérifier. Les définition d'autres prédicats de base de cette bibliothèque sont disponibles dans l'annexe A.2.

Remarque 4.4 *La définition de la macro* Action_Verif *varie en fonction du nombre de processus participant à l'exécution de l'algorithme MPI en raison des offres indiquant l'état d'une ligne de cache.*

4.2.2 Vérification des aspects matériels

Le but de la vérification des aspects matériels est de s'assurer de la correction du fonctionnement du protocole de cohérence de caches et de la bonne évaluation des paramètres d'un accès (type de transfert, niveau de transfert, latence, ...). Ceci permettra par la suite une analyse fiable des performances.

Nous avons vérifié les aspects matériels par des propriétés de sûreté (*safety properties*) [58], qui énoncent que "*quelque chose de mauvais ne se produira jamais*". Ces propriétés peuvent être exprimées de manière concise en μ-calcul régulier au moyen de modalités "[F] R", où la formule régulière R représente les séquences d'actions indésirables qui doivent être absentes dans le modèle.

La propriété 4.1 spécifie les règles de changement des états des caches et les types de transfert engendrés dans le cas du protocole de cohérence de caches *A* (voir la définition de la macro *CC_Load* dans l'annexe A.2).

Propriété 4.1 *S'il existe un accès dont les changements des états de caches et le type de transfert ne vérifient pas l'une des règles : RL1, RL3, RL4 ou RL6, alors il y a une erreur dans la modélisation du protocole de cohérence de caches A (voir la définition 1.4).*

```
[
  true *.
  (
       (*RL1 : LOAD(I,I)->(E,I)*)
    not (CC_Load('I', 'I', 'E', 'I', 'memory', '.*') or CC_Load('I', 'I', 'I', 'E', 'memory', '.*'))
    and (*RL3 : LOAD(I,E)->(S,S) *)
    not (CC_Load('I', 'E', 'S', 'S','memoryInv','.*') or CC_Load('E', 'I', 'S','S','memoryInv', '.*'))
    and (*RL4 : LOAD(I,M)->(E,I) *)
    not (CC_Load('I', 'M', 'E', 'I', 'cache', '.*') or CC_Load('M', 'I', 'I', 'E', 'cache', '.*'))
    and (*RL6 : LOAD(S,S)->(S,S) *)
    not CC_Load('S', 'S', 'S', 'S', 'internal', '.*')
    and (*RL6 : LOAD(E,I)->(E,I) *)
    not (CC_Load('E', 'I', 'E', 'I', 'internal', '.*') or CC_Load('I', 'E', 'I', 'E', 'internal','.*'))
    and (*RL6 : LOAD(M,I)->(M,I) *)
    not (CC_Load('M', 'I', 'M', 'I','internal', '.*') or CC_Load('I', 'M', 'I', 'M', 'internal', '.*'))
  )
] false
```

Dans le tableau 4.1 nous définissons les règles de cohérence entre les différents paramètres d'un accès pour deux processeurs dans une $topology_0$.

Règle	Demandeur	Type transfert	Fournisseur	Niveau transfert	Niveau Invalidation	Latence
RA1	P_0	*Internal*	P_0	*Internal*	Null	C_INT
RA2	P_1	*Internal*	P_1	*Internal*	Null	C_INT
RA3	P_0	*Cache*	P_1	*Intra_Node*	Null	C_FSB
RA4	P_1	*Cache*	P_0	*Intra_Node*	Null	C_FSB
RA5	P_0 / P_1	*Memory*	Memory	*Intra_Node*	Null	M_FSB_1
RA6	P_0	*MemoryInv*	P_1	*Intra_Node*	*Intra_Node*	M_FSB_1
RA7	P_1	*MemoryInv*	P_0	*Intra_Node*	*Intra_Node*	M_FSB_1

TAB. 4.1 – *Les règles de cohérence entre les paramètres d'un accès*

A partir du tableau 4.1, nous définissons la propriété 4.2 pour la vérification de l'évaluation correcte des paramètres d'un accès (voir la définition de la macro *Access* dans A.2).

Propriété 4.2 *S'il existe un accès dont les paramètres ne vérifient pas l'une des règles de cohérence : RA1, RA2, RA3, RA4, RA5, RA6 ou RA7, alors il y a une erreur dans les procédures d'évaluation.*

```
[
 true *.
 (
      (*RA1*)
  not Access('0','INTERNAL','0','INTERNAL_LEVEL','NIL_LEVEL','C_INT')
  and (*RA2*)
  not Access('1','INTERNAL','1','INTERNAL_LEVEL','NIL_LEVEL','C_INT')
  and (*RA3*)
  not Access('0','CACHE','1','INTRA_NODE_LEVEL','NIL_LEVEL','C_FSB')
  and (*RA4*)
  not Access('1','CACHE','0','INTRA_NODE_LEVEL','NIL_LEVEL','C_FSB')
  and (*RA5*)
  not Access('.*','MEMORY','.*','INTRA_NODE_LEVEL','NIL_LEVEL','M_FSB_1')
  and (*RA6*)
  not Access('0','MEMORYINV','1','INTRA_NODE_LEVEL','INTRA_NODE_LEVEL', ''M_FSB_1')
  and (*RA7*)
  not Access('1','MEMORYINV','0','INTRA_NODE_LEVEL','INTRA_NODE_LEVEL', ''M_FSB_1')
 )
] false
```

4.2.3 Vérification des aspects logiciels

La vérification des aspects logiciels dépend fortement de l'algorithme MPI. Nous décrivons dans ce qui suit les propriétés que nous avons spécifiées pour vérifier le comportement de l'algorithme *ping-pong* et des algorithmes de *barrières*.

Algorithme *ping-pong*

Les propriétés de correction pour l'algorithme de *ping-pong* concernant principalement la gestion de verrous.

Il s'agit de vérifier l'exclusion mutuelle des accès aux variables partagées par les deux processus : la prise d'un verrou ne peut pas être effectuée par les deux processus en même temps et un processus ne peut pas libérer un verrou qu'il n'a pas pris auparavant. Nous avons exprimé cela par deux propriétés 4.3 et 4.4 pour la variable verrou *available_lock* (voir la définition des macros *Take_Lock* et *Free_Lock* dans A.2).

Propriété 4.3 *Si un processus P_i prend le verrou adr, et si avant qu'il le libère, il le reprend encore une fois, ou bien si un autre processus P_j réussit à l'obtenir, alors il y a une erreur dans la gestion des verrous.*

```
macro MUTEX(i,adr,id_proc_adr,j) =
(
   Take_Lock(i, adr, id_proc_adr).
   (not (Free_Lock(i, adr, id_proc_adr)))*.
   (Take_Lock(i, adr, id_proc_adr) or Take_Lock(j, adr, id_proc_adr))
)
end_macro
[
  true *.
  (
    MUTEX('0','AVAILABLE_LOCK','0','1') | MUTEX('0','AVAILABLE_LOCK','1','1') |
    MUTEX('1','AVAILABLE_LOCK','0','0') | MUTEX('1','AVAILABLE_LOCK','1','0')
  )
] false
```

Propriété 4.4 *Un processus P_i ne peut pas libérer un verrou tant qu'il ne l'a pas pris auparavant.*

```
macro Lock_UnLock_Lock (i,adr,id_proc_adr)=
   Free_Lock(i, adr, id_proc_adr).
   ( not Take_Lock(i, adr, id_proc_adr))*.
   Free_Lock(i, adr, id_proc_adr)
end_macro
macro Lock_Unlock (i,adr,id_proc_adr)=
   ( not Take_Lock(i, adr, id_proc_adr))*.
   Free_Lock(i, adr, id_proc_adr)
end_macro
```

```
[
 (
  Lock_Unlock ('0','AVAILABLE_LOCK','0') | Lock_Unlock ('0','AVAILABLE_LOCK','1') |
  Lock_Unlock ('1','AVAILABLE_LOCK','0') | Lock_Unlock ('1','AVAILABLE_LOCK','1')
 ) | (
  true *.
  (
   Lock_UnLock_Lock ('0','AVAILABLE_LOCK','0') | Lock_UnLock_Lock ('0','AVAILABLE_LOCK','1') |
   Lock_UnLock_Lock ('1','AVAILABLE_LOCK','0') | Lock_UnLock_Lock ('1','AVAILABLE_LOCK','1')
  )
 )
] false
```

Nous avons également spécifié la propriété 4.5 pour vérifier la conformité des accès aux variables privées : le processus P_0 ne doit pas accéder aux variables privées de P_1 et inversement (voir la définition des macros *Private_Access* dans A.2).

Propriété 4.5 *Protection des accès aux variables privées.*

```
[
  true * .
  (
       Private_Access ('0','LOCAL_AVAILABLE_PTR','1')
    or Private_Access ('1','LOCAL_AVAILABLE_PTR','0')
    or Private_Access ('0','FREE_HEAD_PTR','1')
    or Private_Access ('1','FREE_HEAD_PTR','0')
    or Private_Access ('0','FREE_TAIL_PTR','1')
    or Private_Access ('1','FREE_TAIL_PTR','0')
    or Private_Access ('1','PKT_TMP','0')
    or Private_Access ('0','PKT_TMP','1')
  )
  ] false
```

La propriété 4.6 spécifie l'ordonnancement des événements dans le comportement de *ping-pong* :

- Quand un processus P_i envoie un paquet, le processus P_j ($i \neq j$) le reçoit.
- Quand un processus P_i reçoit un paquet, le processus P_j ($i \neq j$) l'a déjà envoyé.

Propriété 4.6 *Pour un processus donné, un envoi est toujours suivi d'une réception, et une réception est toujours précédée d'un envoi.*

```
[
   true * .
   (
     ( Send('0','1').(not Receive('0'))*.Send('0','1') ) |
     ( Send('1','0').(not Receive('1'))*.Send('1','0') ) |
     ( Receive('1').(not Send('1','0'))*.Receive('1') ) |
     ( Receive('0').(not Send('0','1'))*.Receive('0') )
   )
   ] false
```

Dans le code de la primitive *send*, l'envoi d'un paquet de P_i à P_j débute par la prise du verrou *Incoming_Lock(P_j)* par P_i, afin d'ajouter le paquet dans la liste des paquets reçus *Incoming_List* de P_j. La réception d'un paquet de P_i envoyé par P_j est spécifiée dans la primitive *receive* 3 par la prise de verrou *Incoming_Lock(P_j)* par P_i, afin de consommer le paquet dans sa liste des paquets reçus *Incoming_List*.

Les macros *Send* et *Receive* ci-dessous spécifient la prise de verrou *Incoming_Lock* pour l'envoi et la réception d'un paquet.

```
macro Send(i,j)=
 Action_Verif (i, 'TEST_AND_SET', 'INCOMING_LOCK', j, 'UNLOCK', '.*', '.*' , '.*', '.*', '.*',
                  '.*' , '.*', '.*', '.*')
end_macro

macro Receive(i)=
 Action_Verif (i, 'TEST_AND_SET', 'INCOMING_LOCK', i, 'UNLOCK', '.*', '.*' , '.*', '.*', '.*',
                  '.*' , '.*', '.*', '.*')
end_macro
```

Algorithme *barrière*

Le comportement d'un algorithme de *barrière* spécifie une suite infinie de synchronisations entre plusieurs processus en utilisant une primitive de barrière. Par conséquent, un comportement d'un algorithme de *barrière* est jugé correct si et seulement si :

> *tous les processus participant à l'exécution de la barrière doivent participer au même cycle de barrière : un processus ne peut pas entamer un nouveau cycle tant que les autres processus n'ont pas quitté l'ancien cycle*

Tous les algorithmes barrière que nous avons modélisés (*centralized*, *tournament* et *combining tree*) possèdent une variable booléenne nommée *sense*, considérée comme un pseudo-identifiant du cycle de la barrière en cours. En effet, à l'arrivée à la barrière, tous les processus doivent avoir la même valeur de *sense* et à la sortie ils inversent son contenu de telle sorte que deux cycles consécutifs de barrière ne doivent pas avoir la même valeur de *sense*. Ainsi, s'il existe un processus possédant une valeur différente pour *sense* que les autres processus, cela signifie qu'il n'a pas encore quitté l'ancien cycle de barrière et que les autres processus ont entamé un nouveau cycle ou inversement, dans les deux cas il s'agit d'une erreur fatale dans l'exécution de la barrière.

La spécification des propriétés exprimant la correction du comportement de l'algorithme de *barrière* nécessite la prise en compte des identificateurs des processus participant à son exécution. Pour cela nous avons utilisé le langage MCL [67], qui est une extension de μ-calcul régulier avec des variables typées et des mécanismes de manipulation de données. Ce langage est accepté en entrée par l'outil EVALUATOR 4.0 que nous avons utilisé pour vérifier les propriétés 4.7 et 4.8 sur tous les processus participant à la barrière.

L'étiquette "STORE ! i ! b" est obtenu par le renommage de "ACTION ! Pi ! sense(Pi) ! Store ! b ! latency".

Propriété 4.7 *Un processus P_i ne peut effectuer deux cycles de barrières consécutifs avec la même valeur booléenne de la variable* sense.

```
[
  true * .
   { STORE ? i :nat  ? b :bool } .
   ( not { STORE ! i ! not (b) })* .
   { STORE ! i ! b }
] false
```

Propriété 4.8 *Durant chaque cycle, tous les processus participant à la barrière doivent avoir la même valeur booléenne de* sense.

```
(* N, le nombre de processus participant à la barrière  *)
macro N () = 4 end_macro

(* Tous les processus P_{j≠i} doivent exécuter *)
(* la barrière avec la valeur not(b) de  sense *)
macro Inev_Barrier_Pass (i, b) =
 forall j : nat among { 0 ... N - 1 } . (
 (j <> i) implies
   mu X . (
    [ { STORE ! j ! b } ] false
    and
    (< true > true and [ not { STORE ! j ! not (b) } ] X)
   )
 )
end_macro
(* Le premier cycle de la barrière avec la valeur false de sense*)
[ ( not { STORE ? any  ? any  })* .
 { STORE ? i :nat  ! false }
] Inev_Barrier_Pass (i, true)
and
(* Un cycle intermédiaire *)
[ true * .
 { STORE ? any ? b :bool } .
 (not { STORE ? any  ? any })* .
 { STORE ? i :nat ! not (b) }
] Inev_Barrier_Pass (i, b)
```

Conclusion du chapitre

Dans ce chapitre, nous avons vérifié que les aspects logiciels et matériels de notre système ont été correctement modélisés en utilisant la méthode du *model-checking*.

Nous avons spécifier les propriétés du bon fonctionnement en utilisant la logique temporelle de l'outil EVALUATOR, le μ-calcul régulier. C'est une logique temporelle qui fait partie des logiques temporelles avec opérateurs de point fixe, connues pour leur expressivité et utilisées pour la description de propriétés arborescentes sur les STEs.

La vérification des aspects matériels qui concerne le protocole de cohérence de caches (les états des caches, le type de transfert) et la latence d'accès en fonction des niveaux des transfert dans la topologie considérée, a nécessité la définition de nouveaux types algébriques en LOTOS, à savoir un type spécifiant les états d'une ligne de caches, un type spécifiant les différents types possibles d'un transfert et un type spécifiant les différents niveaux possibles d'un transfert.

La vérification des aspects logiciels dépend fortement de l'algorithme MPI. Dans le cas de l'algorithme *ping-pong* nous avons vérifier principalement la bonne gestion des verrous et le bon ordonnancement des événements en termes d'envoi et de réception de paquets. Quant aux algorithmes de barrière nous avons vérifier que tous les processus participant à l'exécution de la barrière doivent participer au même cycle de barrière : un processus ne peut pas entamer un nouveau cycle tant que les autres processus n'ont pas quitté l'ancien cycle.

Évaluation des performances par analyse numérique

5

Sommaire

Par ce chapitre nous arrivons au cœur du sujet de la thèse : l'évaluation des performances.

Par souci de clarté et de précision, nous allons d'abord mettre au point les étapes de génération de CTMC à partir d'une spécification LOTOS augmentée d'informations stochastiques.

Nous discutons dans la seconde section, qui concerne l'analyse markovienne, les différentes approches que nous avons étudiées pour l'insertion des taux markoviens dans les spécifications LOTOS des algorithmes MPI et la possibilité de génération de CTMC, ainsi que l'évaluation des indices de performances par l'outil BCG_STEADY de CADP.

Nous achevons ce chapitre par la présentation des résultats de l'étude stochastique sur les trois algorithmes que nous avons présentés dans le chapitre 3, il s'agit du *ping-pong*, de la barrière *tournament* et de la barrière *combining tree*.

Les aspects temporels que nous avons pris en compte dans l'étude des performances de notre système peuvent être répartis en deux catégories : les *latences d'accès* et les *délais d'attente*.

1. La latence d'accès correspond à la durée du transfert de la donnée entre la source (mémoire ou cache) et la destination (cache), suite à une demande d'accès en lecture ou en écriture.
2. Le délai d'attente correspond généralement à la durée d'un traitement autre qu'un accès. Par exemple, l'instruction *Wait()* dans la primitive *receive*.

5.1 Génération de la CTMC

Les taux markoviens dans une spécification LOTOS correspondent à des rendez-vous sur des portes de communication que nous avons nommées *portes stochastiques* (voir la section 2.2.3).

Afin d'éviter la lourdeur de la manipulation de plusieurs portes stochastiques, nous avons opté pour la déclaration d'une seule porte, appelée LATENCY pour modéliser les différents aspects temporels de notre système. La modélisation des aspects temporels est faite de la manière suivante :

1. *Modélisation d'une latence d'un accès :*

   LATENCY ! Pi **of** ID_Processor ! var **of** Address ! latency **of** Latency_Value ;

   où *latency* est la latence d'accès du processus P_i à la variable *var*.

2. *Modélisation d'un délai d'attente :*

   LATENCY ! Pi **of** ID_Processor ! wait **of** Nat ;

   où *wait* est l'identificateur d'un délai d'attente. Par exemple, $wait = 1$ pour l'instruction *Wait(1)*.

Dans le cas de la latence d'accès, on précise dans le rendez-vous sur la porte LATENCY l'identificateur du processus réalisant l'accès, le nom de la variable et la latence de l'accès, afin de pouvoir effectuer des statistiques sur les accès. Par exemple, évaluer (en moyenne) le nombre de *miss*[1] réalisé par variable, par processus ou par variable et par processus.

Dans le cas d'un délai d'attente, le rendez-vous sur la porte LAMBDA comporte deux offres, la première pour l'identificateur du processeur et la seconde est un entier naturel identifiant le délai d'attente.

Le taux markovien d'une latence d'un accès ou un délai d'attente correspond à son inverse :

Si $latency > 0$ est une latence d'un accès, alors $\lambda_{latency} = \frac{1}{latency}$ est son taux markovien.

Si $wait > 0$ est un délai d'attente, alors $\lambda_{wait} = \frac{1}{wait}$ est son taux markovien.

Par exemple le taux markovien d'une latence d'accès en cache C_FSB est donné par $\lambda_{C\_FSB} = \frac{1}{\text{C\_FSB}}$.

Lors de la génération de l'IMC, les actions stochastiques sont transformées en taux markoviens de la forme "$prefixe$; *rate* λ", par l'application du renommage (**rename in**), où *préfixe* est une suite de caractères et $\lambda > 0$ est la valeur du taux markovien .

Exemple 5.1 *L'action stochastique suivante décrit la latence d'un accès du processus P_0 à la variable* var *située en sa mémoire locale*

LATENCY ! Po **of** ID_Processor ! var **of** Address ! M_FSB_1 **of** Latency_Value ;

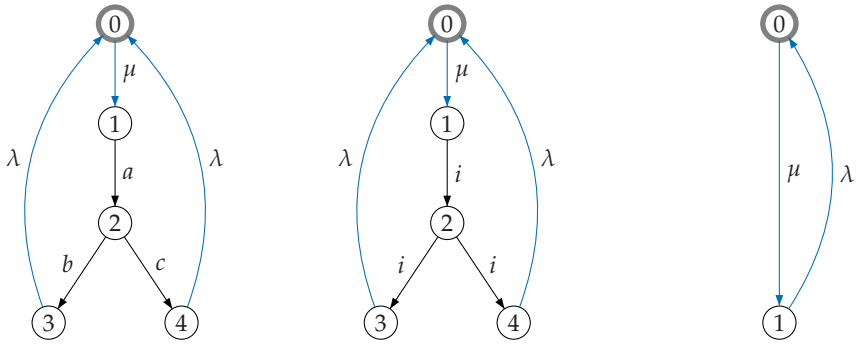
On peut produire plusieurs transitions stochastiques représentant le même taux markovien mais avec des préfixes différents :

[1] miss : la donnée est invalide dans le cache du processus demandeur

- *Un taux markovien avec un préfixe vide : "rate* $\lambda_{M\_FSB\_1}$*".*
- *Un taux markovien d'un accès réalisé par le processus* P_0 *: "P0; rate* $\lambda_{M\_FSB\_1}$*".*
- *Un taux markovien d'un accès à la variable* var *: "var; rate* $\lambda_{M\_FSB\_1}$*".*
- *Un taux markovien d'un accès à la variable* var *réalisé par le processus* P_0 *: "P0_var; rate* $\lambda_{M\_FSB\_1}$*".*

Les IMC représentent un sur-ensemble strict des chaînes de Markov. Il existe en effet une infinité de IMC qui n'appartiennent pas à la classe des chaînes de Markov. La raison est que l'un des concepts de base des IMC, "le potentiel d'interaction avec un environnement", n'est pas exprimable avec les chaînes de Markov. Par conséquent, dans le but d'associer une chaîne de Markov avec une spécification IMC, il est toujours nécessaire que la spécification ne traite pas des interactions avec l'environnement. Pour ce fait, nous procédons à l'abstraction de toutes les actions interactives au moyen de l'opérateur d'abstraction **"hide in"**. Ceci est indispensable pour la détermination de la chaîne de Markov représentant le comportement de la spécification.

L'abstraction des actions interactives s'avère une condition *nécessaire* mais *non suffisante* pour déterminer la chaîne de Markov, en raison du comportement *non-déterministe* qu'elle engendre sur les états comportant plusieurs transitions interactives sortantes. Par exemple, le comportement B de la figure 5.1 spécifie un comportement qui n'est pas complètement déterministe : au niveau de l'état 2 la décision entre la branche gauche est la branche droite est un choix *non-déterministe*.



A : IMC avant abstraction

B : abstraction des actions a et b dans A

C : réduction stochastique de B

FIG. 5.1 – *Exemple de IMC* non-déterministe

La présence du non-déterminisme entrave l'analyse stochastique de beaucoup d'IMC, mais d'autre part un large éventail d'IMC peuvent être transformées en chaînes de Markov par l'application de la réduction stochastique modulo la bisimulation faible (les axiomes cités dans 2.4) après l'abstraction de toutes les actions interactives. Le comportement C illustré dans la figure 5.1 est une CTMC résultant de la réduction stochastique du comportement B.

L'application de la réduction stochastique ne garantit pas souvent une élimination totale des états non-déterministes, il existe en effet des IMC

non-transformables en chaînes de Markov. Par exemple, notons λ_b et λ_c les taux markoviens des transitions qui suivent les actions interactives b et c respectivement dans le comportement A illustré dans la figure 5.1. La réduction stochastique permet l'élimination du non-déterminisme uniquement si $\lambda_b = \lambda_c = \lambda$ et dans ce cas la chaîne de Markov résultant de la réduction stochastique correspond au comportement C, sinon (si $\lambda_b \neq \lambda_c$) la réduction stochastique est incapable de supprimer le non-déterminisme.

Remarque 5.1 *Dans un processus stochastique, tout comportement possède, à n'importe quel point dans le temps, une probabilité unique, alors qu'en présence d'un comportement non-déterministe il est impossible de déterminer les probabilités.*

On note par $\text{SPEC}^{\lambda}_{ALGO}$ la spécification LOTOS de l'algorithme ALGO augmentée de rendez-vous sur la porte stochastique LAMBDA.

Les étapes de transformation de $\text{SPEC}^{\lambda}_{ALGO}$ en IMC puis en CTMC sont définies par l'utilisation des outils CADP, comme suit :

Étape 1 Génération du graphe $\text{SPEC}^{\lambda}_{ALGO}.bcg$ à partir de la spécification $\text{SPEC}^{\lambda}_{ALGO}.lotos$ en utilisant GENERATOR [2] :

"$\text{SPEC}^{\lambda}_{ALGO}.bcg$" = generation of "$\text{SPEC}^{\lambda}_{ALGO}.lotos$"

Étape 2 Le graphe $\text{IMC}_{ALGO}.bcg$ est obtenu par le renommage de toutes les transitions stochastiques figurant dans $\text{SPEC}^{\lambda}_{ALGO}.bcg$ en taux markoviens :

"$\text{IMC}_{ALGO}.bcg$" = total rename
"**LATENCY** ! P0 ! var ! M_FSB_1" → "P0 ; rate $\lambda_{M\_FSB\_1}$",
"**LATENCY** ! P0 ! 1 !" → "P0 ; rate λ_{WAIT_1}",
...
in "$\text{SPEC}^{\lambda}_{ALGO}.bcg$"

Étape 3 Finalement, la chaîne de Markov $\text{CTMC}_{ALGO}.bcg$ est obtenue après l'abstraction de toutes les transitions interactives à l'aide de l'outil BCG_LABELS[3] et l'élimination de toutes les transitions internes i par l'application de la réduction stochastique modulo la bisimulation faible par l'outil BCG_MIN :

"$\text{CTMC}_{ALGO}.bcg$" = branching stochastic reduction
of hide ACTION, WAIT
in "$\text{IMC}_{ALGO}.bcg$"

Il est possible également de générer la CTMC en une seule étape :

"$\text{CTMC}_{ALGO}.bcg$" = branching stochastic reduction
of hide ACTION, WAIT
in total rename
"**LATENCY** ! P0 ! var ! M_FSB_1 !" → "P0 ; rate $\lambda_{M\_FSB\_1}$",
"**LATENCY** ! P0 ! 1" → "P0 ; rate λ_{WAIT_1}",
...
in generation of "$\text{SPEC}^{\lambda}_{ALGO}.lotos$"

Ces étapes sont décrites dans un fichier *script SVL*, permettant ainsi une sorte d'automatisation de la génération de CTMC et l'obtention des mesures de performances.

[2] http://www.inrialpes.fr/vasy/cadp/man/generator.html
[3] http://www.inrialpes.fr/vasy/cadp/man/bcg_labels.html

Ces étapes décrivent une démarche de base pour l'obtention de la CTMC. Nous allons voir dans la section 6 que le problème de l'explosion de l'espace d'états a nécessité la définition d'autres démarches basées principalement sur la génération compositionnelle.

5.2 Analyse stochastique

5.2.1 Insertion des taux markoviens dans la spécification LOTOS

La définition d'une approche permettant l'insertion des aspects temporels aux *bons endroits* dans la spécification LOTOS est d'une très grande importance pour la *réussite* de l'analyse stochastique. En revanche, cela nécessite, non seulement une bonne connaissance du comportement du système, mais aussi bien une bonne compréhension de son modèle.

Le parallélisme et le protocole de cohérence de caches ont joué un rôle primordial dans l'évolution de la complexité du modèle et, par conséquent la difficulté de sa compréhension. Cela a rendu la définition d'une approche pour l'insertion des aspects temporels dans la spécification LOTOS, une mission *non-évidente* malgré le nombre raisonnable de possibilités.

Nous présentons dans cette section les différentes approches que nous avons étudiées et analysées pour la génération du comportement stochastique de notre système. Seule l'approche 3 a permis de mener une analyse stochastique correcte et d'obtenir des résultats conformes aux mesures expérimentales, contrairement aux approches 1 et 2.

Nous partons de la spécification LOTOS décrivant le comportement fonctionnel d'un algorithme MPI nommé ALGO, donnée par le schéma de synchronisation suivant :

$$\text{SPEC}_{ALGO} := (\underset{i=0}{\overset{n-1}{|||}} P_i) \ |[\text{ACTION}]| \ \text{TRANSFER(M, C)} \tag{5.1}$$

avec :

- n est le nombre de processus participant à l'exécution de ALGO.
- $(\underset{i=0}{\overset{n-1}{|||}} P_i)$ est une abréviation pour $P_0 \ ||| \ P_1 \ ||| \ \ldots \ ||| \ P_{n-1}$.
- P_i décrit l'exécution d'un processus parallèle de ALGO sur le processeur P_i :
  - Si ALGO est un algorithme de *ping-pong* alors
    $P_0 \equiv$ PING [ACTION, WAIT](P_0, P_1) et
    $P_1 \equiv$ PONG [ACTION, WAIT](P_1, P_0)
  - Si ALGO est algorithme de *barrière* alors
    $P_i \equiv$ BARRIER [ACTION, WAIT](P_i).
- ACTION est la porte modélisant les accès.
- WAIT est la porte modélisant les délais d'attente.
- TRANSFER est le processus qui gère la mémoire (M) et les caches (C) et satisfait les demandes d'accès.

• **Approche** 1

Dans un premier temps nous avons tenté la génération de l'IMC à partir de la spécification LOTOS sans l'ajout de portes stochastiques.

Approche 1 *Les portes* ACTION *et* WAIT *sont des portes stochastiques, elles sont transformées en taux markoviens lors de la génération de l'IMC.*

D'après le schéma de synchronisation 5.1, cette approche ne satisfait pas la règle 2.1 qui consiste à interdire la transformation des portes de synchronisation en taux markoviens. Rappelons que cette règle est imposée par fait qu'il est impossible de calculer le taux markovien d'une action résultante d'une synchronisation (voir 2.2.2). Cependant la sémantique de la synchronisation sur la porte ACTION dans notre modèle met en cause la raison de cette interdiction, car le taux markovien de l'action résultante de la synchronisation sur ACTION est connu et ne nécessite aucune évaluation, il s'agit du taux markovien de la latence d'accès.
Plus précisément :

> Les participants à la synchronisation sur la porte ACTION sont les processus P_i et le processus TRANSFER. Les rendez-vous sur ACTION modélisent des *demandes d'accès* (des requêtes) d'un point de vue P_i, et des *acquittements* d'un point de vue TRANSFER.
> En se basant sur l'interprétation stochastique de la synchronisation, cette dernière n'est possible entre P_i et TRANSFER qu'*après* la fin de la demande d'accès *et après* la fin de l'acquittement, mais en raison de la modélisation *atomique* des accès dans laquelle il n'y a pas de distinction stochastique entre les différentes phases des accès, les demandes d'accès et les acquittement sont des rendez-vous *instantanés*. *Une synchronisation* sur la porte ACTION modélise la réalisation d'un accès (demande + acquittement) auquel nous associons un taux markovien correspondant à la latence de l'accès.

Malgré la possibilité de transformation de la porte ACTION en taux markovien, l'approche suscite une incohérence entre le comportement fonctionnel et stochastique. Ceci se traduit par la perte de la concurrence entre les processus P_i dans le comportement stochastique (CTMC) en cas des accès en conflit.

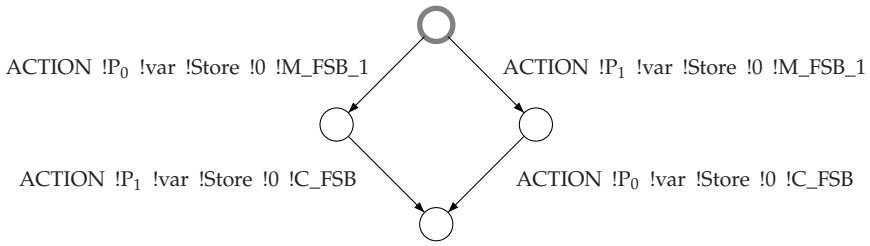
Pour mieux expliquer cette incohérence, nous allons utiliser le schéma de synchronisation 5.1 pour $n = 2$ (P_0 et P_1) avec P_i réalisant une suite infinie d'accès en écriture à une variable partagée *var*, un comportement spécifié par le processus LOTOS PROC défini ci-dessous. On suppose que les processeurs P_0 et P_1 se trouvent sur le même bus dans une architecture appliquant le protocole de cohérence de caches A.

```
process   PROC [ACTION](P : ID_Processor) : noexit :=
  ACTION   ! P ! var ! Store ! Memory_Val(0) ! latency :Latency_Value ;
  PROC [ACTION](P)
endproc
```

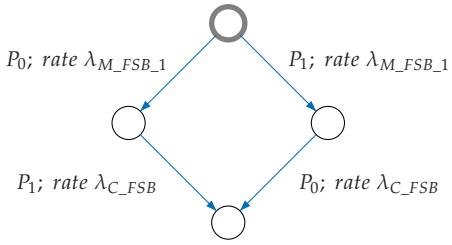
La mise en parallèle de P_0 et P_1 engendre des *conflits d'accès* pour la variable partagée *var*.

Un *conflit d'accès* se produit à chaque fois qu'il y a plusieurs demandes (≥ 2) d'accès simultanées pour la même donnée .

On reconnait un conflit d'accès dans le comportement fonctionnel par les latences figurant dans les actions modélisant les accès. La figure 5.2 illustre le comportement fonctionnel du premier cycle d'accès de P_0 et P_1 (premier accès en écriture). Initialement la variable *var* n'est valide qu'en mémoire :



Comportement fonctionnel



Comportement stochastique

FIG. 5.2 – *Comportement fonctionnel et stochastique du premier cycle d'accès en écriture*

- La branche gauche : le processus P_0 accède en premier à la variable *var* et effectue un transfert depuis la mémoire avec un taux $\lambda_{M\_FSB\_1}$. Ce transfert est ***suivi*** d'un transfert cache par P_1 avec le taux markovien λ_{C_FSB}, car après le premier accès en écriture la donnée n'est valide que dans le cache de P_0.
- La branche droite : on constate le même comportement que celui réalisé au niveau de la branche gauche, mais cette fois c'est P_1 qui accède en premier à la donnée et effectue un transfert depuis la mémoire et P_0 effectue un transfert depuis le cache de P_1 par la suite.

Dans le comportement des deux branches, l'accès au cache ne peut être effectué que si le délai d'accès à la mémoire a pris fin, imposant ainsi une séquentialité stochastique dans les accès pour un comportement initialement prévu en parallèle, d'où la cause de l'incohérence des mesures de performance obtenues par l'application de cette approche.

Dans la machine réelle, un conflit d'accès est résolu au moyen d'un mécanisme de *gestion de conflits* dans lequel il y a une prise en compte de

plusieurs paramètres, comme l'identificateur des processeurs, la distance entre la source et les différentes destinations possibles de la donnée, le type d'accès et l'historique des accès.

La spécification de ce mécanisme en LOTOS nécessite d'abord la considération des différentes phases des accès au niveau de l'architecture matérielle, ce qui engendre la modélisation du comportement de plusieurs composants matériels, par exemple, le bus et les contrôleurs de nœuds et de modules dans l'architecture FAME. Ainsi dans la spécification LOTOS, un accès n'est plus une action atomique mais plutôt une suite d'actions.

Afin d'éviter l'explosion d'espace d'états du comportement fonctionnel et stochastique de notre système, nous avons opté pour la préservation de l'atomicité des accès dans le modèle et par conséquent l'abstraction de la gestion des conflits dans la spécification LOTOS.

• **Approche** 2

Cette approche respecte la règle 2.1 par l'insertion des taux markoviens moyennant une nouvelle porte stochastique.

Approche 2 *Les taux markoviens des latences d'accès correspondent à des rendez-vous sur une porte stochastique nommée* LATENCY *dans le processus* TRANSFER *et,* WAIT *restant une porte stochastique.*

On note par TRANSFER_LATENCY le processus LOTOS spécifiant le nouveau comportement du processus TRANSFER. Il consiste en la réalisation d'un rendez-vous sur la porte stochastique LATENCY après chaque rendez-vous sur la porte ACTION, comme décrit ci-dessous.

```
process   TRANSFER_LATENCY [ACTION, LATENCY](M :Memory, C :Cache) : noexit :=
 ACTION   ? P :ID_Processor   ? adr :Address   ? op :ID_Action
          ? val :Memory_Value   ? latency :Latency_Value
   [
    (((op == Load) and (val == Load_Memory(M, adr))) or
     ((op == Fetch_and_decrement) and
      (val == Load_Memory(Update_Memory(M, adr, op, val), adr))) or
     (op == Store)
    ) and
    (latency == Evaluation_Latency(M, C, op, adr, P))
   ];
 LATENCY   ! P ! adr ! latency ;
 TRANSFER_LATENCY [ACTION, LATENCY](Update_Memory(M, adr, op, val),
                                    UpDate_Cache(C, op, adr, P))
endproc
```

Nous avons constaté que l'utilisation du processus TRANSFER_LATENCY dans le cas des processus P_i qui réalisent aussi des traitements autres que les demandes d'accès, des rendez-vous sur la porte WAIT, par exemple, crée des choix entre les taux markoviens des différents traitements qui sont fonctionnellement réalisés de façon séquentielle, comme expliqué dans l'exemple 5.2.

Exemple 5.2 *Soit le comportement du processus P_i modélisé par le processus LOTOS* PROC *défini comme suit :*

```
process   PROC [ACTION, WAIT](P : ID_Processor) : noexit :=
 ACTION   ! P ! var ! Store ! Memory_Val(0) ! latency :Latency_Value ;
 WAIT     ! P ! 0 of Nat ;
 PROC [ACTION, WAIT](P)
endproc
```

Le processus P_i réalise une suite infinie d'un accès en écriture suivi d'une attente d'un certain délai. La figure 5.3 illustre le comportement stochastique (IMC) résultant de la synchronisation du processus P_0 et TRANSFER_LATENCY *sur la porte* ACTION *pour la première demande d'accès et la première attente.*

D'après la spécification LOTOS de PROC*, le rendez-vous sur la porte* WAIT *a toujours lieu après le rendez-vous sur la porte* ACTION*, ce qui signifie stochastiquement que P_0 ne peut effectuer une attente que si le délai de l'accès à pris fin, c'est à dire le rendez-vous sur la porte* LATENCY *eu lieu. Cet ordonnancement n'a pas était respecté dans l'IMC générée comme illustré dans la figure 5.3 : les taux markoviens de l'accès et de l'attente décrivent un comportement stochastique concurrent pour un comportement prévu séquentiel. Ce parallélisme entre les taux markoviens est dû au fait que P_0 n'a pas été informé de la fin du taux markovien de son accès en écriture (réalisation du rendez-vous sur* LATENCY*).*

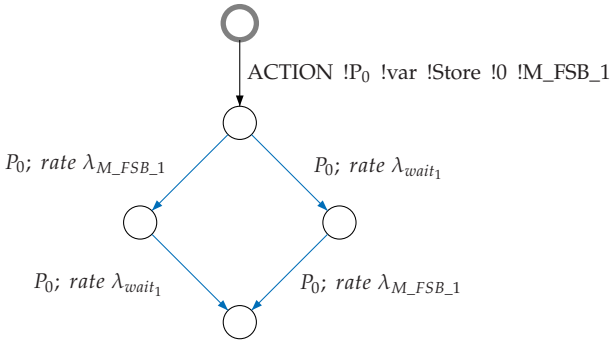


FIG. 5.3 – *IMC par le processus* TRANSFER_LATENCY

Pour pallier ce problème, nous avons défini une nouvelle porte END qui marquera la fin de l'insertion du taux markovien des accès. On note par P_i^{END} et TRANSFER_END, les processus P_i et TRANSFER_LATENCY augmentés par des rendez-vous sur la porte END respectivement. Ainsi, P_i^{END} et TRANSFER_END se synchronisent sur les portes ACTION et END.

```
process   TRANSFER_END [ACTION, LATENCY, END](M :Memory, C :Cache) : noexit :=
 ACTION   ? P :ID_Processor   ? adr :Address   ? op :ID_Action
          ? val :Memory_Value   ? latency :Latency_Value
   [
    (((op == Load) and (val == Load_Memory(M, adr))) or
     ((op == Fetch_and_decrement) and
      (val == Load_Memory(Update_Memory(M, adr, op, val), adr))) or
     (op == Store)
    ) and
    (latency == Evaluation_Latency(M, C, op, adr, P))
   ];
 LATENCY   ! P ! adr ! latency ;
 END   ! P ;
 TRANSFER_END [ACTION, LATENCY, END](Update_Memory(M, adr, op, val),
                                     UpDate_Cache(C, op, adr, P))
endproc
```

Remarque 5.2 *Les synchronisations sur la porte* END *ne font pas partie de la modélisation des accès, ainsi leur atomicité est totalement préservée.*
Le rôle de la porte END *est limité à l'interdiction des conflits entre les taux markoviens des différents traitements : accès et attente.*

On note par SPEC_{ALGO}^{END} le schéma de synchronisation spécifiant les aspects temporels de ALGO par les rendez-vous sur les portes stochastiques LATENCY et WAIT.

$$\text{SPEC}_{ALGO}^{END} := (\overset{n-1}{\underset{i=0}{|||}} P_i^{END}) \;|[\text{ACTION, END}]|\; \text{TRANSFER\_END(M, C)} \qquad (5.2)$$

Le modèle stochastique que nous avons obtenu par l'application des étapes de génération de CTMC décrites dans la section 5.1 sur la spécification SPEC_{ALGO}^{END} dénote le comportement d'une IMC en guise de CTMC : la réduction stochastique n'a pas réussi à éliminer toutes les transitions internes "*i*". L'IMC obtenue fait partie de l'ensemble des IMC *non-transformables* en CTMC.

Il était primordial, avant de se lancer dans la recherche et la définition d'une nouvelle approche garantissant la génération de CTMC, de comprendre la cause et d'explorer l'origine de l'échec de la réduction stochastique dans l'élimination des états non-déterministes.

L'analyse du non-déterminisme au niveau d'IMC après l'abstraction des actions interactives et l'application de la réduction stochastique ne garantit pas l'obtention de résultats concluants a propos de l'origine du non-déterminisme, car ces transformations donnent généralement un aspect *irrégulier* au comportement de l'IMC par rapport au comportement du système : il est pratiquement impossible d'établir le lien entre les actions internes et leur origine en tant que actions interactives après la réduction stochastique. Pour cette raison, nous avons effectué notre étude sur le comportement généré à partir de la spécification SPEC_{ALGO}^{END} au niveau

de trois étapes de transformation : avant abstraction, après abstraction, après abstraction et réduction stochastique, en se basant sur la définition du non-déterminisme donnée ci-dessous.

Définition 5.1 *Un système de transitions étiquetées $M = (S, A, T, s_0)$ est dit non-déterministe ssi :*

$$\exists\, s_{\neg d} \in S,\ \exists\, E \subset S \ \textit{tel que}\ \forall\, s' \in E :\ \exists\, a \in A \ \textit{et}\ s_{\neg d} \xrightarrow{a} s' \in T, |E| \geq 2$$

*Si toutes les actions des transitions sortantes de l'état non-déterministe $s_{\neg d}$ sont identiques, alors on dit qu'il s'agit d'un non-déterminisme **interne**, sinon on dit que c'est un non-déterminisme **externe**.*

Le non-déterminisme interne n'est pas uniquement lié aux actions internes. La différence de base entre le non-déterminisme externe et interne est que le non-déterminisme externe peut être résolu par le biais de certains influences externes, contrairement au non-déterminisme interne qui est effectivement hors du contrôle de l'environnement. La figure 5.4 illustre quelques exemples des différents types de non-déterminisme :

- Le comportement A comprend un non-déterminisme externe (le choix entre les actions interactives b et c). La synchronisation sur l'action c avec le comportement $stop$ engendre un comportement déterministe : le branche avec l'action c sera éliminée.
- Le comportement B comprend un non-déterminisme interne par des actions interactives. La synchronisation sur l'action a avec n'importe quel comportement externe, ne permet pas l'élimination du non-déterminisme.
- Le comportement C comprend un non-déterminisme interne par des actions internes i.

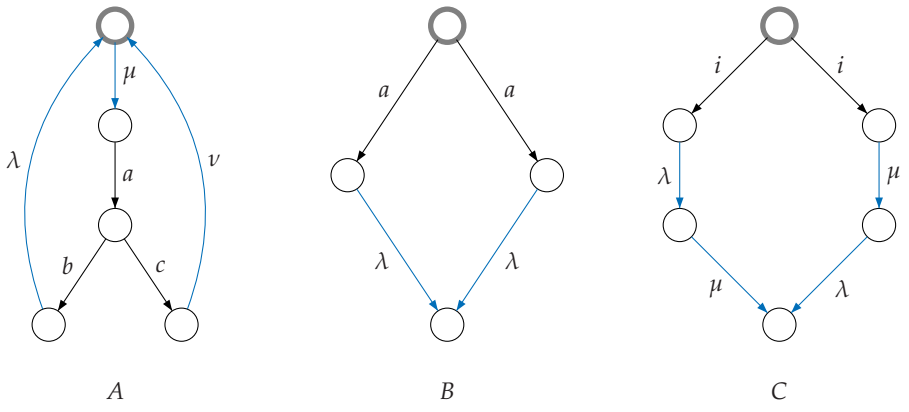


FIG. 5.4 – *Exemples de comportements non-déterministes*

Le non-déterminisme est une notion importante dans les processus interactifs et les algèbres de processus en général. Il est utilisé dans la modélisation de plusieurs aspects, qui autrement ne pourraient pas être exprimés [86] :

- Libre implémentation (*Implementation freedom*) : pour un processus interactif qui peut être considéré comme une spécification abstraite, le non-déterminisme représente une liberté dans son implémentation. C'est à dire, si pour un certain état il y a deux transitions qui peuvent être choisis de façon non-déterministe, alors l'implémentation ne pourrait avoir qu'une seule des deux transitions.
- Libre ordonnancement (*Scheduling freedom*) : représente l'utilisation classique du non-déterminisme. Plusieurs processus s'exécutent en parallèle et il y a une liberté dans le choix du processus exécutant la prochaine transition.
- Environnement externe : les actions interactives représentent des interactions possibles avec l'environnement par le biais de synchronisations. Les capacités d'interaction avec cet environnement influencent la façon dont le choix est déterminé.

Nous avons analysé le comportement de notre système, ainsi que sa modélisation en LOTOS par rapport aux aspects engendrant le non-déterminisme cités ci-dessus et, nous avons mis en évidence ses caractéristiques déterministes et non-déterministes :

- *Les caractéristiques déterministes :*
  (a) Les primitives MPI engendrent un comportement déterministe : l'exécution des instructions est séquentielle et le choix entre elles est toujours conditionné (généralement par le contenu des variables).
  (b) Les changements des états des caches et l'évaluation des types de transfert sont soumis à des règles bien précises : le protocole de cohérence de caches.
  (c) Aucune interaction avec l'environnement externe n'est présente dans le modèle : dans la spécification LOTOS, il n'existe pas de rendez-vous de la forme G ? ... où G est une porte qui ne fait pas l'objet de synchronisation.
  (d) Le comportement fonctionnel généré à partir de la spécification SPEC_{ALGO} (avant l'insertion des taux markoviens) ne comporte pas d'actions internes (zéro i-transitions).
- *Les caractéristiques non-déterministes :*
  (a) Les processus P_i sont complètement parallèles entre eux.
  (b) Dans le modèle LOTOS, toutes les demandes d'accès sont traitées par un seul processus central TRANSFER (TRANSFER_END dans SPEC^{END}_{ALGO}).
  (c) Dans le cas d'un conflit d'accès, les accès sont fonctionnellement parallèles mais stochastiquement séquentiels (exemple illustré dans la figure 5.2).

Le comportement des processus P_i^{END} et P_i généralement, est une suite infinie de demandes d'accès et, par conséquent, une suite infinie de demandes de synchronisation sur la porte ACTION avec le processus TRANSFER. Dans la spécification LOTOS des processus P_i^{END} il n'y a pas de notion de passage de temps concernant les accès, c'est à dire entre deux

demandes d'accès il y a *zéro temps passé*, ce qui implique que les processus P_i^{END} sont en demande *permanente* de synchronisation sur la porte ACTION.

Supposons que le processus TRANSFER_END répond à une demande d'accès de P_0^{END} par la synchronisation sur la porte ACTION, ainsi il ne peut satisfaire d'autres demandes qu'après l'insertion du taux markovien de la latence d'accès de P_0^{END} et l'informer que le rendez-vous sur LATENCY a lieu par la synchronisation sur la porte END, par conséquent :

1. Tous les processus $P_{i\neq 0}^{END}$ sont en attente de la réalisation de la synchronisation entre TRANSFER_END et P_0^{END} sur la porte END, ce qui signifie qu'ils ne peuvent réaliser un accès *qu'après* un délai moyen correspondant à la latence de l'accès de P_0^{END}, contredisant ainsi le comportement parallèle prévu ente les processus P_i^{END}.
2. Suite à la réalisation de la synchronisation entre TRANSFER_END et P_0 sur la porte END, le processus TRANSFER_END reçoit des demandes d'accès de tous les processus y compris P_0^{END} (les processus $P_{i\neq 0}^{END}$ étaient déjà en attente et P_0^{END} entame une nouvelle demande), ainsi il se trouve dans une situation de sélection non-déterministe : toutes les demandes d'accès arrivent en même temps.

D'un point de vue stochastique , la probabilité que les demandes d'accès arrivent *en même temps* est nulle selon la propriété 2.5 de la loi de distribution exponentielle (voir la section 2.2.1), et pour cette raison, la réduction stochastique est incapable d'éliminer les états non-déterministes dans l'IMC générée à partir de la spécification SPEC_{ALGO}^{END}.

• **Approche** 3

Approche 3 *Insertion des taux markoviens par des rendez-vous sur la porte stochastique* LATENCY *dans un nouveau processus LOTOS nommé* LATENCY_PROC.

On note par $\text{SPEC}_{ALGO}^{LATENCY}$ le schéma de synchronisation spécifiant les aspects temporels de ALGO par les rendez-vous sur les portes stochastiques LATENCY dans le processus LATENCY_PROC,

$$\begin{aligned} \text{SPEC}_{ALGO}^{LATENCY} \quad = \quad & \underset{i=0}{\overset{n-1}{|||}} \left(P_i \; |[\text{ACTION, WAIT}]| \; \text{LATENCY\_PROC}(P_i)\right) \\ & |[\text{ACTION}]| \\ & \text{TRANSFER(M, C)} \end{aligned} \tag{5.3}$$

Le processus LATENCY_PROC possède un comportement récursif d'une fonctionnalité *noexit*. Il est paramétré par les portes interactives ACTION et WAIT et, par la porte stochastique LATENCY, ainsi que l'identificateur du processus. Son rôle consiste en l'insertion des taux markoviens suite à chaque accès (synchronisation sur la porte ACTION) ou à une attente (synchronisation sur la porte WAIT) par la réalisation de rendez-vous sur la porte LATENCY.

```
process  LATENCY_PROC [ACTION, LATENCY, WAIT](P : ID_Processor) : noexit :=
 ACTION  ! P  ? adr :Address  ? op :ID_Action  ? val :Memory_Value
          ? latency :Latency_Value ;
 LATENCY  ! P ! adr ! latency ;
 LATENCY_PROC [ACTION, LATENCY, WAIT](P)
[ ]
 WAIT  ! P ? id_wait :Nat ;
 LATENCY  ! p ! id_wait ;
 LATENCY_PROC [ACTION, LATENCY, WAIT](P)
endproc
```

Le processus LATENCY_PROC(P_i) participe à tous les rendez-vous que P_i réalise sur la porte ACTION et WAIT :

ACTION : pour obtenir la latence de l'accès LATENCY échangée sur l'offre de type **Latency_Value**.

WAIT : pour préserver l'ordonnancement temporel des accès et des attentes.

Remarque 5.3 *Dans cette approche, la porte* WAIT *est interactive, contrairement aux approches précédentes ou elle est considérée comme une porte stochastique. Ceci dans le but d'assurer une insertion des taux markoviens des différents traitements (accès et attente) conforme au comportement fonctionnel prévu initialement et interdire ainsi les conflits entre eux (voir l'exemple 5.2).*

Nous clarifions quelques points importants concernant la définition de la spécification $\text{SPEC}_{ALGO}^{LATENCY}$:

1. Il est possible de reproduire le comportement généré par $\text{SPEC}_{ALGO}^{LATENCY}$ *sans* l'utilisation des processus LATENCY_PROC(P_i). Ceci, par la réalisation des rendez-vous sur la porte LATENCY au niveau des processus P_i après chaque demande d'accès (rendez-vous sur ACTION). En revanche, cette méthode éprouve une certaine lourdeur, pour les modifications qu'elle engendre dans la spécification LOTOS de P_i (insertion des rendez-vous sur la porte LATENCY), ainsi que la complexité qu'elle entraine pour la génération compositionnelle (section 6.2) de l'IMC.
2. Chaque processus P_i se synchronise avec son propre processus LATENCY_PROC(P_i) afin d'éviter le problème de non-déterminisme ainsi que la perte de la concurrence stochastique entre les processus P_i. En effet, le modèle généré par l'utilisation d'un seul processus LATENCY_PROC est similaire à celui généré par l'utilisation du processus TRANSFER_END (section 5.2.1).
3. Toutes les informations temporelles du système (les latences des accès et les délais d'attente) sont prises en charge par le processus LATENCY_PROC afin d'assurer la cohérence stochastique des évènements.

L'application de cette approche nous a permis de générer le comportement stochastique CTMC de notre système et par conséquent de pouvoir évaluer ses performances.

5.2.2 Évaluation des indices de performances

Nous définissons l'indice de performances *LM* d'un algorithme MPI comme étant la latence moyenne de ses processus parallèles. Il est mesuré en terme des latences d'accès et des délais d'attente.

L'interprétation de cet indice dépend fortement de la *mission*[4] accomplie par l'algorithme :

$LM_{ping-pong}$: la latence moyenne d'un échange d'un message entre processus.

$LM_{barrier}$: la latence moyenne de synchronisation entre plusieurs processus.

Remarque 5.4 *Il existe bien d'autres indices de performances, comme le débit maximal des données, mais que nous n'avons pas pris en compte dans notre procédure d'évaluation des performances.*

Définition 5.2 *Soient LM_{ALGO1} et LM_{ALGO2} les indices de performances des algorithmes MPI* ALGO1 *et* ALGO2 *respectivement, avec* ALGO1 *et* ALGO2 *accomplissant le même type de mission. Nous avons :*

$$\textit{Si } LM_{ALGO1} < LM_{ALGO2} \textit{ alors } \textsc{algo1} \textit{ est plus } \textbf{\textit{performant}} \textit{ que } \textsc{algo2}$$

Question ? *Comment évaluer l'indice de performance LM à partir de la spécification LOTOS ?*

La figure 5.5 illustre le comportement de l'algorithme de *ping-pong* en fonction de l'ordre d'exécution des primitives *send* et *receive* et le comportement de la *barrière* entre deux processus en fonction de l'ordre d'arrivée à la barrière par l'exécution de la primitive *barrier_prim* (P_0 suivi de P_1 et inversement).

En se basant sur la définition des indices de performances de chaque algorithme, nous avons :

cas de *ping-pong* : un cycle de *ping-pong* comprend deux exécutions de la primitive *send*, la première par P_0 et la seconde par P_1 et deux exécutions de la primitive *receive*, la première par P_1 et la seconde par P_0. Ceci signifie l'envoi et la réception de *deux* messages. Si on note $LCycle_{ping-pong}$ la latence d'un cycle, alors :

$$LM_{ping-pong} = \frac{LCycle_{ping-pong}}{2}$$

Avec 2 le nombre de messages échangés durant un cycle.

cas de *barrière* : un cycle de *barrière* comprend une seule exécution de la primitive *barrier_prim* par processus : P_0 suivi de P_1 *ou* P_1 suivi de P_0. Par conséquent, si $LCycle_{barrier}$ est la latence d'un cycle, alors :

$$LM_{barrier} = LCycle_{barrier}$$

Remarque 5.5 *Les comportements des algorithmes illustrés dans la figure 5.5 sont obtenus à partir de la réduction des graphes générés depuis les spécifications LOTOS suite à des renommages et des abstractions.*

[4]La mission d'un algorithme de *ping-pong* consiste en l'échange de messages entre processus et, en la synchronisation entre les processus pour un algorithme de *barrière*.

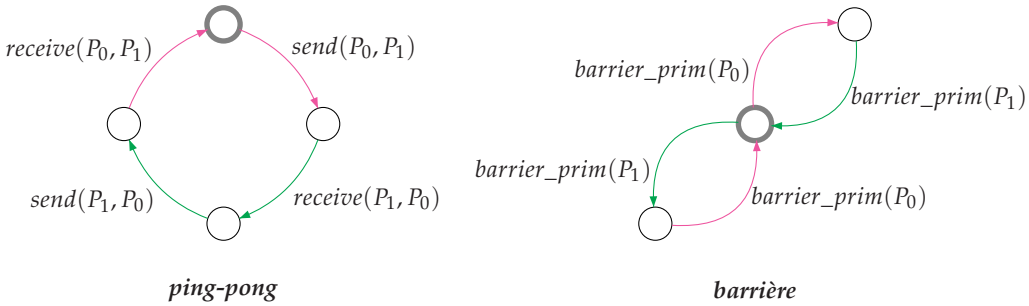


FIG. 5.5 – *Le comportement de ping-pong et de barrière entre deux processus*

Définition 5.3 *Si LM est la latence moyenne de la mission d'un algorithme MPI et LCycle est la latence moyenne d'un cycle de son exécution, alors le débit moyen DM est donné par*

$$DM = \frac{1}{LCycle} \tag{5.4}$$

Par conséquent

$$LM_{ping-pong} = \frac{1}{2\ DM_{ping-pong}}$$

$$LM_{barrier} = \frac{1}{DM_{barrier}}$$

représente le nombre moyen de missions réalisées par unité de temps (unité de temps de la latence LM).

$DM_{ping-pong}$: le nombre moyen de messages échangés par unité de temps.

$DM_{barrier}$: le nombre moyen de synchronisations réalisées entre les processus par unité de temps.

Si on note par $\{X_t, t \in T\}$ la CTMC représentant le comportement stochastique de l'algorithme de *ping-pong* ou de *barrière*, alors d'après le théorème 2.3, nous avons :

$$DM = \sum_{i \in S}\ \pi_i^* \lambda_{start}^i$$

où :

- S est l'ensemble d'état de la CTMC $\{X_t, t \in T\}$.
- La quantité π_i^* représente la distribution stationnaire de la CTMC à l'état i.
- λ_{start}^i est la somme des taux markoviens selon lesquels on quitte l'état i. Ces taux représente les taux markovien des transitions correspondant au début de la mission de l'algorithme.

L'outil BCG_STEADY de CADP permet l'évaluation du débit de toutes les transitions markoviennes préfixées dans une CTMC.

Nous avons utilisé le préfixe START pour les transitions markoviennes qui marquent le début de la mission de l'algorithme, afin de les distinguer des autres transitions. Ainsi la quantité retournée par BCG_STEADY pour les transitions de préfixe START correspond au débit moyen DM de l'algorithme.

Nous définissons deux types de transitions par rapport à la mission de l'algorithme dans le graphe BCG modélisant son comportement :

start : représente toutes les transitions qui marquent le début de la mission de l'algorithme.

$\neg start$: représente toutes les transitions qui ne marquent pas le début de la mission de l'algorithme.

Le début de la mission d'un algorithme MPI correspond à l'exécution de la première instruction de ses primitives. Le tableau 5.1 illustre quelques exemples de ces instructions dans l'algorithme *ping-pong* et la barrière *tournament*.

Primitive	1^{ere} instruction
send	**if** *Local_Available_Ptr[i] != Null* **then** ... **else** ...
	ACTION ! Pi ! Local_Available_Ptr(Pi) ! Load ? val :Memory_Value ? latency :Latency_Value [val <> Null_Memory] ; ... [ ] **ACTION** ! Pi ! Local_Available_Ptr(Pi) ! Load ! Null_Memory ? latency :Latency_Value ; ...
tournament	**case** *rounds* [*vpid*] [*round*].*role* **of** *loser :* *rounds*[vpid][*round*].*opponent*^:= *sense* ; ... *winner :* **repeat until** *rounds[vpid][round].flag = sense* ; ...
	ACTION ! vpid ! role(vpid,round) ! Load ! loser ? latency :Latency_Value ; ... [ ] **ACTION** ! vpid ! role(vpid,round) ! Load ! winner ? latency :Latency_Value ; ...

TAB. 5.1 – *Les instructions qui marquent le début de mission*

Dans les différentes primitives que nous avons modélisées, la première instruction est une structure conditionnelle basée sur le contenu d'une variable . Par exemple, le contenu de la variable *Local_Available_Ptr* dans la primitive *send* et celui de la variable *role* dans la primitive *tournament*. Ceci entraîne des difficultés dans la détection des actions de type *start* dans les graphes comportementaux et stochastiques :

1. Les instructions conditionnelles engendrent un choix entre plusieurs actions de type *start*.
2. Les actions de type *start* ne portent aucune information indiquant qu'elles correspondent au début de l'exécution de la primitive, ce qui crée une ambigüité avec d'autres actions identiques de type $\neg start$.

Pour pallier ce problème, nous avons proposé deux méthodes pour l'identification des actions de type *start* dans la spécification LOTOS :

Méthode 1 *Ajouter aux rendez-vous sur les portes* ACTION, LATENCY *et* WAIT *une nouvelle offre de type* ***Nat*** *sur laquelle deux valeurs possibles sont échangées :* 0 *pour une action de type* start *et* 1 *sinon.*

Exemple 5.3 *Le processus* PROC *défini ci-dessous réalise une suite infinie d'accès en lecture suivie d'une attente d'un certain délai. La dernière offre du rendez-vous sur la porte* ACTION *(resp. la porte* WAIT*) donnée par "* !0 **of** Nat*" (resp. "*!1 **of** Nat*") indique que l'action générée est de type start (resp. ¬start)*

```
process  PROC [ACTION, WAIT](P : ID_Processor) : noexit :=
 ACTION  ! P ! var ! Load ? val :Memory_Value ! latency :Latency_Value ! 0 of Nat ;
 WAIT    ! P ! 1 of Nat ! 1 of Nat ;
 PROC [ACTION, WAIT](P)
endproc
```

Ci-dessous la nouvelle définition du processus LATENCY_PROC *suite à l'ajout du type des actions dans les rendez-vous sur toutes les portes :*

```
process  LATENCY_PROC [ACTION, LATENCY, WAIT](P : ID_Processor) : noexit :=
 ACTION  ! P  ? adr :Address  ? op :ID_Action  ? val :Memory_Value
          ? latency :Latency_Value  ? action_type :Nat ;
 LATENCY  ! P ! adr ! latency ! action_type ;
 LATENCY_PROC [ACTION, LATENCY, WAIT](P)
[ ]
 WAIT    ! P ? id_wait :Nat ? action_type :Nat ;
 LATENCY  ! p ! id_wait ! action_type ;
 LATENCY_PROC [ACTION, LATENCY, WAIT](P)
endproc
```

Notons que les rendez-vous sur la porte ACTION *dans le processus* TRANSFER *subissent également l'ajout de l'offre indiquant le type de l'action (start ou non).*

Á l'étape de 2 de la génération de CTMC décrite dans la section 5.1, qui consiste à renommer toutes les transitions stochastiques en taux markoviens, on opte pour un renommage permettant à BCG_STEADY d'évaluer le débit DM en précisant le préfixe START *pour les taux markoviens issus de transitions de type start, comme suit :*

"**LATENCY** ! P ! var ! C_INT !0" → "START ; *rate* λ_{C_INT}",
"**LATENCY** ! P ! var ! M_FSB_1 !0"→ "START ; *rate* $\lambda_{M\_FSB\_1}$",
"**LATENCY** ! P ! var ! C_FSB !0" → "START ; *rate* λ_{C_FSB}",
"**LATENCY** ! P ! 1 !1" → "*rate* λ_{wait_1}"

Méthode 2 *Ajouter un rendez-vous sur la porte* WAIT *dans les processus* P_i *avant la réalisation de n'importe quel traitement (accès ou attente) :*

WAIT ! P **of** ID_Processor !0 **of** Nat ;

où P est l'identificateur du processeur, et le chiffre 0 *est réservé aux actions de type* start.

Exemple 5.4 *On reprend le comportement du processus* PROC *de l'exemple 5.4 mais en marquant le début de la mission par un nouveau rendez-vous sur la porte* WAIT *et sans modifier les rendez-vous sur les portes* ACTION *et* WAIT.

```
process   PROC [ACTION, WAIT](P : ID_Processor) : noexit :=
  WAIT    ! P ! 0 of Nat ;
  ACTION  ! P ! var ! Load ? val :Memory_Value ! latency :Latency_Value ;
  WAIT    ! P ! 1 of Nat ;
  PROC [ACTION, WAIT](P)
endproc
```

Á la différence de la méthode précédente, où les taux markoviens des transitions de type start correspondent aux taux markoviens des latences d'accès, dans cette méthode, la transition stochastique modélisant le début de la mission de l'algorithme ne fait pas partie de son comportement. Par conséquent, nous lui avons attribué un taux markovien suffisamment grand pour qu'il n'ait pas un impact sur la latence globale ou encore sur la performance de l'algorithme (un taux très grand implique une latence très petite qui tend vers 0*). Ainsi, nous avons appliqué le renommage suivant :*

"**LATENCY** ! P ! 0" → "START ; *rate* 100000",
"**LATENCY** ! P ! var ! C_INT" → "*rate* λ_{C_INT}",
"**LATENCY** ! P ! var ! M_FSB_1" → "*rate* $\lambda_{M\_FSB\_1}$",
"**LATENCY** ! P ! var ! C_FSB" → "*rate* λ_{C_FSB}",
"**LATENCY** ! P ! 1" → "*rate* λ_{wait_1}"

L'utilisation de la méthode 1 présente un avantage en cas d'explosion combinatoire de l'espace d'états du modèle, car elle n'engendre pas de nouveaux états. En revanche, elle nécessite la modification de tous les rendez-vous sur la porte ACTION et WAIT dans tous les processus de la spécification LOTOS : P_i, TRANSFER et LATENCY_PROC.

L'inconvénient de cette méthode est que pour certaines études de performances, elle nécessite des modifications au niveau de la spécification LOTOS au lieu de simples modifications au niveau du fichier script SVL. Par exemple, dans l'algorithme de barrière *centralized*, le premier accès est un accès en lecture à la variable privée *local_sense*. Si nous nous intéressons à l'étude de l'impact des variables sur les performances de l'algorithme, nous allons comparer les résultats de deux modèles :

Modèle *a* : la CTMC générée comporte les taux markoviens concernant tous les accès aux variables , y compris les taux markoviens des accès aux variables privées. Nous utilisons de ce fait la spécification LOTOS de la primitive *centralized* avec la précision que le premier accès génère une transition de type *start*, comme suit :

```
process   BARRIER [ACTION](P : ID_Processor) : exit :=
 ACTION   ! P ! local_sense(P) ! Load ? val :Memory_Value
          ? latency :Latency_Value ! 0 of Nat ;
 ACTION   ! P ! local_sense(P) ! Store ! Inv_Value(val)
          ? latency :Latency_Value ! 1 of Nat ;
 ACTION   ! P ! Count ! Fetch_and_decrement ? val :Memory_Value
          ? latency :Latency_Value ! 1 of Nat ;
  ...
endproc
```

Modèle *b* : la CTMC générée comporte les taux markoviens concernant uniquement les accès aux variables partagées. Si nous utilisons la spécification de la primitive *centralized* du modèle précédent, on perd tout trace des transitions de type *start*, puisqu'on procède à l'abstraction des taux markoviens des accès aux variables privées. Pour éviter cela, il est nécessaire de modifier le type des rendez-vous en terme de *start* et $\neg start$. Dans la primitive *centralized* le 3^{eme} rendez-vous sur la porte ACTION génère une transition de type *start*, car elle spécifie une accès à la variable partagée *count*.

```
process   BARRIER [ACTION](P : ID_Processor) : exit :=
 ACTION   ! P ! local_sense(P) ! Load ? val :Memory_Value
          ? latency :Latency_Value ! 1 of Nat ;
 ACTION   ! P ! local_sense(P) ! Store ! Inv_Value(val)
          ? latency :Latency_Value ! 1 of Nat ;
 ACTION   ! P ! Count ! Fetch_and_decrement ? val :Memory_Value
          ? latency :Latency_Value ! 0 of Nat ;
  ...
endproc
```

L'utilisation de la méthode 1 nécessite des modifications au niveau de la spécification des primitives de telle sorte qu'elles soient en cohérence avec l'étude stochastique. Ce problème ne surgit pas avec la méthode 2 puisqu'il s'agit d'un rendez-vous qui ne fait pas partie du comportement de l'algorithme. En revanche, la méthode 2 fait sensiblement augmenter la taille de l'espace d'états.

Le tableau suivant indique la taille de la CTMC de la barrière *centralized* générée par l'application de la méthode 1 et 2, pour le modèle *b* de l'exemple précédent dans lequel nous avons masqué les latences d'accès aux variables privées.

Processus	Méthode 1		Méthode 2	
	États	Transitions	États	Transitions
2	38	76	39	77
3	177	517	179	503
4	601	2.116	604	2.067

TAB. 5.2 – *Comparaison de la taille de CTMC obtenue par l'application de méthodes différentes pour distinguer les transitions de type start*

La différence entre les graphes est très petite, est elle de l'ordre de 1,4% (l'équivalent de 2 états environ). Pour cette raison nous avons opté pour l'utilisation de la méthode 2 pour la facilité de son utilisation et son adaptation à toutes les analyses de performances sans avoir besoin de faire des modifications au niveau de la spécification LOTOS.

5.3 Applications

Nous présentons dans cette section les indices de performances des algorithmes de *ping-pong* et de *barrières* dans l'architecture FAME.

Les comportements stochastiques sont obtenus par l'application des étapes de génération de CTMC présentées dans la section 5.1, après l'insertion des taux markoviens dans la spécification LOTOS suivant l'approche 5.2.1, ainsi que l'identification du début de la mission de l'algorithme par le rendez-vous sur la porte WAIT selon la méthode 2.

Les indices de performances sont évalués par l'outil BCG_STEADY.

Remarque 5.6 *Afin de faciliter la référence à l'indice de performance dans le cas des deux variantes d'algorithme MPI (*ping-pong *et* barrière*), on appelle* ***latence globale*** *la latence moyenne d'échange d'un message dans le cas de* ping-pong *et la latence moyenne de synchronisation dans le cas des* barrières*.*

Nous avons comparé les différents résultats (taille des graphes, les latences) en utilisant le pourcentage de différence.

Rappelons qu'un pourcentage de différence entre deux valeurs x et y est évalué selon la formule suivante :

$$|x-y| \left(\frac{x+y}{2}\right)^{-1} 100$$

5.3.1 Algorithme *ping-pong*

Les mesures expérimentales de la latence moyenne d'un échange d'un message de longueur nulle réalisé par l'algorithme *ping-pong* (présenté dans 3.3.1) sur l'architecture FAME ont montré qu'il n'est pas *performant*. Cependant il n'y avait aucun indice justifiant ces mesures. En d'autre terme, il était impossible de connaître la raison de la *non-performance* de l'algorithme : est ce que c'est dû à des aspects *matériels* ou *logiciels* ?

Répondre à cette question fut l'objectif de départ de ce travail de thèse.

Le tableau 5.3 ci-dessous illustre les latences globales mesurées expérimentalement dans le cas du protocole de cohérence de caches A. Les expériences ont montré que le nombre de paquets disponibles initialement dans la liste "Local_Available_list" n' a aucun impact sur la latence moyenne d'un échange de message.

Topologie	LM (μs)
$topology_0$	1,5
$topology_1$	3
$topology_2$	4,5

TAB. 5.3 – *Les Latences expérimentales du* ping-pong *dans le cas du protocole de cohérence de cache A dans l'architecture FAME*

Nous avons évalué l'indice de performance de l'algorithme *ping-pong* en faisant varier les paramètres suivants :

- Le nombre de paquets disponibles initialement dans la liste "Local_Available_list".
- Le protocole de cohérence de caches : A et B.
- La topologie : $topology_0$, $topology_1$ et $topology_2$.

Notons que notre étude de l'algorithme *ping-pong* 3.3.1 a fait l'objet de la publication [8].

• ***La taille de l'espace d'états***

L'augmentation du nombre de paquets disponibles initialement dans la liste "Local_Available_list" a provoqué une explosion combinatoire de l'espace d'états des différents comportements (fonctionnel et stochastique). Ainsi, la génération de l'IMC s'est limitée à un nombre de paquets ≤ 2.

Nous avons tenté de contourner ce problème une utilisant le principe de génération compositionnelle, que nous présenterons dans la section 6, cependant, l'explosion de l'espace d'états surgit dès que le nombre de paquets est supérieur ou égal à 4.

Le tableau 5.4 indique la taille de l'espace d'états des graphes BCG des comportements stochastiques IMC et CTMC. Notons que, pour un nombre de paquets égal à 3, la CTMC est obtenue par génération compositionnelle.

Topologie	Paquets	Type	Protocole *A*		Protocole *B*	
			États	Transitions	États	Transitions
$topology_0$	1	IMC	9.242	18.484	9.118	18.236
		CTMC	2.477	4.954	2.407	4.814
	2	IMC	127.756	255.512	120.686	241.372
		CTMC	14.948	29.896	13.718	27.436
	3	IMC	–	–	–	–
		CTMC	33.289	66.578	29.186	58.372
$topology_{1,2}$	1	IMC	9.341	18.682	9.217	18.434
		CTMC	2.522	5.044	2.452	4904
	2	IMC	127.855	255.710	120.785	241.570
		CTMC	14.993	29.986	13.763	27.526
	3	IMC	–	–	–	–
		CTMC	33.316	66.632	29.213	58.426

TAB. 5.4 – *La taille de l'espace d'états du comportement stochastique de l'algorithme de* ping-pong *sur FAME et MESCA*

Remarque 5.7 *La taille de l'espace d'états des comportements stochastiques générés dans le cas de la* $topology_2$ *est identique à celle des comportements stochastiques générés dans le cas de la topologie* $topology_1$*.*

La procédure de génération de CTMC implique une augmentation considérable de la taille de l'espace d'états au niveau d'IMC. On constate en effet que les IMC engendrent un espace d'états *beaucoup* plus grand que celui du comportement fonctionnel et stochastique CTMC. La figure 5.6 illustre la variation de la taille de l'espace d'états dans le cas de 2 paquets dans le cas de l'architecture FAME. Pour générer la CTMC, on passe d'un graphe de 23.571 états (modélisant le comportement fonctionnel, voir le tableau 3.3) à un autre graphe 5 fois plus grand (environ) et qui fait 127.756 états (IMC), pour obtenir enfin une CTMC de 14.948 états, un graphe plus petit que les deux précédents.

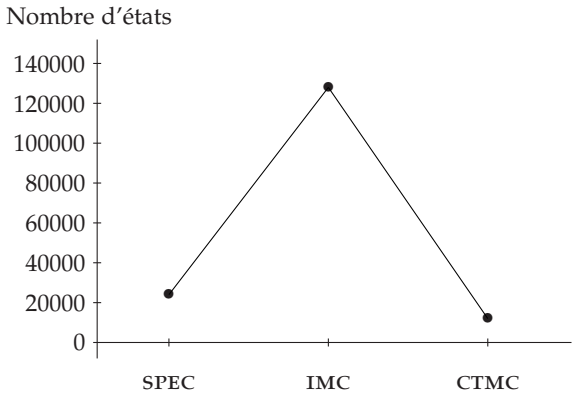


FIG. 5.6 – *Variation de la taille de l'espace d'états entre le comportement fonctionnel et stochastique*

Cette variation entre la taille des graphes fonctionnels et stochastiques est un résultat naturel de l'application du principe d'insertion des taux markoviens dans la spécification LOTOS, qui consiste en la réalisation de rendez-vous sur la porte LATENCY suite à chaque synchronisation sur la porte ACTION, augmentant ainsi de façon saisissante la taille de l'espace d'états des IMC.

Le problème d'explosion d'espace d'états a empêché l'étude de l'impact du nombre de paquets sur la latence globale, dès lors, il était impossible de confirmer ou d'infirmer la conclusion des mesures expérimentales (*le nombre de paquets n'impacte pas la latence*).

• ***Analyse des indices de performances***

Le tableau 5.5 illustre les latences moyennes d'échange de message dans le cas du protocole de cohérence de caches *A*.

Topologie	Paquets	*LM* (μs)
$topology_0$	1	1,4216
	2	1,3021
	3	1,2643
$topology_1$	1	4,6921
	2	4,2291
	3	4,1189
$topology_2$	1	7,8603
	2	7,0794
	3	6,9026

TAB. 5.5 – *Les latences globales du* ping-pong *dans le cas du protocole de cohérence de caches A, évaluées par BCG_STEADY*

La différence (moyennant le nombre de paquets) entre les latences évaluées par BCG_STEADY et celles mesurées expérimentalement est de 12,05% pour $topology_0$, 36,51% pour $topology_1$ et 47,06% pour $topology_2$. Cette différence est fondamentalement une conséquence de l'abstraction de la gestion des conflits et la modélisation atomique des accès.

Nous considérons ces résultats comme acceptables pour deux principales raisons :

1. Le pourcentage de différence ne dépasse pas 50%.
2. Le but de l'étude n'est pas de construire un modèle permettant la prédiction avec exactitude absolue des mesures expérimentales, mais de montrer qu'un modèle assez abstrait du comportement d'un algorithme MPI sur une machine CC-DSM possède suffisamment de potentiel pour comparer les différentes topologies et obtenir des latences cohérentes par rapport à la distance entre les processeurs (plus les processeurs sont distants, plus la latence augmente).

Le tableau 5.6 illustre les latences globales dans le cas du protocole de cohérence de caches *B*.

Topologie	Paquets	*LM* (μs)
$topology_0$	1	2,2100
	2	1,9270
	3	1,8544
$topology_1$	1	6,1682
	2	5,3920
	3	5,2159
$topology_2$	1	10,2451
	2	8,9548
	3	8,6674

TAB. 5.6 – *Les latences globales du* ping-pong *dans le cas du protocole de cohérence de caches B, évaluées par BCG_STEADY*

Les valeurs des latences illustrées dans le tableau précédent montrent que le protocole *A* est plus performant que le protocole *B* (selon la définition 5.2). Ceci est justifié par la présence de séquences d'accès en lecture suivie d'accès en écriture de la même variable partagée (voir la section 1.1.2).

Par exemple, dans la primitive *send*, les instructions 27 et 28 se traduisent pas deux accès à la variable *Incoming_Head_Ptr*, le premier en lecture et le second en écriture :

```
ACTION  ! Pi ! Incoming_Head_Ptr(Pj)  ! Load  ! Null_Memory
        ? latency :Latency_Value ;
ACTION  ! Pi ! Incoming_Head_Ptr(Pj)  ! Store ! Val  ? latency :Latency_Value ;
```

Le tableau 5.7 donne le nombre de *miss* effectués sur les variables de type paquet, pointeur et verrou. On constate qu'il y a en moyenne 14 *miss* dans un échange de message dans le cas du protocole *A*, dont 14, 28% pour les paquets, 50% pour les pointeurs et 35, 72% pour les verrous. Alors que dans le cas du protocole *B*, il y a en moyenne 17 *miss*, dont 17, 65% pour les paquets, 52, 94% pour les pointeurs et 29, 41% pour les verrous.

Ces résultats montrent que plus de 84% du temps est consacré à la *gestion de l'échange* (mise à jour des variables pointeurs et verrous) qu'à *l'échange effectif* de paquet. Ce qui justifie par conséquent la *non-performance* de l'algorithme.

	Pointeur	Verrou	Paquet
Protocole *A*	7	5	2
Protocole *B*	9	5	3

TAB. 5.7 – *Nombre moyen de* miss *effectués sur les variables de l'algorithme de* ping-pong *pendant l'échange d'un message*

• ***Comparaison avec d'autres primitives* send *et* receive**

Nous avons comparé les performances de l'algorithme *ping-pong* utilisant les primitives présentées dans la section 3.3.1 avec un autre algorithme de *ping-pong* basé sur le même principe d'échange de message mais en utilisant des primitives (*send* et *receive*) différentes.

On note par *ping-pong2*, *send2* et *receive2* le nouvel algorithme *ping-pong* et ses primitives respectivement, afin de les distinguer de l'algorithme 3.3.1.

Les primitives utilisées dans l'algorithme *ping-pong2* remplacent les listes par des tampons sans verrous, exploitant le fait que ces listes ont un seul lecteur (algorithmes similaires à ceux décrits dans [34]).

L'absence de la notion de "*nombre de paquets disponible initialement*" utilisée dans l'algorithme *ping-pong* a permis d'éviter le problème d'explosion combinatoire dans la génération des comportements fonctionnels et stochastiques de l'algorithme *ping-pong2*. En effet, l'évaluation des indices des performances de l'algorithme *ping-pong2* a nécessité uniquement la variation de la topologie et du protocole de cohérence de caches.

Le tableau 5.8 indique la taille de l'espace d'états des graphes BCG spécifiant le comportement fonctionnel (SPEC) et stochastique (IMC et CTMC) de l'algorithme *ping-pong2*. On constate que le plus grand graphe stochastique est une IMC qui fait 600 états seulement.

Topologie	Type	Protocole A		Protocole B	
		États	Transitions	États	Transitions
$topology_0$	SPEC	74	148	68	136
	IMC	576	1152	501	1002
	CTMC	187	374	153	306
$topology_{1,2}$	SPEC	74	148	68	136
	IMC	600	1.200	525	1.050
	CTMC	200	400	166	332

TAB. 5.8 – *La taille de l'espace d'états du comportement stochastique de l'algorithme de* ping-pong2

Les latences d'échange de message de l'algorithme *ping-pong2* présentées dans le tableau 5.9, montrent bien la bonne performance de cet algorithme par rapport à celle du *ping-pong*.

Topologie	LM (μs)	
	Protocole A	Protocole B
$topology_0$	0,6107	0,7444
$topology_1$	1,6186	1,8976
$topology_2$	2,6791	3,1345

TAB. 5.9 – *Latences moyennes d'un échange de message réalisé par le* ping-pong2 *évaluées par BCG_STEADY*

La différence entre les latences globales de l'algorithme *ping-pong2* et les latences mesurées expérimentalement de l'algorithme *ping-pong* dépasse les 50% dans toutes les topologies, elle est de 84,36% dans le cas de la $topology_0$ et de 60,30% et 51,04% dans la $topology_1$ et $topology_2$ respectivement.

Cette différence se justifie par le nombre *miss* effectués durant un échange de message. Le nombre total des *miss* effectués dans cet algorithme est égal à 6 (voir le tableau 5.10) alors que l'algorithme *ping-pong* effectue en moyenne 14 *miss* avec le protocole A (différence de 80%) et 17 *miss* avec le protocole B (différence de 95,65% environ).

Notons également que, dans l'algorithme *ping-pong2* 66,66% seulement des *miss* sont réalisés sur les variables de gestion (pointeurs), par rapport à 85,03% pour l'algorithme *ping-pong*.

	Pointeur	Paquet
Protocole *A* et *B*	4	2

TAB. 5.10 – *Nombre moyen de* miss *effectués sur les variables de l'algorithme de* ping-pong2 *pendant l'échange d'un message*

5.3.2 Algorithmes barrière

Le nombre de processus participant à l'exécution d'un algorithme de barrière est un paramètre *fondamental* pour l'analyse de son indice de performances. Cependant, il représente également la cause *fondamentale* de l'explosion combinatoire de l'espace d'états des modèles fonctionnels et stochastiques des algorithmes.

Nous présentons dans ce qui suit la taille des graphes BCG spécifiant les comportements stochastiques ainsi que l'indice de performance des algorithmes de barrières (*tournament* et *combining tree*) que nous avons présentés dans la section 3.3.

Les résultats des performances sont obtenus par l'application de la méthode présentée dans la section 5.1 pour la génération du comportement stochastique et l'utilisation de BCG_STEADY pour l'évaluation des latences. L'augmentation rapide de la taille de l'espace d'états à permis d'effectuer l'analyse pour au plus 3 processus.

• **La primitive barrière *tournament***

Le tableau 6.1 indique la taille de l'espace d'états des comportements stochastiques de l'algorithme barrière *tournament* pour un nombre de processus allant de 2 à 4.

La différence entre la taille de l'espace d'états des graphes IMC et CTMC est très importante, elle aller jusqu' un facteur 200. Ceci a provoqué une explosion combinatoire prématurée étant donné le nombre restreint des processus participant à l'exécution de la barrière (maximum 4).

Processus	Topologie	Type	États	Transitions
2	$topology_0$	IMC	906	1.812
		CTMC	156	312
	$topology_{1,2}$	IMC	906	1.812
		CTMC	163	326
3	$topology_0$	IMC	18.529	55.587
		CTMC	1.643	4.872
	$topology_{1,2}$	IMC	18.529	55.587
		CTMC	1.802	5.347
4	$topology_0$	IMC	286.536	1.146.144
		CTMC	13.516	52.650
	$topology_{1,2}$	IMC	286.536	1.146.144
		CTMC	14.845	57.879
	$topology_3$	IMC	286.536	1.146.144
		CTMC	14.245	55.513

TAB. 5.11 – *La taille de l'espace d'états du comportement stochastique de l'algorithme de* tournament

Le tableau 5.12 présente la latence globale de la primitive *tournament* en fonction du nombre de processus et lde a topologie du système.

L'algorithme de *tournament* montre une certaine performance quand tous les processus participant à son exécution se trouvent dans le même nœud (cas de la $topology_0$). En effet, la latence globale ne dépasse pas $0,84\ \mu s$ pour une exécution entre 4 processus.

La latence globale de la barrière *tournament* est sensible au nombre de processus. Par exemple, elle augmente de 0,25% en passant de 2 à 3 processus dans les différentes topologies. En revanche, dans certains cas, le bon choix de la topologie peut améliorer ce que le nombre de processus a dégradé en terme de performances. Par exemple, le *tournament* entre 4 processus dans une $topology_0$, $topology_1$ ou $topology_3$ est plus performant que le *tournament* entre 3 processus dans une $topology_2$.

Processus	Topologie	LM (μs)
2	$topology_0$	0,4193
	$topology_1$	1,0573
	$topology_2$	1,7459
3	$topology_0$	0,6949
	$topology_1$	1,7627
	$topology_2$	2,9125
4	$topology_0$	0,8327
	$topology_1$	2,1572
	$topology_2$	3,5722
	$topology_3$	2,5696

TAB. 5.12 – *Latences moyennes de synchronisation par la barrière* tournament

La variation du protocole de cohérence de caches n'a aucun impact sur le comportement de la barrière *tournament*, on obtenait en effet les mêmes graphes BCG (fonctionnel et stochastique) et les mêmes indices de performance dans le cas du protocole A et B. Ceci s'explique par l'absence d'accès en lecture suivi d'un autre en écriture pour la même variable partagée dans le comportement de *tournament* (voir l'exemple 1.1).

Il nous est impossible d'en conclure la *performance* ou la *non-performance* de la barrière *tournament* vu les indices de performances obtenus pour un nombre relativement petit de processeurs (≤ 4) par rapport à la machine réelle (≥ 16).

• **Algorithme barrière *combining tree***

En vue de la taille de l'espace d'états des graphes BCG décrivant le comportement fonctionnel de la barrière *combining tree* (atteignant 498.896 états pour 4 processus), la génération du comportement stochastique fut impossible pour un nombre de processus ≥ 4.

Le tableau 5.13 indique la taille des graphes BCG de IMC et CTMC de la barrière *combining tree* dans le cas de protocole de cohérence de caches A et B.

Processus	ID_Tree	Topologie	Type	Protocole de caches *A*		Protocole de caches *B*	
				États	Transitions	États	Transitions
2	1	$topology_0$	IMC	4.472	8.944	2.669	5.338
			CTMC	683	1.366	429	858
		$topology_{1,2}$	IMC	4.506	9.012	2.703	5.406
			CTMC	707	1.414	453	906
3	1	$topology_0$	IMC	342.853	1.028.559	114.956	344.868
			CTMC	25.380	75.798	9.116	27.170
		$topology_{1,2}$	IMC	351.393	1.054.179	117.550	352.650
			CTMC	27.030	80.704	9.958	29.685
	2	$topology_0$	IMC	248.572	745.716	124.196	372.573
			CTMC	25.380	75.798	9.116	27.170
		$topology_{1,2}$	IMC	261.431	784.293	126.005	378.015
			CTMC	12.527	37.184	6.333	18.733

TAB. 5.13 – *La taille de l'espace d'états du comportement fonctionnel et stochastique de l'algorithme barrière* combining tree

On constate que le taux de différence entre la taille des différents graphes BCG dépasse dans la plus part des cas les 100%. La différence moyenne entre la taille :

1. des graphes stochastiques par rapport aux deux protocoles de cohérence de caches est en moyenne de 72, 70%
2. de la IMC et la taille de CTMC est en moyenne de 163, 44%
3. des IMC par rapport au nombre de processus est en moyenne de 192, 55%
4. des CTMC par rapport au nombre de processus est en moyenne de 183, 42%

Le nombre de processus et leur organisation en arbre combinatoire ont un impact important sur la taille des graphes stochastiques, notamment les IMC.

Le tableau 5.14 illustre les indices de performance de la barrière *combining tree*.

Processus	ID_Tree	Topologie	*LM* (μs)	
			Protocole *A*	Protocole *B*
2	1	$topology_0$	0,5393	0,3576
		$topology_1$	1,2551	0,9723
		$topology_3$	2,0631	1,6120
3	1	$topology_0$	1,0834	0,7332
		$topology_1$	2,4785	1,9191
		$topology_3$	4,0675	3,1751
	2	$topology_0$	0,5945	0,4545
		$topology_1$	1,3235	1,1106
		$topology_3$	2,1585	1,8158

Tab. 5.14 – *Latences moyennes de synchronisation par la barrière* combining tree

Cependant, cette fois, le protocole de cohérence de caches *B* a joué un rôle avantageux, en générant des graphes IMC et CTMC remarquablement plus petits que ceux générés pour le protocole *A*.

L'impact positif du protocole *B* ne s'est pas limité à la taille des graphes stochastiques, mais aussi à la latence globale de la primitive, en faisant de la barrière *combining tree* un algorithme plus performant avec le protocole de cohérence de caches *B*.

Conclusion du chapitre

Dans ce chapitre, nous présentons la procédure d'évaluation d'indice de performance de notre système. On part d'un modèle décrivant le comportement fonctionnel du système dont les propriétés de bon fonctionnement ont été vérifiées, nous procédons par la suite à l'insertion des aspects temporelles (latences et délais d'attente) pour effectuer en fin l'analyse markovienne et obtenir l'indice de performance.

La définition d'une approche permettant l'insertion des aspects temporels aux bons endroits dans la spécification LOTOS et assurant la réussite de l'analyse stochastique est une mission non triviale. L'approche doit prendre en compte l'interprétation et l'impact du parallélisme et du protocole de dans le modèle stochastique.

Dans l'étude stochastique de l'algorithme *ping-pong*, la différence (moyennant le nombre de paquets échangés) entre l'indice de performance évalué par BCG_STEADY et celui mesuré expérimentalement ne dépasse pas 50%, elle est principalement due à l'abstraction de la gestion des conflits et la modélisation atomique des accès.

l'étude de performances des algorithmes de barrière permettent une comparaisons entre les différentes primitives de barrière pour un nombre très restreint (maximum 3) de processus en raison de l'explosion combi-

natoire de l'espace d'états des modèles fonctionnels et stochastiques des algorithmes.

Passage à l'échelle

6

Sommaire

Chacun des algorithmes MPI que nous avons étudiés, possède un ou plusieurs *facteurs* favorisant l'explosion combinatoire de l'espace d'états du modèle spécifiant son comportement fonctionnel ou stochastique. Par exemple, le nombre de paquets dans l'algorithme de *ping-pong* et l'organisation des processus en arbre combinatoire dans la barrière *combining tree*. En revanche, tous les algorithmes de barrière partagent un facteur en commun : le *nombre de processus* participant à l'exécution de la barrière.

En effet, la nature de algorithme n'est pas l'unique raison de d'explosion d'espace d'états, puisque le principe d'insertion des taux markoviens dans la spécification LOTOS favorise la croissance de la taille de l'espace d'états, par conséquent la génération du comportement fonctionnel ne garantit pas toujours la génération du comportement stochastique (IMC et CTMC).

L'insertion des taux markoviens dans la spécification LOTOS implique des rendez-vous sur la porte LATENCY après chaque synchronisation sur les portes ACTION et WAIT, ce qui engendre une augmentation remarquable de la taille de l'espace d'états.

Par exemple, soit *size*(G) la taille de l'espace d'états du modèle *G*. Dans le cas du protocole de cohérence de cache *A* et moyennant les différentes topologies ainsi que le nombre de paquets et le nombre de processus, nous avons :

- *ping-pong* :
  $size(\text{IMC}) \approx 5\ size(\text{SPEC})$
  $size(\text{IMC}) \approx 6\ size(\text{CTMC})$
  $size(\text{CTMC}) \approx size(\text{SPEC})$
- *tournament* :
  $size(\text{IMC}) \approx 22\ size(\text{SPEC})$
  $size(\text{IMC}) \approx 11\ size(\text{CTMC})$
  $size(\text{CTMC}) \approx 1{,}6\ size(\text{SPEC})$
- *combining tree* :
  $size(\text{IMC}) \approx 15\ size(\text{SPEC})$
  $size(\text{IMC}) \approx 12\ size(\text{CTMC})$
  $size(\text{CTMC}) \approx 1{,}25\ size(\text{SPEC})$

La différence entre la taille du comportement stochastique IMC et les comportements SPEC et CTMC est incontestable. Quant à la différence entre la taille du comportement fonctionnel SPEC et stochastique CTMC, elle est relativement raisonnable.

La procédure de génération de CTMC 5.1 nécessite un passage obligatoire par la génération d'un graphe intermédiaire (IMC)qui peut être excessivement grand par rapport au graphe initial (SPEC) et au graphe résultant (CTMC).

Trouver une solution pour contourner le problème d'explosion combinatoire d'états consiste à pousser la génération des comportements stochastiques des différents algorithmes pour un nombre important de *paquets* dans le cas de l'algorithme de *ping-pong* 3.3.1 et de processus dans le cas des algorithmes de *barrière*, de telle sorte que l'on puisse générer graphes initiaux (SPEC) intermédiaires (IMC) et finaux (CTMC) avec les ressources de calcul disponibles.

Nous avons étudié principalement 3 solutions. La première consiste en l'abstraction de toutes les informations qui n'influencent pas la latence globale. La seconde solution tente la réduction de l'espace d'états des graphes intermédiaires par la génération compositionnelle. La troisième solution propose une nouvelle méthode d'insertion des taux markoviens dans la spécification LOTOS sans l'ajout de nouveaux rendez-vous.

Nous avons choisi l'algorithme de barrière *centralized* pour la présentation de ces solutions, pour la simplicité de son comportement fonctionnel, ainsi que le nombre modeste de variables utilisés : deux variable partagées *sense* et *count* et une variable privée *local_sense* pour chaque processus.

6.1 Plus d'abstraction

Afin d'augmenter l'efficacité de la réduction stochastique, nous avons effectué l'abstraction,

1. de toutes les transitions markoviennes représentant les latences d'accès aux variables privées ;
2. de tous les préfixes des transitions markoviennes. Elles sont par conséquent de la forme "*rate* λ", sauf pour les transitions qui marquent le début de la mission de l'algorithme, qui sont de la forme "START; *rate* λ" ;
3. de toutes les transitions markoviennes représentant les latences d'accès internes, c'est à dire les transitions de la forme "*rate* λ_{C_INT}".

La première abstraction est justifiée par l'insensibilité de la latence globale aux latences d'accès aux variables privées, car à part le premier accès effectuant un transfert depuis la mémoire, tous les autres accès qui suivent effectuent des transferts internes possédant des latences très petites.

La seconde abstraction permet de favoriser la réduction stochastique comme le prouve l'exemple 6.1.

Exemple 6.1 *L'absence de préfixe dans les taux markoviens des transitions du comportement E, illustré dans la figure 6.1, a permis sa minimisation en E_{red} (par l'application de l'axiome A7 du tableau 2.4). Par contre la présence de préfixes différents (a et b) dans les taux markoviens des transitions sortantes de l'état initial dans le comportement E′ a empêché sa minimisation et par conséquent le graphe E'_{red} résultant de la réduction de E′ est identique à E′.*

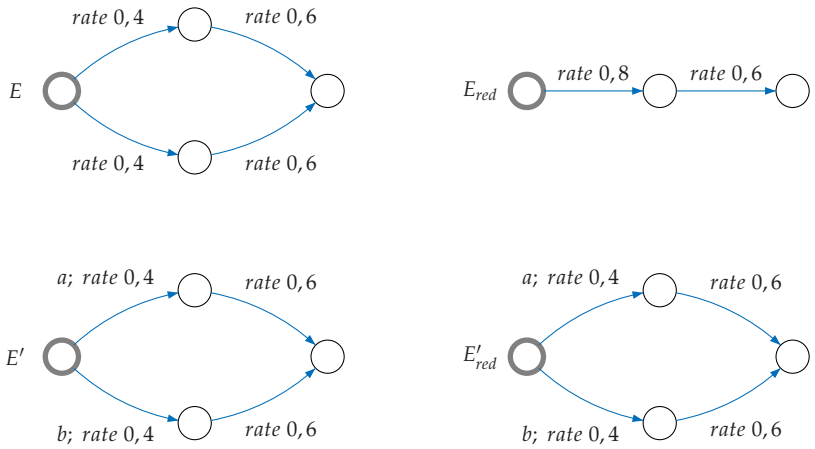


FIG. 6.1 – *Exemples de réduction stochastique*

Remarque 6.1 *Le débit moyen DM de la transition "START; rate λ" représente la somme des débits de la mission de l'algorithme de tous les processus participant à son exécution. Pour obtenir la latence globale LM par processus il faut alors diviser d'abord le DM sur le nombre de processus, ou encore appliquer la formule :*

$$LM = \frac{nb\_proc}{DM}$$

La troisième abstraction, qui consiste à éliminer toutes les latences d'accès internes, est justifiée par le fait que la latence d'un transfert interne est négligeable par rapport aux latences des transferts caches et mémoire. Cependant, cette abstraction n'est pas toujours possible pour tous les algorithmes en raison du non-déterminisme qu'elle engendre, qui est finalement dû aux demandes d'accès simultanées à une même variable partagée par plusieurs processus. Pour plus de clarté, nous proposons l'exemple suivant :

Exemple 6.2 *On suppose que la latence d'un transfert interne est nulle.*

Soient deux processus P_0 et P_1 exécutant l'instruction suivante à partir de l'instant t_0 :

repeat until *var = true*

sachant qu'initialement la variable var se trouve en mémoire et son contenu est positionné à $false$.

À l'instant t_n ($n > 0$) le processus P_2 accède en écriture à la variable var et positionne son contenu à true.

Mis à part le premier accès qui implique un transfert depuis la mémoire, tous les accès de P_0 et P_1 réalisés avant t_n ont engendré des transferts internes.

À l'instant t_n quand le processus P_2 accède en écriture à var il invalide l'état de var dans les caches de P_0 et P_1. Par conséquent, le prochain accès de P_0 et P_1 nécessite un transfert depuis le cache de P_2.

Comme les précédents accès de P_0 et de P_1 étaient d'une latence nulle (transfert interne), leur demandes d'accès à var pour une dernière lecture arrivent en même temps à P_2 le mettant ainsi dans une situation de choix non-déterministe : servir P_0 en premier et par conséquent P_1 accède à la donnée en mémoire ou inversement.

Le tableau 6.1 illustre la taille des graphes IMC et CTMC de l'algorithme barrière *centralized* avant et après l'abstraction des latences d'accès aux variables privées ainsi que les informations sur les transitions markoviennes, dans le cas du protocole de cohérence de caches *A*.

La différence en moyenne entre les IMC avant et après abstraction est de 124% environ et, elle est de 111% entre les CTMC. Certes elle est remarquable et on peut même espérer l'atteinte d'un nombre important de processus, mais avec la procédure de génération de CTMC décrite en section 5.1, l'application de l'abstraction ne peut avoir lieu qu'après une génération complète de la IMC : abstraction tardive car l'explosion a déjà eu lieu à la génération de l'IMC.

Mais nous allons voir dans ce qui suit que l'intérêt de ces abstractions apparaît bien durant la génération compositionnelle.

Processus	Topologie	Type	États	
			Avant abstraction	Après abstraction
2	$topology_0$	IMC	1.515	523
		CTMC	111	39
	$topology_{1,2}$	IMC	1.553	545
		CTMC	151	53
3	$topology_0$	IMC	46.547	10.497
		CTMC	643	179
	$topology_{1,2}$	IMC	48.796	11.280
		CTMC	1.999	582
4	$topology_0$	IMC	1.289.205	182.237
		CTMC	2.731	604
	$topology_{1,2}$	IMC	1.371.503	200.061
		CTMC	10.966	2.474

TAB. 6.1 – *Comparaison des graphes stochastiques de la barrière* centralized *avant et après abstraction (protocole A)*

6.2 GÉNÉRATION COMPOSITIONNELLE

La génération compositionnelle consiste à générer le comportement du système à partir de la composition de ses sous-comportements générés indépendamment.

Notre but principal dans l'application de la génération compositionnelle est de pouvoir éviter l'explosion intermédiaire des graphes, notamment lors de l'insertion des taux markoviens.

Le schéma de synchronisation spécifiant les aspects temporels de l'algorithme de barrière *centralized* dans une architecture CC-DSM

$$\begin{aligned} \text{CENTRALIZED} \quad = \quad & \Big|\Big|\Big|_{i=0}^{n-1} \left(\text{BARRIER}(\text{P}_i) \; ||[\text{ACTION, WAIT}]|| \; \text{LATENCY\_PROC}(\text{P}_i)\right) \\ & ||[\text{ACTION}]|| \\ & \text{TRANSFER}(\text{M, C}) \end{aligned}$$

est par définition une composition de sous-comportements :

1. Le processus BARRIER(P_i) spécifie le comportement de la primitive de barrière *centralized* réalisée par P_i.
2. Le processus LATENCY_PROC(P_i) permet l'insertion des taux markoviens de tous les accès ainsi que d'autres traitements comme les attentes, réalisés par P_i.

3. Le processus TRANSFER(M, C) assure la gestion de la mémoire et les caches.

Il est également possible de décomposer encore le comportement généré par le processus TRANSFER en sous-comportements spécifiant la gestion de la mémoire, les caches et les variables de façon indépendante.

Remarque 6.2 *Seul le comportement du processus* TRANSFER *offre la possibilité de la génération du comportement en mémoire et/ou en caches des différentes variables de façon indépendante. En effet, la décomposition du comportement du processus* BARRIER *par variable détruit la structure de l'algorithme en terme d'ordre des accès et l'utilisation d'un mécanisme permettant de respecter l'ordre des accès aux variables engendre souvent des sous-comportements plus grands en terme d'espace d'états que le comportement global de* BARRIER.

La génération individuelle des sous-comportements directement à partir de leur définition en LOTOS sans l'utilisation d'interfaces engendre des séquences de transitions qui n'appartiennent pas au comportement originel du système. Nous appelons ces séquences*parasites*, car d'une part, elles ne peuvent être éliminées que par le rassemblement de tous les sous-comportements par des synchronisations et, d'une autre part, leur présence provoque souvent l'explosion de l'espace d'états des sous-comportements.

Une séquence parasite peut être une simple transition étiquetée par une action obsolète. Par exemple, l'action suivante désigne un accès en écriture à la variable *count* par un processus d'identificateur nul (*Nil_Processor*) :

ACTION ! Nil_Processor ! count ! Store ! 4 ! C_FSB

Nous avons éliminé ce genre de séquences parasites en imposant des *contraintes sur les actions*.

Une séquence parasite peut correspondre également à une suite d'accès qui ne respecte pas l'ordre défini dans l'algorithme. Par exemple, il n'y a jamais d'accès en écriture à la variable *count* suivi d'un accès en lecture à la variable *sense* par le même processus :

ACTION ! P_0 ! count ! Store ! 4 ! C_FSB

ACTION ! P_0 ! sense ! Load ! False_Val ! C_FSB

Nous avons éliminé ce genre de séquences parasites en imposant des *contraintes sur les comportements*.

Les contraintes sur les actions

On entend par les *contraintes sur les actions* des gardes imposées sur les offres de réception ("?") d'un rendez-vous. Ces contraintes permettent l'élimination d'actions qui n'appartiennent pas l'ensemble d'actions du comportement d'origine.

Nous avons défini pour chaque sous-comportement une *interface-action* sous forme d'un processus LOTOS comportant toutes les contraintes sur

les actions. Ainsi, le processus LOTOS spécifiant un sous-comportement et respectant les contraintes imposées sur toutes les actions, est obtenu par une composition totalement synchronisée du processus spécifiant le sous-comportement sans contraintes et son processus interface-action. Par exemple, le processus ROOT_BARRIER spécifie le sous-comportement de BARRIER respectant toutes les contraintes imposées par INTERFACE_BARRIER :

ROOT_BARRIER = BARRIER |[ACTION, WAIT]| INTERFACE_BARRIER

Le tableau 6.2 décrit les différents sous-comportements de l'algorithme barrière *centralized*, ainsi que leurs interfaces.

Processus	Interface-action	S/comportement	Description
BARRIER	INTERFACE_BARRIER	ROOT_BARRIER	Comportement de la primitive de barrière réalisé par un seul processus
LATENCY	INTERFACE_LATENCY	ROOT_LATENCY	Comportement des latences d'accès aux variables partagées et les latences d'attente pour un seul processus.
TRANSFER	INTERFACE_TRANSFER	ROOT_TRANSFER	Comportement de toutes les variables partagées en mémoire et en caches.
	INTERFACE_CACHE	ROOT_CACHE	Comportement de toutes les variables partagées en caches.
	INTERFACE_MEMORY	ROOT_MEMORY	Comportement de toutes les variables partagées en mémoire.
	INTERFACE_LOCAL_SENSE	ROOT_LOCAL_SENSE	Comportement de la variable privée *local_sense* en mémoire et en caches.
	INTERFACE_SENSE	ROOT_SENSE	Comportement de la variable partagée *sense* en mémoire et en caches.
	INTERFACE_COUNT	ROOT_COUNT	Comportement de la variable partagée *count* en mémoire et en caches.

TAB. 6.2 – *Sous-comportement de la barrière* centralized

Le processus INTERFACE_BARRIER décrit ci-dessous est un exemple d'interface-action pour le processus BARRIER dans le cas d'une $topology_0$.

```
process   INTERFACE_BARRIER [ACTION, WAIT] : noexit :=
 ACTION   ? P :ID_Processor   ? adr :Address   ? op :ID_Action
          ? val :Memory_Value   ? latency :Latency_Value
  [
   (P <> Null_Proc) and
   (
    ( ( adr == local_sense(P) ) and
      ( (op == Load) or (op == Store) ) and
      ( (val == True_Val) or (val == False_Val) ) and
      ( (latency == C_INT) or (latency == M_FSB_1) )
    ) or
    ( ( adr == sense ) and
      ( (op == Load) or (op == Store) ) and
      ( (val == True_Val) or (val == False_Val) ) and
      ( (latency == C_INT) or (latency == C_FSB) or (latency == C_FSB_1))
    ) or
    ( ( adr == count ) and
      ( ( (op == Store) and (val = Count_Value)) or
        ( (op == Fetch_and_Decrement) and (val <> True_Val) and) ) and
        ( (val <> False_Val) and (Less(val, Count_Val) == True_Val) ) ) and
      ( (latency == C_INT) or (latency == C_FSB) or (latency == C_FSB_1))
    )
   )
  ];
 INTERFACE_BARRIER [ACTION, WAIT]
  []
 WAIT   ? p :ID_Processor   ? val :of  Nat
  [ (p <> Null_Proc) and (val == 0) ];
 INTERFACE_BARRIER [ACTION, WAIT]
endproc
```

Le rendez-vous sur la porte ACTION a lieu si :

1. L'identificateur de processus est différent de nul
2. et, si l'accès concerne la variable privée *local_sense*, il sera uniquement pour une lecture ou une écriture et la valeur lue ou écrite est de type booléen (*true* ou *false*). Le transfert peut être interne ou depuis la mémoire locale.
3. sinon, si l'accès concerne la variable partagée *sense*, il doit vérifier les mêmes contraintes que l'accès à la variable *local_sense* sauf, pour les transferts, où il est également autorisé de faire des transferts depuis des mémoires et des caches distants.
4. sinon, si l'accès concerne la variable partagée *count*, il sera pour une écriture et la valeur écrite doit être égale au nombre de processus participant à l'exécution de la barrière (*Count_Value*), sinon pour une lecture-écriture atomique (*Fetch_and_Decrement*) et la valeur retournée doit être inférieure à *Count_Value*. Le transfert peut être depuis la mémoire ou le cache (interne, local ou distant).

Le rendez-vous sur la porte WAIT modélise le début de mission de l'al-

gorithme uniquement, par conséquent, l'identificateur de processus doit être différent de nul et la valeur *val* doit être 0.

La définition des processus interfaces pour contraindre les actions nécessite une très bonne connaissance du fonctionnement de l'algorithme : le type des variables (partagées ou privées), le type d'accès pour chaque variable (lecture, écriture, ...), le type du contenu de chaque variable (booléen, entier naturel, valeur maximale, valeur minimale, ...).

Une interface-action efficace doit couvrir, plus au moins exactement, toutes les actions du système, car si elle autorise des actions en plus, elle provoque une augmentation de la taille de l'espace d'états et, en contre partie, si elle autorise un nombre très strict d'actions au point d'éliminer des séquences non-parasites, elle implique la génération d'un comportement différent de celui prévu initialement.

C'est pour ces raisons d'ailleurs que notre choix s'est porté sur l'algorithme barrière *centralized* pour l'étude de la génération compositionnelle. Contrairement aux autres algorithmes de barrière et même de *ping-pong*, l'algorithme *centralized* utilise uniquement deux variables partagées et une seule variable privée, et tous les processus participant à son exécution possèdent un comportement symétrique (par exemple, il y pas de notion de processus père ou fils comme dans les algorithmes de barrière *combining tree* et *tournament*).

Le tableau 6.3 représente une comparaison entre la taille du graphe généré à partir du processus BARRIER et celui généré à partir du processus ROOT_BARRIER dans le cas de la $topology_0$ et du protocole de cohérence de caches A.

Processus	Taille	BARRIER	ROOT_BARRIER
2	États	33	12
	Transitions	1.821	44
3	États	35	12
	Transitions	2.101	47
4	États	37	12
	Transitions	2.401	50

TAB. 6.3 – *Taille du sous-comportement du processus barrier avec et sans l'utilisation d'interface*

La différence entre la taille des sous-comportements générés avec et sans l'interface est très importante et elle augmente en fonction du nombre de processus participant à l'exécution de la barrière. Elle est de 93, 33% pour deux processus, 97, 89% pour trois processus et 102, 04% pour quatre processus. Notons également une explosion en nombre de transitions au niveau des sous-comportements générés sans interface.

Les contraintes sur le comportement

Seuls les sous-comportements générés à partir du processus TRANSFER nécessitent des interfaces contraignant leur comportement, car ils génèrent tous les ordres possibles d'accès aux variables partagées, y compris ceux définis par l'algorithme, entrainant ainsi une augmentation considérable de leur espace d'états.

Pour ôter toutes les séquences parasites, nous avons utilisé le sous-comportement généré à partir du processus BARRIER comme interface comportementale, il s'agit en effet de projeter son comportement sur les sous-comportements du processus TRANSFER selon le principe de la génération *semi-compositionnelle* [59, 24, 57, 60].

Cette projection est assurée dans la boîte à outil CADP par l'outil PROJECTOR[1].

Dans un script svl, l'appel à cet outil est effectué au moyen des opérateurs de semi-composition $-||$ et $-|[\ ]||$ où :

$-||$: signifie l'application de contraintes sur toutes les actions.

$-|[a_0, \ldots, a_n]||$: signifie l'application de contraintes sur l'ensemble d'actions $\{a_0, \ldots, a_n\}$.

Définition 6.1 *Soient M_1 et M_2 deux systèmes de transitions étiquetées et* PROJ *un ensemble d'actions. Nous avons :*
$M_1 - |[\text{PROJ}]| M_2$ est un système de transitions étiquetées donné par le quadruplet (S, A, T, s_0) avec :

- *S est un ensemble d'états s_1 de M_1 tel que pour un certain état s_2 de M_2, (s_1, s_2) est accessible dans $M_1 - |[\text{PROJ}]| M_2$.*
- *T est un ensemble de transitions (s_1, a, s_1') de M_1, tel que pour certains états s_2, s_2' de M_2 il existe une transition $((s_1, s_2), a, (s_1', s_2'))$ dans $M_1 - |[\text{PROJ}]| M_2$.*
- *s_0 est l'état initial de M_1.*

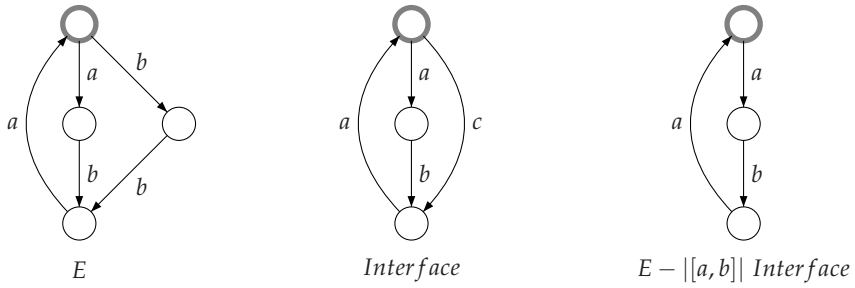


FIG. 6.2 – *Exemple de projection*

Le tableau 6.4 illustre la taille du sous-comportement généré à partir du processus ROOT_TRANSFER avec et sans l'utilisation de l'interface comportementale, qui est la composition parallèle du sous-comportement (généré à partir du processus ROOT_BARRIER) des différents processus participant à l'exécution de l'algorithme.

[1] http://www.inrialpes.fr/vasy/cadp/man/projector.html

Processus	Taille	Interface comportementale	ROOT_TRANSFER	
			Avec interface	Sans interface
2	États	64	77	51
	Transitions	576	726	110
3	États	512	273	175
	Transitions	8.064	3.906	645
4	États	4.096	819	533
	Transitions	98.304	15.756	2.812
5	États	32.768	2.263	1.521
	Transitions	1.105.920	54.750	10.525
6	États	262.144	5.977	4.147
	Transitions	11.796.480	174.306	35.572

TAB. 6.4 – *Taille du sous-comportement du processus T avec et sans l'utilisation d'interface*

La taille de l'interface comportementale pour $n+1$ processus est 8 fois plus grande que la taille de l'interface comportementale pour n processus. Ce taux d'augmentation limite la génération d'une interface contenant le comportement de tous les processus participant à la barrière, réduisant ainsi l'efficacité de l'interface comportementale de contraindre les sous-comportements du processus TRANSFER.

La taille du sous-comportement généré à partir du processus ROOT_BARRIER subit une réduction de 20% en moyenne par l'application de l'interface comportementale. Un taux jugé non-suffisant quand la taille du sous-comportement sera de l'ordre des 10.000 états et plus.

D'après le tableau 6.2 qui illustre les différents sous-comportements de la barrière *centralized* qu'il est possible de générer indépendamment les un des autres, on peut définir plusieurs stratégies de composition pour établir le comportement stochastique CTMC final.

Nous présentons dans ce qui suit les différentes compositions que nous avons étudiées et, pour faciliter ceci nous utilisons les notations décrites dans le tableau 6.5 ci-dessous.

Notation	Description
`Red`(G)	Réduction modulo la bisimulation faible du graphe G.
`Gen`(PROC)	Génération du comportement du processus LOTOS PROC.
`Hide`(($a_0, \ldots, a_n$), G)	Abstraction de toutes les actions $a_0, \ldots, a_n$ dans le graphe G.
`Projc`(G_1, G_2)	Application de la projection sur toutes les portes communes.
`StRed`(G)	Réduction stochastique modulo la bisimulation faible du graphe G.
`Synch`(G_1, G_2)	Composition parallèle entre les graphes G_1 et G_2.
`ASynch`(G_1, G_2)	Composition parallèle avec synchronisation sur les actions communes entre les graphes G_1 et G_2.
W_P(i)	Rendez-vous sur la porte WAIT réalisé par P_i.
LS_P(i)	Rendez-vous sur la porte ACTION pour un accès à la variable *local_sense* réalisé par P_i.
S_P(i)	Rendez-vous sur la porte ACTION pour un accès à la variable *sense* réalisé par P_i.
C_P(i)	Rendez-vous sur la porte ACTION pour un accès à la variable *count* réalisé par P_i.

TAB. 6.5 – *Notations utilisées pour la présentation des différentes générations compositionnelles*

6.2.1 Stratégie de génération compositionnelle "$Comp_1$"

L'algorithme 6 décrit le principe de la génération compositionnelle "$Comp_1$" permettant la construction du comportement stochastique CTMC en 4 étapes :

Étape 1 (*lignes 2-7*) : génération du sous-comportement stochastique IMC de la primitive barrière ($B\_L(i)$) de tous les processus, avec l'abstraction des accès à la variable privée *local_sense*.

Étape 2 (*lignes 9-15*) : génération des interfaces comportementales.

Étape 3 (*lignes 16*) : génération du comportement de la mémoire et les caches (T) pour toutes les variables partagées avec l'application de l'interface comportementale $Interface(n-1)$ spécifiant le comportement de tous les processus participant à la barrière.

Étape 4 (*lignes 17-26*) : génération de l'IMC en $n-1$ phases avec IMC(i), un graphe comportant **uniquement** les taux markoviens des processus $n-1, n-2, \ldots, i$ (sans leurs actions interactives respectives) et les actions interactive des processus $i-1, i-2, \ldots, 0$ (sans leurs taux markoviens respectifs). Finalement la CTMC est obtenue par la minimisation stochastique de IMC(0) (*ligne 26*).

Algorithme 6 : $Comp_1$

```
   Data : i process identifier ; n number of process ;
 1 begin
 2   for i=0 ; i < n ; i++ do
 3     B(i) = Red (Gen (ROOT_BARRIER)) ;
 4     L(i) = StRed (Gen (ROOT_LATENCY));
 5     LS(i) = Red (Gen (ROOT_LOCAL_SENSE)) ;
 6     B_LS(i) = Red (Hide (LS_P(i), Synch (B(i), LS(i))));
 7     B_L(i) = StRed (Synch (B_LS(i), L(i)));
 8   end
 9   for i=0 ; i < n ; i++ do
10     if i == 0 then
11       Interface(i) = Red (Hide ((W_P(i),LS_P(i)), B(i)));
12     else
13       Interface(i) = Red (Hide ((W_P(i),LS_P(i)), ASynch (B(i),
         Interface(i − 1))));
14     end
15   end
16   T = Red (Gen (Proj (ROOT_TRANSFER, Interface(n − 1))));
17   for i=n-1 ; i ≥ 0 ; i- - do
18     if i == 0 then
19       IMC(i) =StRed (Hide ((C_P(i), S_P(i) ), Synch (IMC(i + 1),
         B_L(i))));
20     else if i == n-1 then
21       IMC(i) =StRed (Hide ((C_P(i), S_P(i) ), Proj (Synch (T, B_L(i)),
         Interface(i-1))));
22     else
23       IMC(i) =StRed (Hide ((C_P(i), S_P(i) ), Proj (Synch
         (IMC(i + 1), B_L(i)), Interface(i-1))));
24     end
25   end
26   CTMC =StRed (IMC(0));
27 end
```

Le tableau 6.6 illustre la taille de l'espace d'états des différents sous-comportements. On constate que dans la plupart des cas, la taille des IMC(i) intermédiaires est souvent supérieure à la taille de la CTMC finale.

S/Comportement	Taille	Nombre processus (n)				
		2	3	4	5	6
B_L(i)	États	34	34	34	34	34
	Transitions	62	68	74	80	86
Interface(i)	États	64	512	4.096	32.768	262.144
	Transitions	576	8.064	98.304	1.105.920	11.796.480
T	États	51	175	533	1.521	4.147
	Transitions	110	645	2.812	10.525	35.574
IMC(5)	États					19.816
	Transitions					152.421
IMC(4)	États	–	–	–	6.764	38.486
	Transitions	–	–	–	41.684	258.860
IMC(3)	États	–	–	2.092	12.000	111.722
	Transitions	–	–	9.769	63.255	657.789
IMC(2)	États	–	552	3.916	30.020	276.978
	Transitions	–	1.772	12.390	135.975	1.420.589
IMC(1)	États	114	680	6.560	60.326	292.854
	Transitions	228	1.843	22.291	245.530	1.353.458
IMC(0)	États	39	503	604	29.327	139.139
	Transitions	77	1.468	2.067	135.803	766.463
CTMC	États	39	179	604	14.757	75.013
	Transitions	77	503	2.067	68.038	404.084

TAB. 6.6 – *Taille des graphes BCG générés dans la génération compositionnelle "$Comp_1$"*

La figure 6.3 illustre la variation de la taille des graphes IMC(i) pour la barrière *centralized* entre 6 processus. Le nombre d'états augmente pour atteindre un pic de 292.854 (correspondant à la taille de l'IMC(1)), mais en lui ajoutant le comportement du processus P_0 et en appliquant la réduction stochastique on obtient un graphe 2 fois plus petit (IMC(0)).

La stratégie de génération compositionnelle 6.2.1 nous a permis la génération de CTMC pour un nombre de processus égal à 6. Un résultat jugé bon par rapport à la méthode de génération directe (décrite dans la section 5.1), mais insuffisant, car au-delà de 6 processus, une explosion d'espace d'états a lieu. Cette stratégie souffre du même problème que la génération directe 5.1 : explosion combinatoire de l'espace d'états des graphes intermédiaires.

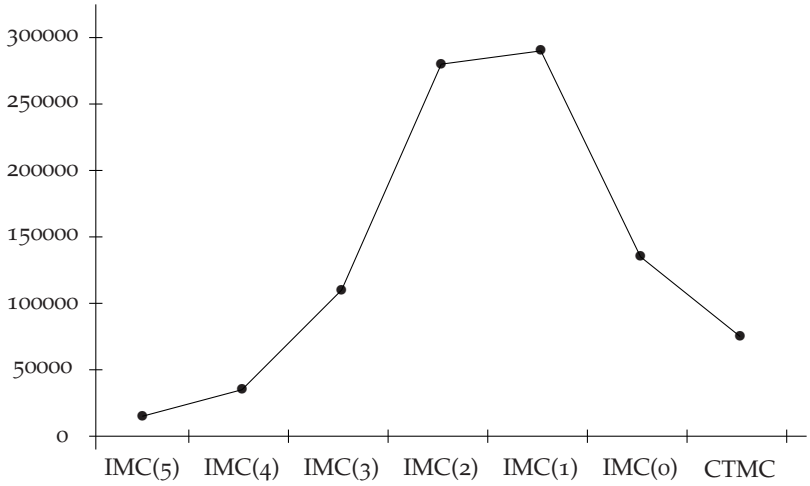


FIG. 6.3 – *La taille de l'espace d'états des IMC(i) pour un nombre de processus égal à 6*

6.2.2 Stratégie de génération compositionnelle "$Comp_2$"

Cette génération est basée sur le même principe de décomposition que celui de la $Comp_1$ mais en considérant le comportement de la mémoire et des caches par variable privée. Le comportement stochastique CTMC est construit en 4 étapes également :

Étape 1 (*lignes 2-7*) : génération du sous-comportement stochastique IMC de la primitive barrière ($B\_L(i)$) de tous les processus, avec l'abstraction des accès à la variable privée *local_sense*.

Étape 2 (*lignes 9-15*) : génération des interfaces comportementales.

Étape 3 (*lignes 16-17*) : génération du comportement de la mémoire et les caches des variables partagées variable *count* (*C*) et *sense* (*S*) en leur imposant des contraintes comportementales par l'interface $Interface(n-1)$.

Étape 4 (*lignes 18-31*) : génération de la CTMC, qui peut se faire au moyen de deux méthodes différentes :

(a) génération de IMC_C en $n-1$ phases avec IMC_C(i), un graphe comportant **uniquement** les taux markoviens pour les accès à la variable *count* ainsi que les actions interactives pour les accès à la variable *sense* des processus $n-1, n-2, \ldots, i$ et les actions interactives pour les accès aux variables *count* et *sense* des processus $i-1, i-2, \ldots, 0$. Finalement la CTMC_C est obtenue par minimisation stochastique suite à l'abstraction des actions interactives pour les accès à la variable *sens* dans la composition synchrone entre IMC_C(0) et le comportement de la variable *sense* en cache et mémoire (*ligne 30*).

(b) génération de IMC_S en $n-1$ phases avec IMC_S(i), un graphe comportant **uniquement** les taux markoviens pour les accès à la variable *sense* ainsi que les actions interactives pour les accès à la variable *count* des processus $n-1, n-2, \ldots, i$ et les actions interactives pour les accès aux variables *sense* et *count* des processus $i-1, i-2, \ldots, 0$. Finalement la CTMC_S est obtenue par

minimisation stochastique suite à l'abstraction des actions interactives pour les accès à la variable *sense* dans la composition synchrone entre IMC_S(0) et le comportement de la variable *count* en cache et mémoire (*ligne 31*).

Algorithme 7 : $Comp_2$

```
   Data : i process identifier ; n number of process ;
 1 begin
 2     for i=0 ; i < n ; i++ do
 3         B(i) = Red (Gen (ROOT_BARRIER)) ;
 4         L(i) = StRed (Gen (ROOT_LATENCY));
 5         LS(i) = Red (Gen (ROOT_LOCAL_SENSE)) ;
 6         B_LS(i) = Red (Hide (LS_P(i), Synch (B(i), LS(i))));
 7         B_L(i) = StRed (Synch (B_LS(i), L(i)));
 8     end
 9     for i=0 ; i < n ; i++ do
10         if i == 0 then
11             Interface(i) = Red (Hide ((W_P(i),LS_P(i)), B(i)));
12         else
13             Interface(i) = Red (Hide ((W_P(i),LS_P(i)), ASynch (B(i),
               Interface(i − 1))));
14         end
15     end
16     C = Red (Gen (Proj (ROOT_COUNT, Interface(n − 1))));
17     S = Red (Gen (Proj (ROOT_SENSE, Interface(n − 1))));
18     for i=n-1 ; i ≥ 0 ; i- - do
19         if i == 0 then
20             IMC_C(i) =StRed (Hide (C_P(i), Synch (IMC_C(i+1),
               B_L(i))));
21             IMC_S(i) =StRed (Hide (S_P(i), Synch (IMC_S(i+1),
               B_L(i))));
22         else if i == n-1 then
23             IMC_C(i) =StRed (Hide (C_P(i), Proj (Synch (C, B_L(i)),
               interface(i − 1))));
24             IMC_S(i) =StRed (Hide (S_P(i), Proj (Synch (S, B_L(i)),
               interface(i − 1))));
25         else
26             IMC_C(i) =StRed (Hide (C_P(i), Proj (Synch
               (IMC_C(i + 1), B_L(i)), interface(i − 1))));
27             IMC_S(i) =StRed (Hide (S_P(i), Proj (Synch (IMC_S(i + 1),
               B_L(i)), interface(i − 1))));
28         end
29     end
30     CTMC_C =StRed (Hide (S_P.*, Synch (IMC_C(0), S)));
31     CTMC_S =StRed (Hide (C_P.*, Synch (IMC_S(0), C)));
32 end
```

Le tableau 6.7 illustre la taille de l'espace d'états des différents sous-comportements. Avec cette stratégie, 3 est le nombre maximal de processus pour obtenir une CTMC. Elle souffre en effet de l'explosion d'espace d'états des graphes intermédiaires générés durant la quatrième étape.

S/Comportement	Taille	Nombre processus (n)	
		2	3
B_L(i)	États	34	34
	Transitions	62	68
Interface(i)	États	64	512
	Transitions	576	8.064
C	États	7	13
	Transitions	12	33
S	États	11	21
	Transitions	48	156
IMC_C(2)	États	–	232
	Transitions	–	776
IMC_S(2)	États	–	498
	Transitions	–	3.188
IMC_C(1)	États	104	4.316
	Transitions	258	17.162
IMC_S(1)	États	210	11.122
	Transitions	729	58.840
IMC_C(0)	États	641	58.537
	Transitions	1994	284.451
IMC_S(0)	États	1.394	73.729
	Transitions	4.189	376.841
CTMC_C	États	39	179
	Transitions	77	503
CTMC_S	États	39	179
	Transitions	77	503

TAB. 6.7 – *Taille des graphes BCG générés dans la génération compositionnelle "$Comp_2$"*

6.2.3 Stratégie de génération compositionnelle "$Comp_3$"

Á la différence des stratégies de génération compositionnelle présentées précédemment, qui se basent sur la décomposition définie dans le schéma de synchronisation en LOTOS, la génération compositionnelle $Comp_3$ se base sur la décomposition existante au niveau de l'architecture matérielle : mémoire, cache.

La construction de la CTMC par $Comp_3$ peut être réalisée en 3 étapes principales :

Étape 1 (*lignes 2-14*) : génération du comportement fonctionnel du système ($F(n-1)$) sans le protocole de cohérence de caches, par conséquent, les actions modélisant les accès aux variables ne comportent pas les latences d'accès.

Étape 2 (*lignes 15-25*) : génération du comportement stochastique IMC des variables partagées en cache uniquement ($C\_L(n-1)$), c'est à dire les actions modélisant les accès ne comportent pas l'offre qui représente le contenu des variables . Notons que le comportement fonctionnel $F(n-1)$ joue le rôle d'une interface comportementale pour $C\_L(n-1)$.

Étape 3 (*lignes 26*) : génération du comportement stochastique CTMC par la réduction stochastique suite à l'abstraction des toutes les actions interactives dans la composition synchrone entre le comportement fonctionnel $F(n-1)$ et l'IMC $C\_L(n-1)$.

Algorithme 8 : $Comp_3$

```
   Data : i process identifier ; n number of process ;
 1 begin
 2   for i=0 ; i < n ; i++ do
 3     B(i) = Red (Gen (ROOT_BARRIER)) ;
 4     LS(i) = Red (Gen (ROOT_LOCAL_SENSE)) ;
 5     B_LS(i) = Red (Hide (LS_P(i), Synch (B(i), LS(i))));
 6   end
 7   M = Red (Gen (ROOT_MEMORY)) ;
 8   for i=0 ; i < n ; i++ do
 9     if i == 0 then
10       F(i) = Red (Synch (B_LS(i), M)));
11     else
12       F(i) = Red (Synch (B_LS(i), F(i − 1))));
13     end
14   end
15   for i=0 ; i < n ; i++ do
16     L(i) = StRed (Gen (ROOT_LATENCY));
17   end
18   C = Red (Gen (ROOT_CACHE)) ;
19   for i=0 ; i < n ; i++ do
20     if i == 0 then
21       C_L(i) = StRed (Proj (Synch (L(i), C), F(n − 1))));
22     else
23       C_L(i) = StRed (Proj (Synch (L(i), C_L(i − 1)),
         F(n − 1))));
24     end
25   end
26   CTMC =StRed (Hide (S_P.*, C_P.*), Synch (F(n-1), C_L(n-1))));
27 end
```

Le tableau 6.8 illustre la taille de l'espace d'états des différents sous-comportements. Le niveau d'efficacité de cette stratégie est le même que celui de la stratégie 6.2.1, car elle réussie à générer la CTMC pour 6 processus. Elle souffre également du même problème de l'explosion d'espace d'états des graphes intermédiaires générés durant la quatrième étape.

Le but de la génération compositionnelle était de pallier l'explosion d'espace d'états des IMC intermédiaires, qui constitue le facteur limitatif de la génération directe décrite dans la section 5.1.

Les 2 stratégies de génération compositionnelle 6.2.1 et 6.2.3 ont permis d'aller jusqu'à 6 processus pour l'algorithme barrière *centralized* contre 4 pour la génération directe, mais les résultats sont moins bons pour les autres algorithmes de barrière plus complexes (*tournament* et *combining tree*).

S/Comportement	Taille	Nombre processus (n)				
		2	3	4	5	6
B_LS(i)	États	10	10	10	10	10
	Transitions	14	16	18	20	22
M	États	6	8	10	12	14
	Transitions	56	144	192	290	408
F($n-1$)	États	18	68	262	1.032	4.106
	Transitions	36	204	1.048	5.160	24.636
C	États	18	44	100	222	490
	Transitions	144	528	1.600	4.440	11.760
C_L($n-1$)	États	282	3.922	47.984	558.899	6.371.231
	Transitions	1.006	21.134	345.233	5.026.734	68.747.854
CTMC	États	39	177	604	1.625	3.774
	Transitions	77	503	2.067	6.365	16.331

Tab. 6.8 – *Taille des graphes BCG générés dans la génération compositionnelle "Comp$_3$"*

6.3 Nouvelle méthode pour l'insertion des taux markoviens

La méthode d'insertion des taux markoviens sur laquelle s'est basée notre génération de CTMC était l'un des facteurs favorisant l'explosion d'espace d'états des modèles, notamment en appliquant la méthode 5.1. Elle nécessite en effet la génération de graphes intermédiaires 10 fois plus grands que le graphe CTMC final. Ceci est dû à l'insertion des taux markoviens à partir de la réalisation de rendez-vous sur la porte LATENCY après chaque accès ou attente.

L'idéal est de pouvoir générer le comportement stochastique à partir de la spécification LOTOS d'origine, c'est à dire sans les rendez-vous sur la porte LATENCY, mais tout en assurant la cohérence et l'exactitude du modèle obtenu (absence de non-déterminisme, concurrence stochastique des accès en conflit vérifiée).

Cette approche idéale dans l'insertion des taux markoviens dans la spécification LOTOS a été suggérée par Holger Hermanns. Elle est basée sur le même principe de l'approche 5.2.1, qui consiste à transformer les transitions issues des rendez-vous sur la porte ACTION en taux markoviens, mais pour la latence de l'accès antérieur. Plus précisément :

Soit la séquence d'accès suivante :

ACTION ! P_0 ! var ! Load ! False_Val ! M_FSB_1

ACTION ! P_0 ! var ! Store ! True_Val ! C_FSB

Le taux markovien du premier accès (M_FSB_1) est obtenu par la transformation de la transition modélisant le second accès, qui est en écriture, en une transition markovienne d'un taux $\lambda_{M\_FSB\_1}$.

Cette nouvelle méthode nécessite l'ajout de deux offres dans les rendez-vous sur la porte ACTION, il s'agit du nom de la variable ainsi que la

latence de l'accès précédent. Ci-dessous la nouvelle version de la primitive *centralized* obtenue après ces modifications :

```
process  BARRIER [ACTION](P : ID_Processor, pr_adr : Address, pr_l : Latency_Value) : exit :=
  ACTION  ! P ! local_sense(P) ! Load ? val :Memory_Value
          ? latency1 :Latency_Value ! pr_adr ! pr_l;
  ACTION  ! P ! local_sense(P) ! Store ! Inv_Value(val)
          ? latency2 :Latency_Value !!;ocal_sense(P) ! latency1;
  ACTION  ! P ! Count ! Fetch_and_decrement ? val :Memory_Value
          ? latency3 :Latency_Value ! local_sense(P) ! latency2;
  ...
endproc
```

Pour le premier cycle d'exécution de la primitive *centralized*, *pr_adr* et *pr_l* prennent la valeur *Nil*. Par la suite, ils prennent les valeurs du dernier accès dans le cycle d'exécution antérieur de la barrière.

Les transitions markoviennes de la CTMC sont obtenues en appliquant le renommage des transitions comme suit :

1. "**WAIT** ! P ! 0" → "START ; rate 100000",
2. "**ACTION** ! P ! .* ! .* ! .* ! .* ! Nil ! Nil" → "HIDE_LABEL",
3. "**ACTION** ! P ! .* ! .* ! .* ! .* ! local_sense(P) ! .*" → "HIDE_LABEL",
4. "**ACTION** ! P ! .* ! .* ! .* ! .* ! count ! M_FSB_1" → "rate $\lambda_{M\_FSB\_1}$",

   ...

où :

1. Le premier renommage correspond à la transition markovienne qui marque le début de la mission de l'algorithme.
2. Le second renommage correspond au premier cycle de l'exécution, l'action résultante est destinée à l'abstraction.
3. Le troisième renommage indique que l'accès antérieur était pour la variable privée *local_sense*, qui est finalement destinée à l'abstraction afin de favoriser la réduction stochastique.
4. Le quatrième renommage indique que l'accès antérieur était pour la variable partagée *count* et par conséquent la latence de son accès est prise en compte par la considération de la transition markovienne correspondante.

Le tableau 6.9 illustre la taille des graphes CTMC générés directement à partir de la spécification LOTOS (sans l'application de génération compositionnelle) pour la $topology_0$ et le protocole de cohérence de cache A. Finalement cette méthode a permis la génération de CTMC pour un nombre de processus allant de 2 à 5.

La différence entre la taille de IMC obtenue par l'insertion des taux markoviens par les rendez-vous sur la porte LATENCY selon l'approche 2 et, la taille des IMC par la transformation des transitions issues des rendez-vous sur la porte ACTION est en moyenne de 33,46%, un pourcentage qui n'a pas finalement réussi la génération de IMC pour un nombre de processus > 5.

Processus	Taille	Avant LATENCY		Sans LATENCY	
		États	Transitions	États	Transitions
2	IMC	523		413	826
	CTMC	39	77	39	77
3	IMC	10.497		7.434	22.302
	CTMC	179	503	179	503
4	IMC	182.237		118.085	472.340
	CTMC	604	2.067	604	2.067
5	IMC	–	–	1.710.988	8.554.940
	CTMC	–	–	1.625	6.365

TAB. 6.9 – *Insertion des taux markoviens sur la porte* ACTION

La génération compositionnelle pour cette nouvelle méthode d'insertion de taux markoviens a joué un effet inverse sur la taille des graphes intermédiaires. Or, dans le cas de la génération compositionnelle $Comp_1$ 6.2.1, les tailles des sous-comportements stochastiques (IMC(i)) sont 30% plus grands que ceux décrits dans le tableau 6.6.

En effet, le niveau de précision des interfaces contraignant les actions et le comportement n'était pas suffisant pour assurer une réduction efficace et éviter l'explosion d'espace d'états des graphes intermédiaires.

La définition d'interfaces efficaces et pertinentes, nécessite l'assistance du comportement de chaque algorithme état par état et transition par transitions, ce qui s'avère très couteux en terme de temps, pour un résultat qui n'est pas garanti à la fin, puisque les graphes stochastiques finaux (CTMC) étaient d'une taille importante même pour un nombre modeste de processus.

La raison profonde des difficultés rencontrés est le fait que, pour les algorithmes de barrière étudiés, la taille de la CTMC finale minimisée augmente rapidement avec le nombre de processus. Cela limite l'applicabilité de l'analyse numérique pour cette classe d'algorithmes. Une tentative pour aller plus loin est d'utiliser la méthode de simulation, ce qui fera l'objet du chapitre 7.

7 Évaluation des performances par simulation

The first ninety percent of the task takes ten percent of the time, and the last ten percent takes the other ninety percent.
–Ninety-ninety rule of the project schedules

No amount of experimentation can ever prove me right, a single experiment can prove me wrong.
–Albert Einstein

Sommaire

Suite au problème d'explosion combinatoire que nous avons rencontré, avec lequel il était impossible de faire une étude complète et concluante des performances, nous avons décidé de changer de méthode et d'évaluer les performances par simulation, mais tout en exploitant les modèles utilisés par la méthode d'analyse numérique. L'équipe VASY a développé à cette occasion un outil de simulation à la volée, appelé CUNCTATOR, qui a récemment été intégré dans la boîte à outils CADP.

Contrairement aux environnements de simulation standard comme Opnet[1], Omnet[2] ou NS-2[3], qui n'offrent pas la possibilité de vérifier la correction fonctionnelle des modèles, avec CUNCTATOR nous utilisons des modèles que nous avons déjà validés dans la phase de vérification. En plus de la complexité de l'implémentation des protocoles de cohérence des caches dans des langages de programmation impératifs, dans ces environnements, les modèles sont décrits par un ensemble de déclarations et de bibliothèques très peu structurées et les liens entre les différents composants sont édités généralement en code C. Ainsi les résultats engendrés sont souvent non crédibles [6, 82, 36].

Il existe cependant des outils basés sur l'utilisation de modèles formels, citons par exemple :

[1] http://www.opnet.com
[2] http://www.omnetpp.org
[3] http://www.isi.edu/nsnam/ns

MRMC [4] (*Markov Reward Model Checker*) : est un *model checker* pour les chaînes de Markov à temps discret et continu. MRMC nécessite en entrée un ensemble de fichiers décrivant le modèle stochastique du système, parmi eux un fichier contenant la matrice génératrice de la CTMC. Étant donné la non-évidence de la spécification de ces fichiers pour les systèmes de grande taille, MRMC permet l'utilisation des modèles générés automatiquement à partir de deux outils :

(a) PRISM [5] est un outil de vérification pour les modèles probabilistes. MRMC traite des modèles qui peuvent être générés à partir d'une spécification PRISM.

(b) PEPA Workbench [90] est un outil permettant l'évaluation des performances de CTMC générées à partir de spécifications décrites en langage PEPA (*Performances Evaluation Process Algebra*) [45].

Möbius [6] : est un outil de modélisation, d'évaluation des performances et de simulation de systèmes. C'est un environnement qui supporte plusieurs formalismes répertoriés en deux catégories : les formalismes atomiques comme SANs (*Stochastic Activity Networks*) [69] et le langage PEPA [10], et les formalismes composés, qui correspondent à une composition de formalismes atomiques suivant différentes techniques comme la composition de graphes. En outre, il offre la possibilité de définir de nouveaux formalismes et leur appliquer les différentes fonctionnalités qu'il possède.

MRMC est un simulateur destiné plutôt à la vérification, il permet la détection de l'état d'équilibre à la volée [51] mais il n'effectue aucune analyse dessus. Möbius est simulateur intéressant qui répond bien à nos besoins en terme de simulation mais au coût de l'intégration du langage LOTOS dans l'environnement Möbius, sinon au coût de la re-description de notre système en d'autres formalismes que l'outil possède. Dans les deux cas le coût s'est révélé trop élevé en terme de temps de réalisation.

Nous avons consacré la première section de ce chapitre à la présentation du principe d'évaluation des performances à la volée. Nous décrivons pas la suite l'outil de simulation CUNCTATOR et la technique d'analyse des résultats des simulations que nous avons appliquées. Nous achevons ce chapitre par les présentation des indices de performances évalués par CUNCTATOR des algorithmes de barrière *centralized*, *tournament* et *combining tree*.

[4] http://www.mrmc-tool.org
[5] http://www.prismmodelchecker.org
[6] http://www.mobius.illinois.edu

7.1 Évaluation des performances à la volée

L'obtention de l'indice de performance par l'utilisation de BCG_STEADY nécessite la construction de la totalité de la chaîne de Markov représentant le comportement stochastique du système.

Le calcul effectué par BCG_STEADY repose sur la résolution d'un système de $|S|$ équations, où $|S|$ est la taille de l'espace d'états de la chaîne de Markov [89]. Cette contrainte rend son utilisation très limitée pour le besoin d'une analyse complète et détaillée des performances de nos systèmes, en de l'explosion combinatoire de l'espace d'états (voir chapitre 6).

CUNCTATOR réalise un calcul équivalent à celui de BCG_STEADY sans la nécessité d'une vue complète des états de la CTMC. En effet, il parcourt la chaîne de Markov à la volée et réalise des calculs au niveau de chaque état visité.

Principe d'évaluation d'indice de performance à la volée

Soit S l'espace d'état de la CTMC représentant le comportement stochastique de l'algorithme MPI et, soit DM_{sim} le débit moyen de sa mission évalué par CUNCTATOR.

Rappelons que le débit moyen DM est évalué selon la formule

$$DM = \sum_{i \in S} \pi_i^* \lambda_{start}^i \tag{7.1}$$

où, λ_{start}^i correspond à la somme des taux markoviens des transitions de type *start* sortantes de l'état i. Rappelons également qu'une transition de type *start* décrit le début de la mission de l'algorithme (voir la section 5.2).

Supposons que S_v est l'ensemble des états visités au cours de la simulation. Initialement cet ensemble ne comporte que l'état initial de la chaîne de Markov. Ci-dessous, la procédure de construction de l'ensemble S_v.

Begin
$S_v = \{s_0\}$
$s_c = s_0$ **while** (!stop)
{
`/* Ensemble des états successeurs de` s_c `*/`
S_c = `successor` (s_c)
`/* Sélection aléatoire d'un état de l'ensemble` S_c `*/`
s_c = `random_select` (S_c)
`/* Ajouter le nouvel état courant` s_c `à l'ensemble` S_v `*/`
S_v = `add_state` (S_v, s_c)
}
END

Le cardinal de l'ensemble S_v dépend de la *condition d'arrêt* de la simulation que nous définissons dans la section suivante. On suppose que le booléen *stop* représente la condition d'arrêt de la simulation. L'état s_c

représente l'état courant, initialement il correspond à l'état initial s_0. L'ensemble S_c comporte tous les états successeurs de l'état courant s_c (c'est à dire les états directement accessibles à partir de s_c). Le prochain état à visiter est sélectionné aléatoirement parmi les états de l'ensemble S_c.

Remarque 7.1 *CUNCTATOR utilise le générateur de nombres aléatoires congruent de Park & Miller [87], dit le* standard minimal *:*

$$x_n = 7^5 x_{n-1} \bmod(2^{31} - 1) \tag{7.2}$$

*Ce générateur est considéré comme un bon générateur standard convenablement testé et dont le code est portable sur toutes les machines. Il satisfait en particulier les 3 exigences requises pour un bon générateur de qualité [**?**] suivantes :*

1. *Une période maximale de $2^{31} - 2$, ce qui est suffisamment long pour la plupart des applications.*
2. *Satisfait de manière acceptable les tests d'indépendances statistique ;*
3. *Réalisable par un algorithme sur une machine d'arithmétique entière à 32 bits.*

À la visite de chaque état i, la quantité DM_i est évaluée, avec

$$DM_i = D_i \, \lambda_{start}^i$$

La quantité D_i représente le temps passé à l'état i. Rappelons que (voir le rappel sur les CTMC dans la section 2.2.1) :

$$0 \leq \ Prob_i = Prob\{D_i \leq u \mid X_t = i\} = 1 - e^{-\lambda_i u} \ \leq 1, \ \ u \geq 0$$

En choisissant une valeur aléatoire entre 0 et 1 pour $Prob_i$, qu'on note par $RProb_i$, nous obtenons le temps de séjour D_i à l'état i par la formule :

$$D_i = log\left(\frac{1}{1 - RProb_i}\right)\frac{1}{\lambda_i}$$

En effet D_i représente la valeur simulée de π_i^* qui correspond à la proportion de temps passé à l'état i. Suite à l'évaluation du temps de séjour de tous les états de l'ensemble S_v, on déduit *le temps de séjour total* D_T :

$$D_T = \sum_{i \in S_v} D_i$$

Par conséquent, D_T représente la valeur simulée de $\sum_{i \in S} \ \pi_i^*$.

L'équation 7.1 est une écriture simplifiée de l'équation 7.3, puisque $\sum_{i \in S} \ \pi_i^* = 1$.

$$DM = \frac{\sum_{i \in S} \ \pi_i^* \lambda_{start}^i}{\sum_{i \in S} \ \pi_i^*} \tag{7.3}$$

Ainsi, en remplaçant π_i^* par sa valeur simulée D_i et $\sum_{i \in S} \ \pi_i^*$ par D_T dans l'équation 7.3, nous obtenons la valeur simulée DM_{sim} de DM :

$$DM_{sim} = \frac{\sum_{i \in S_v} \ DM_i}{D_T} \tag{7.4}$$

7.2 Outil de simulation : CUNCTATOR

CUNCTATOR[7] est un outil de simulation de chaînes de Markov continues à la volée. Il prend en entrée une CTMC générée à partir d'une spécification LOTOS (*.lotos), d'une expression de composition (*.exp), d'un graphe BCG (*.bcg) ou d'une séquence (*.seq). En sortie, il produit des mesures de débit (*throughput*) des étiquettes des transitions du graphe.

En fonction du format de la CTMC que CUNCTATOR prend en entrée, il fait appel à des outils différents de CADP, comme illustré dans le tableau 7.1.

Format	Outil
Spécification LOTOS (*spec*`.lotos`)	`caesar.open`
Graphe BCG (*spec*`.bcg`)	`bcg.open`
Expression de composition (*spec*`.exp`)	`exp.open`
Séquence (*spec*`.seq`)	`seq.open`

Tab. 7.1 – *Outils CADP utilisés par CUNCTATOR*

Comme CUNCTATOR était conçu spécialement pour pallier le problème de l'explosion combinatoire de l'espace d'états, il se base sur l'utilisation de différents outils de CADP (le choix de l'outil est en fonction du format de la chaîne en entrée, voir le tableau 7.1) afin qu'il ne puisse voir de la chaîne de Markov qu'un seul état s_c et les états successeurs de s_c, à la fois.

En raison du problème d'explosion combinatoire de l'espace d'états des modèles même avec en utilisant la génération compositionnelle (voir chapitre 6), nous avons appliqué CUNCTATOR sur des spécifications LOTOS.

caesar.open [*caesar_opts*] *spec*.lotos [*cc_opts*] **cunctator** [*cunctator_opts*]

caesar_opts correspondent aux options de l'outil CAESAR.OPEN[8] et *cc_opts* aux options du compilateur C, quant à *cunctator_opts*, elles représentent les options de CUNCTATOR dont nous évoquons les principales ci-dessous :

-action *label_list*
Spécifie la liste des transitions ordinaires à évaluer leur débit. Par exemple "**-action** START" signifie que l'outil va évaluer le débit des transitions étiquetées par START.

-time *total_sejourn_time*
Représente la somme des temps de séjour de tous les états visités durant la simulation (elle correspond à la valeur de D_T dans la formule 7.4). Durant la simulation, quand cette $D_T = total\_sejourn\_time$, la simulation s'arrête. La valeur par défaut de *total_sejourn_time* est 0 signifiant un temps de séjour total infini.

[7] http://www.inrialpes.fr/vasy/cadp/man/cunctator.html
[8] http://www.inrialpes.fr/vasy/cadp/man/caesar.open.html

-depth *depth*
Spécifie le nombre de transitions à parcourir durant la simulation. Quand le nombre des transitions parcourues atteint la valeur *depth*, la simulation s'arrête. *depth* prend la valeur 0 par défaut, elle signifie un nombre transitions infini.

-seed *seed*
Spécifie la graine du générateur de nombres aléatoires, utilisée pour la calcul des probabilités $RProb_i$ et la sélection de l'état successeur. Le *seed* prend des valeurs strictement supérieures à 0. 1 est la valeur par défaut.

-save *context_file*
Sauvegarde le contexte de la fin de la simulation dans un fichier binaire *context_file*, qui peut être utilisé dans la prochaine invocation de CUNCTATOR avec l'option **-restore**. Le fichier *context_file* comporte : le dernier état courant s_c de la simulation avant son l'arrêt, la somme des temps de séjour D_T et la quantité $\sum_{i \in S_v} DM_i$.

-restore *context_file*
Restaure le contexte sauvegardé dans le fichier *context_file* par une précédente invocation de CUNCTATOR avec l'option **-save** *context_file* et commence la nouvelle simulation à partir de ce contexte.

CUNCTATOR possède 2 options pour arrêter la simulation : **-time** et **-depth**. L'utilisateur peut également arrêter la simulation avec la commande **Ctrl-C**. Dans notre application de CUNCTATOR, nous avons utilisé l'option **-time** comme condition d'arrêt de la simulation. Notons que l'*arrêt de la simulation* n'implique pas la *convergence du résultat*.

La ligne de commande suivante décrit l'évaluation de l'indice de performance par CUNCTATOR de l'algorithme barrière *centralized* entre 4 processus dans la $topology_0$:

```
caesar.open -cc "-DNB_PROC=4 -DTOPOLOGY=0" centralized.lotos
            -cc "-DNB_PROC=4 -DTOPOLOGY=0" cunctator
            -action START -rename f.rename -hide f.hide
            -time 1000 -seed 31
```

La simulation s'arrête quant la somme des temps de séjour D_T atteint la valeur 1000 (**-time** 1000). La graine de la simulation est initialisée à 31 (**-seed** 31).

Puisque la chaîne de Markov est générée à partir d'une spécification LOTOS, la transformation des transitions stochastiques en taux markoviens est effectuée par l'option **-rename** en utilisant les règles de renommage définies dans le fichier *f.ren*. Avec l'option **-hide** on effectue l'abstraction de toutes les transitions interactives indiquées dans le fichier *f.hide*.

A la fin de la simulation, CUNCTATOR affiche le résultat comme suit :

```
Simulation statistics:
      transitions explored: 3023697
      invisible transitions explored: 1511850 (50.01%)
      time elapsed: 1000.001653
      nondeterminism detected
      throughput for "START": 7.655982143
```

avec :

- *transitions explored* : nombre de transitions stochastiques explorées.
- *invisible transitions explored* : nombre de transitions internes i explorées.
- *time elapsed* : la somme des temps de séjour dans les états visités (D_T).
- *throughput for* "START" : le débit moyen de la mission DM_{sim} de l'algorithme.

Remarque 7.2 *Comme il est impossible d'effectuer la réduction stochastique à la volée, CUNCTATOR traite les transitions internes i suivant le principe du progrès maximal [38, 97, 74], c'est à dire qu'il donne la priorité aux transition internes par rapport aux transitions invisibles issues du même état.*

Afin de faciliter la référence à la simulation par CUNCTATOR et ces résultats, nous adoptons le vocabulaire suivant :

- **Expérience** : notée par X, elle représente une simulation par CUNCTATOR. Par exemple, la ligne de commande précédente est une expérience.
- **Paramètre** : la spécification LOTOS de l'algorithme MPI, le nombre de processus et la topologie du système.
- **Objectif** : la liste des actions dont on veut évaluer le débit, ainsi que les fichiers de renommage et d'abstraction appropriés. Précisément, il s'agit des valeurs données aux options **-action**, **-rename** et **-hide**.
- **Facteur** : correspond à l'option qui affecte le résultat d'une l'expérience. On distingue deux types de facteurs :

  facteur principal désignant l'option **-time**,

  facteur secondaire désignant l'option **-seed**.
- **Observation** : notée par x, elle correspond à l'indice de performance résultant de l'expérience X. Par exemple, dans le résultat de la ligne de commande précédente x est égal à $7,655982143$.
- **Échantillon** : noté par E, il décrit un ensemble d'observations $x_1, ..., x_n$ issues des expériences $X_1, \ldots, X_n$ respectivement. L'unique différence entre les expériences d'un même échantillon est la condition d'arrêt : elles utilisent un facteur principal différent les uns des autres. En revanche, elles partagent le *même paramètre*, les *mêmes objectifs* et le *même facteur secondaire*.

7.3 Méthodes d'analyse de données

Notre simulation appartient à la famille des simulations sans condition d'arrêt (*non-terminating simulation*), destinées à l'étude des systèmes en régime stationnaire. Ces simulations ne possèdent pas de critères d'arrêt percevables, puisque dans la théorie on s'intéresse au comportement du système quand $t \rightarrow \infty$. Cela constitue une de leurs difficultés majeures, puisqu'une simulation trop courte engendre des résultats qui peuvent être très variables (le système est encore en régime transitoire) et, d'autre part, une simulation trop longue s'avère très coûteuse en termes de temps et de ressources.

Plusieurs méthodes ont été proposées pour la détection de la fin du régime transitoire [49], elles sont basées sur le principe que les résultats de cette phase ne doivent pas faire part du calcul final, puisque nous nous intéressons aux performances du système en régime stationnaire. Ces méthodes sont généralement heuristiques, dès lors qu'il est difficile de caractériser la phase transitoire et impossible d'identifier exactement sa fin, ainsi que la forte dépendance des caractéristiques du système. En outre, de nombreuses études sur l'analyse statistique des données issues des simulations en phase d'équilibre, ont donné naissance à des méthodes basées sur la théorie de l'estimation, dans le but d'estimer des paramètres relatifs au système simulé [49, 26].

Nous nous sommes concentrés dans notre étude sur les méthodes d'analyse des résultats en régime stationnaire, ce choix étant dicté par la nature du système que nous simulons :

- La simulation est effectuée sur des chaînes de Markov de nature stationnaire, car elles décrivent le comportement stochastique de boucles d'accès aux données (voir la section 5.2). La phase transitoire initiale est justifiée par l'état initial des caches (les données sont invalides dans les caches). Dans un échantillon d'observations, on détecte facilement les données liées à la phase transitoire.
- Les observations d'un échantillon ne dépendent pas les unes des autres, car elles sont issues d'expériences indépendantes.

Nous décrivons dans ce qui suit le principe de l'estimation par l'intervalle de confiance, ainsi que la méthode de son application dans l'analyse des données issues de simulations.

7.3.1 Intervalle de confiance

La nature infinie de notre simulation nous mène auprès d'une analyse d'un échantillon d'observations de la population infinie. Dès lors les résultats conduisent plutôt à *une estimation* qu'à *la vraie* valeur de l'indice de performance.

La valeur estimée est en quelque sorte une *estimation ponctuelle* qui s'avère à elle seule une donnée insuffisante. L'idée est alors de construire autour de la valeur estimée un intervalle de *confiance* ayant une probabilité prédéterminée pour recouvrir la vraie valeur de l'indice de performance. En d'autres termes, cet intervalle permet de cadrer avec une certaine confiance, un ensemble de valeurs susceptibles de contenir la vrai

valeur de l'indice de performance. La largeur de l'intervalle de confiance reflète la précision avec laquelle à été estimé l'indice de performance [49, 26, 61, 15, 94].

Nous avons choisi la moyenne comme estimateur en raison de sa compatibilité avec les observations. On dit qu'on évalue un intervalle de confiance d'une moyenne.

L'indice de performance μ est estimé par la définition, autour de la moyenne $\overline{X}_n$ des observations $x_1, ..., x_n$ des expériences $X_1, ..., X_n$ respectivement, d'un intervalle de confiance $[B_1, B_2]$ qui contienne μ avec une probabilité $1 - \alpha$:

$$Prob(B_1 \leq \mu \leq B_2) \ = \ 1 - \alpha$$

On dit que l'intervalle de confiance $[B_1, B_2]$ est d'un *niveau de confiance* $100(1 - \alpha)$. Le niveau de confiance est exprimé généralement en pourcentage proche de 100%. La quantité $1 - \alpha$ représente le *coefficient de confiance* et α est le *niveau de signification*, il est exprimé comme une fraction très proche de zéro, on l'appelle également le *coefficient de risque* puisque $Prob(\mu \notin [B_1, B_2]) = \alpha$.

Théorème 7.1 *Soit $(X_n)_{n \in N^*}$ une suite de variables aléatoires indépendantes de même loi, d'espérance μ et de variance σ^2. Posons :*

$$\forall n \in N^* \quad \overline{X}_n = \frac{X_1 + ... + X_n}{n} \quad et \quad Z_n = \frac{\sqrt{n}}{\sigma}(\overline{X}_n - \mu).$$

La suite des sommes partielles normalisées $(Z_n)_{n \in N^*}$ converge en distribution vers la loi normale $N(0, 1)$, c'est-à-dire :

$$\forall a, b \quad -\infty \leq a < b \leq +\infty \quad \lim_{n \to \infty} Prob[a < Z_n < b] = \int_a^b \frac{1}{\sqrt{2\pi}} e^{-x^2/2} dx$$

Selon le théorème central limite 7.1, la moyenne $\overline{X}_n$ est une variable aléatoire de la loi normale même si les observations $x_1, ..., x_n$ ne le sont pas, cela du fait que la somme de variables aléatoires indépendantes et de loi identique $(x_1, ..., x_n)$ tend pour $n \to \infty$ vers la loi de répartition normale d'espérance μ et de variance σ^2 :

$$\overline{X} \ \rightsquigarrow \ N(\mu, \sigma/\sqrt{n})$$

Utilisant ce théorème, l'intervalle de confiance d'une moyenne est donné par :

$$\left[\overline{X}_n - \frac{z_{1-\alpha/2} S_n}{\sqrt{n}}, \ \overline{X}_n + \frac{z_{1-\alpha/2} S_n}{\sqrt{n}} \right] \tag{7.5}$$

Étant donné que la valeur de μ est inconnue, la valeur de la variance σ^2 est inconnue par conséquent. Ce qui fait qu'on l'estime par la variance empirique S_n^2. On a :

$$\overline{X}_n = \frac{1}{n} \sum_{i=1}^n x_i \quad \text{et} \quad S_n^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \overline{X}_n)^2.$$

La valeur de $z_{1-\alpha/2}$ est le point percentile de loi *student* si $n \leq 30$ et de la loi *normal* sinon [61].

La longueur de l'intervalle $\Delta = \frac{z_{1-\alpha/2}S_n}{\sqrt{n}}$ diminue lorsque la taille de l'échantillon n augmente. C'est un résultat général et qui va dans le bon sens : un intervalle de confiance devient plus large quand on exige un niveau de confiance plus élevé, et plus étroit quand on dispose d'un plus grand nombre d'observations.

7.3.2 Méthode de saisie et d'analyse de données

Plusieurs méthodes de saisie et d'analyse de données ont été proposées pour surmonter les problèmes théoriques qui découlent de la nature des observations recueillies durant la phase d'équilibre des simulations. Elles se distinguent par la façon avec laquelle elles estiment la variance nécessaire pour déterminer l'intervalle de confiance [49, 26, 61]. Nous avons utilisé dans notre analyse la méthode dite *des simulations parallèles* (*Independent Replication*), couramment employée lorsque la période transitoire du système est assez courte.

La méthode des simulations parallèles consiste en la production de plusieurs échantillons d'expériences avec des graines différentes. On procède comme suit :

Étape 1. On réalise m échantillons de taille n. Seule la valeur du facteur secondaire (la graine) différencie les échantillons.

Étape 2. On calcule la moyenne de chaque échantillon E_j

$$\overline{X}_j = \frac{1}{n}\sum_{i=1}^{n} x_{i,j} \quad j = 1, 2, ..., m$$

Étape 3. On calcule la moyenne des moyennes $\overline{X}_j$ des échantillons :

$$\overline{\overline{X}} = \frac{1}{m}\sum_{j=1}^{m} \overline{X}_j$$

Étape 4. On calcule la variance des moyennes $\overline{X}_j$:

$$\overline{S}^2 = \frac{1}{m-1}\sum_{j=1}^{m} (\overline{X}_j - \overline{\overline{X}})^2$$

L'intervalle de confiance est donné par la suite :

$$[B_1, B_2] = \left[\overline{\overline{X}} \mp \frac{z_{1-\alpha/2}\overline{S}}{\sqrt{mn}}\right]$$

Nous avons choisi la méthode des simulations parallèles pour l'analyse des résultats en raison de la sensibilité des observations à la variation de la graine de la simulation. En effet, certaines valeurs de la graine engendrent des intervalles de confiance qui ne recouvrent point la vraie valeur de l'indice de performance.

Le tableau 7.2 suivant illustre les valeurs des bornes de l'intervalle de confiance pour 5 graines différentes, avec :

1. Les échantillons comportent les indices de performances de la barrière *centralized* entre 4 processus dans la $topology_0$ évalués par CUNCTATOR.

2. Les échantillons sont de même taille : 300 observations.
3. Si x_i^j est la i^{eme} observation de l'échantillon E_j issue de l'expérience X_i^j, alors i est le facteur principal de l'expérience X_i^j, c'est à dire la valeur donnée à l'option **-time**.
4. Si x_i^j (resp. x_i^k) est la i^{eme} observation de l'échantillon E_j (resp. E_k) issue de l'expérience X_i^j (resp. X_i^k), alors les expériences X_i^j et X_i^k utilisent les *mêmes paramètres*, les *mêmes objectifs* et le *même facteur principal*, mais un *facteur secondaire* différent, c'est à dire des graines différentes et ceci pour $\forall j, k \in \{0, 1, 2, 3, 4\}$ avec $j \neq k$ et $i = 1 \,..\, 300$.
5. Nous avons évalué l'intervalle de confiance pour chacun des échantillons en utilisant la formule 7.5 avec un niveau de confiance de 99%.

Échantillon	Graine	B_1	B_2
E_0	9	0,52	0,53
E_1	31	0,58	0,59
E_2	932	26,66	80
E_3	1235	0,91	0,95
E_4	86972	4,16	7,01

TAB. 7.2 – *Variation de bornes de l'intervalle de confiance en fonction de la graine*

Notons que l'indice de performance μ de la barrière *centralized* entre 4 processus dans la $topology_0$ évalué par BCG_STEADY, vaut 0,53. D'après le tableau 7.2 on constate que le niveau d'appartenance de l'indice de performances μ à l'intervalle de confiance varie d'un échantillon à un autre :

E_0 : $\mu \in [0,52,\ 0,53]$. Avec une graine égale à 9, les expériences engendrent des observations qui recouvrent la valeur de l'indice de performance évalué par BCG_STEADY.

E_1 : $\mu \notin [0,58,\ 0,59]$. Avec un graine égale à 31, les observations sont très proches de la vrai valeur de μ. C'est un bon résultat en vue la différence entre μ et B_2 qui ne dépasse pas 8%.

E_2 : $\mu \notin [26,66,\ 80]$. Avec une graine égale à 932, les expériences engendrent des observations très divergents de la valeur de μ, par conséquent, le niveau de son appartenance à l'intervalle de confiance est très mauvais.

E_3 : $\mu \notin [0,91,\ 0,95]$. Avec une graine égale à 1235, la différence entre μ et B_2 est autour des 50%.

E_4 : $\mu \notin [4,16,\ 7,01]$. Un constat similaire à celui de l'échantillon E_2, avec une graine égale à 86972, les expériences engendrent des résultats très divergents de la vraie valeur de μ.

La méthode des simulations parallèles permet d'augmenter la chance d'obtenir des résultats convergents vers la vraie valeur recherchée de l'indice de performance.

7.4 Application

Nous présentons dans cette section l'indice de performance des barrières *centralized*, *tournament* et *combining tree* dans l'architecture FAME et le protocole de cohérence de caches *A*, sous forme d'intervalle de confiance évalué en utilisant la méthode des simulations parallèles, de la manière suivante :

1. Nous avons réalisé 10 échantillons, ce qui implique 10 graines différentes.
2. Les observations d'un échantillon sont obtenues par des expériences en utilisant les options **-save** et **-restore** afin d'augmenter la performance de la simulation en temps de calcul.
3. Les échantillons sont de même taille n.
4. L'arrêt de la simulation a lieu quand on obtient un intervalle de confiance $[B_1, B_2]$ avec $(\frac{B_2 - B_1}{B_1}) * 100 < \epsilon$. La quantité ϵ est appelée *critère d'interruption*. En effet, c'est le critère d'interruption qui implique la taille des échantillons (n), car si l'intervalle de confiance ne le vérifie pas, alors on réalise encore des expériences.
   Nous avons fixé ce critère à 5 dans nos applications, ce qui signifie que B_2 ne dépasse B_1 que d'au plus 5%.
5. Le facteur principal qui implique la fin d'une expérience correspond au *numéro* de l'expérience, c'est à dire, on lance la i^{eme} expérience avec un facteur principal égal à i (**-time** i).

Pour plus d'efficacité et de facilité dans l'analyse des résultats de la simulation, nous avons automatisé l'évaluation de l'intervalle de confiance au moyen de shell-script qui invoque CUNCTATOR pour les 10 graines en parallèle, ensuite il évalue l'intervalle de confiance, si ce dernier ne vérifie pas le critère d'interruption, il invoque de nouveau CUNCTATOR pour les 10 graines en parallèle mais avec un nouveau facteur principal.

Remarque 7.3 *Dans les tableaux ci-dessous, μ correspond à l'indice de performance évalué par BCG_STEADY.*

7.4.1 Algorithme barrière *centralized*

Processus	Topologie	Échantillon	B_1	B_2	Durée (sc)	μ
4	$topology_0$	163	0,53	0,56	44,74	0,53
	$topology_1$	362	1,06	1,11	105,14	1,25
	$topology_3$	1.439	3,38	3,57	463,10	2,05
8	$topology_0$	40	0,64	0,67	37,57	–
	$topology_1$	83	1,48	1,56	85,17	–
	$topology_3$	1.878	7,01	7,33	2259,72	–
16	$topology_0$	188	0,97	1,01	1010,34	–
	$topology_1$	987	3,23	3,39	6005,30	–
	$topology_3$	4.086	11,59	12,21	28109,67	–

TAB. 7.3 – *Intervalle de confiance pour les observations obtenues par CUNCTATOR pour la barrière* centralized

La durée nécessaire à la réalisation des expériences pour l'obtention de l'intervalle de confiance augmente avec le nombre de processeurs participant à la barrière : on passe de 44,74 secondes pour 4 processus à 7,80 heures pour 16 processus.

L'intervalle de confiance obtenu pour 4 processus pour les différentes topologies ne recouvre pas exactement l'indice de performance μ évalué par BCG_STEADY, mais la différence moyenne entre la borne B_2 et l'indice de performance μ est de 5,5% pour la $topology_0$, 11,86% pour la $topology_1$ et de 54,09% pour la $topology_3$: une différence qui augmente quand les processus participant à la barrière sont de plus en plus distants les uns des autres.

La variation des bornes de l'intervalle de confiance est cohérente par rapport à la variation de la topologie : la valeur de B_1 et B_2 augmente quand la distance entre les différents processus s'amplifie. Ce constat est valable pour 4, 8 et 16 processus, comme l'indique le tableau 7.3.

7.4.2 Algorithme barrière *tournament*

Le tableau 7.4 indique les intervalles de confiance de l'indice de performance de la barrière *tournament* pour 4 et 8 processus.

Dans la barrière *tournament* pour 4 processus, la différence moyenne entre la borne B_2 et l'indice de performance μ évalué par BCG_STEADY, ne dépasse pas 22%, un résultat jugé satisfaisant.

Le coût des simulations en temps de calcul pour la barrière *tournament* est très important, il a fallu plus de 8 heures pour obtenir l'intervalle de confiance pour 8 processus : 13 fois plus long par rapport à l'algorithme barrière *centralized*.

Processus	Topologie	Échantillon	B_1	B_2	Durée (sc)	μ
4	$topology_0$	137	0,87	0,82	130,73	0,83
	$topology_1$	352	1,65	1,74	385,93	2,15
	$topology_3$	1.290	2,85	2,89	1679,26	3,57
8	$topology_0$	30	5,34	5,61	15501,12	–
	$topology_1$	66	9,62	10,10	27463,47	–
	$topology_3$	68	11,61	12,19	31136,92	–

TAB. 7.4 – *Intervalle de confiance pour les observations obtenues par CUNCTATOR pour la barrière* tournament

7.4.3 Algorithme barrière *combining tree*

Le tableau 7.5 indique l'intervalle de confiance pour l'indice de performance de la barrière *combining tree* pour la configuration $tree_3$ de l'arbre combinatoire des processus (voir la section 3.3.3).

Le coût des simulations en temps de calcul pour la barrière *combining* est trop élevé par rapport à la taille des échantillons : était de 2 jours, la durée nécessaire pour obtenir 10 échantillons de 227 observations, pour un nombre de processus égal à 8 dans la $topology_1$.

Processus	Topologie	Échantillon	B_1	B_2	Durée (sc)
4	$topology_0$	208	0,56	0,59	119,74
	$topology_1$	519	1,77	1,86	288,20
	$topology_3$	2.157	3,14	3,30	1376,91
8	$topology_0$	30	0,87	0,91	31050,32
	$topology_1$	227	1,70	1,79	210425,20

TAB. 7.5 – *Intervalle de confiance pour les observations obtenues par CUNCTATOR pour la barrière* combining tree

Les indices de performance obtenus par simulation en utilisant CUNCTATOR sont cohérents par rapport à la variation de la topologie, ils augmentent quand la distance entre les différents processus s'amplifie.

Les résultats illustrés dans les tableaux 7.3, 7.4 et 7.5 sont insuffisants pour comparer les performances des différents algorithmes barrière *centralized*, *tournament* et *combining tree*, étant donnée la sensibilité des observations à la valeur de la graine, l'augmentation du nombre d'échantillon s'avère importante. En revanche, à l'heure ou nous écrivons ces dernières lignes de ce manuscrit, l'évaluation de l'indice de performance pour la barrière *tournament* et la barrière *combining tree* par CUNCTATOR sont en cours.

Conclusion du chapitre

Dans ce chapitre, nous proposons une autre méthode pour l'évaluation des performances basée sur la simulation en raison du problème d'explosion combinatoire de l'espace d'états des modèles fonctionnels et stochastiques.

Le simulateur CUNCTATOR réalise un calcul équivalent à celui de BCG_STEADY sans la nécessité d'une vue complète des états de la CTMC. Il parcourt la chaîne de Markov à la volée et réalise des calculs au niveau de chaque état visité.

Nous avons utilisé la méthode des simulations parallèles pour l'analyse des résultats en raison de la sensibilité des observations à la variation de la graine de la simulation. En effet, certaines valeurs de la graine engendrent des intervalles de confiance qui ne recouvrent point la vraie valeur de l'indice de performance.

Les indices de performance obtenus par simulation en utilisant CUNCTATOR sont cohérents par rapport à la variation de la topologie, ils augmentent quand la distance entre les différents processus s'amplifie.

Le coût des simulations en temps de calcul pour les différents algorithmes de barrières que nous avons étudié est très important, ce qui a impliquer des des résultats insuffisants pour comparer les performances des différents algorithmes barrière, étant donnée la sensibilité des observations à la valeur de la graine, l'augmentation du nombre d'échantillon s'avère importante.

Conclusion générale

L'évaluation des performances des algorithmes MPI sur des architectures multiprocesseurs est une tâche difficile, car l'efficacité d'un programme parallèle dépend à la fois des aspects matériels et logiciels du système. La méthodologie que nous avons proposée permet d'évaluer, à partir d'un modèle formel, les performances d'un algorithme distribué sur une machine multiprocesseurs. La modélisation concerne trois aspects impactant les performances : l'algorithme distribué lui même, le protocole de cohérence de caches et la topologie du système.

Pour spécifier le comportement de l'algorithme MPI, nous avons choisi le langage LOTOS normalisé par l'ISO, créé initialement pour la description de protocoles et de systèmes distribués, mais également utilisé pour la description de matériel [9, 35]. LOTOS bénéficie d'un environnement de développement et de vérification performant, la boîte à outils CADP [25], employée pour valider de nombreuses études de cas industrielles.

Pour effectuer l'évaluation de performances, nous avons adopté l'approche proposée en [23], qui consiste à augmenter le modèle LOTOS avec des informations quantitatives (latences) permettant de le transformer en une chaîne de Markov interactive [41] et en chaîne de Markov à temps continu par la suite, que l'on peut analyser avec, entre autres, les outils offerts par CADP [43]. Cette approche permet de vérifier les propriétés fonctionnelles du système (correction de l'algorithme distribué et du protocole de cohérence de caches) et d'évaluer les performances (latences et et le nombre de défauts de cache par variables) en utilisant la même spécification LOTOS.

Nous avons appliqué cette méthodologie pour évaluer les performances de deux benchmarks MPI. Le premier est un algorithme de communication point à point, appelé *ping-pong*. Il est destiné à mesurer la latence d'un échange de message entre deux processus en utilisant les primitives *send* et *receive*. Nous avons évalué ses performances pour deux variantes de primitives sur l'architecture FAME de Bull. Les résultats obtenus de manière automatique en appliquant la chaîne d'outils aux spécifications formelles concordent avec les mesures expérimentales effectuées sur la machine réelle. La modélisation formelle nous a permis d'analyser facilement les performances de cet algorithme dans différents environnements matériels en faisant varier les topologies du système et le protocole de cohérence de caches.

Le second algorithme concerne la synchronisation entre plusieurs processus en utilisant les primitives de barrière. Nous avons modélisé, vérifié et évalué les performances de trois algorithmes de barrière différents [68] : *centralized*, *tournament* et *combining tree* dans des environnement matériels différent : architecture FAME, topologie, protocole de cohérence de caches.

Cependant, cette méthodologie atteint ces limites quand l'espace d'états des modèles explose. En effet, les algorithmes MPI que nous avons modélisés et en particulier les algorithmes de *barrière*, possèdent un facteur favorisant l'explosion combinatoire de l'espace d'états du modèle spécifiant le comportement fonctionnel ou stochastique, il s'agit du nombre de processus participant à la barrière.

Pour pallier à ce problème d'explosion combinatoire de l'espace d'états des modèles, nous avons procédé à l'analyse des performances par simulation, tout en exploitant les spécifications LOTOS utilisées pour l'évaluation des performances par analyse numérique. L'équipe VASY a développé à cette occasion un outil de simulation à la volée, appelé CUNCTATOR, qui a récemment été intégré dans la boîte à outils CADP. En effet, le simulateur CUNCTATOR réalise un calcul équivalent à celui de BCG_STEADY sans la nécessité d'une vue complète de l'espace d'états, il parcourt la chaîne de Markov à la volée et réalise des calculs au niveau de chaque état visité.

Les indices de performances obtenus par CUNCTATOR sont cohérents et très proches des indices de performances évalués par l'analyse numérique par BCG_STEADY en absence de l'explosion d'espace d'états. En revanche, si le problème d'explosion de l'espace d'état est évité dans le simulateur CUNCTATOR, le coût de la simulation en temps de calcul peut être trop élevé dans certains cas (généralement en fonction de l'algorithme et le nombre de processus participant à son exécution).

Dans la continuité directe de notre travail de thèse, nous pouvons dégager plusieurs directions de recherche.

Premièrement, il serait intéressant d'expérimenter la vérification et l'évaluation de performances massivement parallèle pour améliorer le passage à l'échelle sur des modèles de taille plus conséquente.

Un autre moyen de combattre l'explosion d'états des chaînes de Markov est d'étudier la réduction à la volée de ces chaînes modulo la bisimulation stochastique de branchement ; cela est loin d'être trivial, car les définitions de cette relation disponibles dans la littérature ne sont pas adaptées au calcul à la volée.

Enfin, la méthodologie que nous avons proposée pourrait être partiellement automatisée, en traduisant les primitives MPI, décrites dans un langage algorithmique de haut niveau, vers des spécifications formelles LOTOS (éventuellement complétées par des descriptions des mémoires, topologies et caches en langage C).

Annexes

A

A.1 Arbre binomial

Un arbre binomial est défini récursivement comme suit :

- L'arbre binomial d'ordre 0 est un simple nœud.
- L'arbre binomial d'ordre k possède une racine de degré k et ses racines d'arbre binomiaux d'ordre $k-1, k-2, \ldots, 2, 1, 0$ dans cet ordre.

Un arbre binomial d'ordre k peut être construit à partir de deux arbres d'ordre $k-1$ en faisant de l'un le fils le plus gauche de la racine de l'autre.

Un arbre binomial d'ordre k possède 2^k nœuds et a pour taille k.

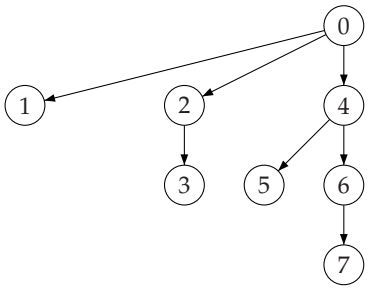


Fig. A.1 – *Exemple d'un arbre binomial*

A.2 Macros pour les prédicats de base

```
macro CC_Load (state_after_0, state_after_1, state_before_0, state_before_1,
                transfer_type, id_provider) =
  Action_Verif ('.*', 'LOAD', '.*','.*','.*', state_after_0, state_after_1, state_before_0,
                state_before_1, transfer_type, id_provider, '.*','.*','.*')
end_macro
```

```
macro Access (id_proc, transfer_type, id_provider, transfer_level, inv_level, latency)=
  Action_Verif (id_proc, '.*','.*','.*','.*','.*','.*','.*','.*','.*',transfer_type, id_provider,
                transfer_level, inv_level, latency)
end_macro
```

```
macro Take_Lock (i, adr, j) =
  Action_Verif (i, 'TEST_AND_SET', adr, j, 'UNLOCK','.*','.*','.*','.*','.*','.*','.*','.*','.*','.*')
end_macro
```

```
macro Free_Lock (i, adr, j)=
  Action_Verif (i, 'STORE', adr, j, 'UNLOCK', '.*','.*','.*','.*','.*','.*','.*','.*','.*','.*')
end_macro
```

```
macro Private_Access (id_proc, adr, id_proc_adr)=
  Action_Verif (id_proc, '.*', adr, id_proc_adr, '.*','.*','.*','.*','.*','.*','.*','.*','.*','.*')
end_macro
```

Bibliographie

[1] James K. Archibaldt. A Cache Coherence Approach for Large Multiprocessor Systems. *ACM International Conference on Supercomputing*, pages 337–345, 1988. (Cité page 12.)

[2] M. Bernardo and R. Gorrieri. A tutorial on EMPA : a theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theoret. Comput. Sci*, 202 :1–54, 1998. (Cité page 42.)

[3] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1) :25–59, January 1987. (Cité page 26.)

[4] Ed Brinksma and H.Hermanns. Process Algebra and Markov Chains. *Computer Science, Lectures on Formal Methods and Performance Analysis*, 2001. (Cité page 42.)

[5] R. Calkin, R. Hempel, H.C. Hoppe, and P. Wypior. Portable programming with the PARMACS message-passing library. *Parallel Computing*, 20 :615–632, April 1994. (Cité page 19.)

[6] D. Cavin, Y. Sasson, and A. Schiper. On the accuracy of manet simulators. In *Proc. 2nd ACM International Wrkshop on Principles of mobile computing*, pages 38–43, 2002. (Cité page 159.)

[7] L. M Censier and P. Feautirer. A new solution to coherence problems in multicaches systems. *IEEE transctions on computers*, C 27 :1112–1118, December 1978. (Cité page 12.)

[8] G. Chehaibar, M. Zidouni, and R. Mateescu. Modeling Multiprocessor Cache Protocol Impact on MPI Performance. In *Proceedings of the 2009 IEEE International Workshop on Quantitative Evaluation of Large Scale Systems and Technologies QuEST'09*. IEEE Computer Society, May 2009. (Cité page 124.)

[9] Ghassan Chehaibar, Hubert Garavel, Laurent Mounier, Nadia Tawbi, and Ferruccio Zulian. Specification and Verification of the PowerScale Bus Arbitration Protocol : An Industrial Experiment with LOTOS. In *Proc. FORTE/PSTV'96 (Kaiserslautern, Germany)*, pages 435–450, 1996. (Cité page 175.)

[10] G. Clark and W. H. Sanders. Implementing a stochastic process algebra within the Mobius modeling framework. In *Proc. of the Joint International Workshop, PAPM-PROBMIV 2001*, volume 2165, pages 200–215. Berlin : Springer, September 2001. (Cité page 160.)

[11] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 1999. (Cité page 87.)

[12] D.R. Cox. A use of complex probabilities in the theory of stochastic processes. In *Proc. Cambridge Philos. Soc*, volume 51, pages 313–319, 1955. (Cité page 42.)

[13] Jack Dongarra, Steven Huss-Lederman, Steve Otto, Marc Snir, and David Walker. *MPI : The Complete Reference*. MIT Press, 1996. (Cité pages 2 et 20.)

[14] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifcation*. Springer-Verlag, 1985. (Cité page 26.)

[15] Pierre-Jean Erard and Pontien Déguénon. *Simulation par événements discrets*. Presses Polytechniques et Universitaires Romandes (PPUR), 1ere edition edition, 1996. (Cité page 167.)

[16] Karl Feind and Kim McMahon. An Ultrahigh Performance MPI Implementation on SGI ® ccNUMA Altix ® Systems. *Computational Methods in Science and Technology*, pages 67–70, 2006. (Cité page 2.)

[17] R. Finkel, D. Hensgen, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17 :1–17, 1998. (Cité page 72.)

[18] M. J. Fischer and R. E. Ladner. Propositional Dynamic Logic of Regular Programs. *Journal of Computer and System Sciences*, 18 :194–211, 1979. (Cité page 89.)

[19] M.J. Flynn. Very-high-speed computing systems. In *Proceedings of the IEEE*, volume 54, pages 1091–1909, 1966. (Cité page 6.)

[20] Hubert Garavel. *Compilation et vérification de programmes LOTOS*. PhD thesis, Université Joseph Fourier, Grenoble, Novembre 1989. (Cité pages 26, 28, 31 et 47.)

[21] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T.Vuong, editor, *Proc. FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North Holland, December 1989. (Cité pages 26 et 47.)

[22] Hubert Garavel. Binary Coded Graphs : Definition of the BCG Format. Technical report, Laboratoire de Génie Inforamtique - Institut IMAG, Grenoble, January 1991. (Cité page 47.)

[23] Hubert Garavel and Holger Hermanns. On Combining Functional Verification and Performance Evaluation using CADP. In *Lars-Henrik Eriksson and Peter A. Lindsay, editors, Proc. FME*, volume 2391 of *LNCS*, pages 410–429. Springer-Verlag, 2002. (Cité pages 40, 44 et 175.)

[24] Hubert Garavel and Frédéric Lang. SVL : a Scripting Language for Compositional Verification. In *Proceedings of the 21st IFIPWG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001*, pages 377–392, 2001. (Cité pages 48 et 146.)

[25] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2006 : A Toolbox for the Construction and Analysis of Distributed Processes. In *Proc. CAV*, volume 4590 of *LNCS*, pages 158–163. Springer-Verlag, 2007. (Cité pages 46 et 175.)

[26] Christos G. Gassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. New York : Springer, 2nd edition, 2008. (Cité pages 166, 167 et 168.)

[27] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM : Parallel Virtual Machine : A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA, 1994. (Cité page 19.)

[28] James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. *International Symposium on Computer Architecture*, pages 124–131, 1983. (Cité page 11.)

[29] James R. Goodman and Philip J. Woest. The Wisconsin Multicube : A New Large-Scale Cache-Coherent Multiprocessor. *International Symposium on Computer Architecture*, pages 422–431, 1988. (Cité page 12.)

[30] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU Ultracomputer - Designing a MIMD, Shared-Memory Parallel Machine. *International Symposium on Computer Architecture*, pages 27–42, 1982. (Cité page 11.)

[31] N. Gotz, U. Herzog, and M. Rettelbach. Multiprocessor and distributed system design : the integration of functional specification and performance analysis using stochastic process algebras. In *Performance Evaluation of Computer and Communication Systems,Lecture Notes in Computer Science*, volume 729, pages 121–146. Springer, Berlin, 1993. (Cité page 42.)

[32] J. F. Groote and F. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In *Proceedings of the 17th colloquium on Automata, languages and programming*, pages 628–638. Springer-Verlag New York, Inc. New York, NY, USA, 1990. (Cité page 47.)

[33] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the (MPI) Message-Passaing Interface Standard. *Parallel Computing*, 22 :789–828, Sept 1996. (Cité page 20.)

[34] William Gropp and Ewing Lusk. A High-Performance MPI Implementation on a Shared-Memory Vector Supercomputer. *Parallel Computing*, 22 :1512–1526, January 1997. (Cité pages 2 et 128.)

[35] J. He and K. J. Turner. Specification and verification of synchronous hardware using lotos. (Cité page 175.)

[36] J. Heidemann, N. Bulusu, J. Elson, C. Intanagonwiwat, K. Lan, Y. Xu, W. Ye, D. Estrin, and R. Govindan. Effects of detail in wireless network simulation. In *Proc. SCS Multiconference on Distributed Simulation*, pages 3–11, 2001. (Cité page 159.)

[37] J. Hennessy, A. Gupta, and M. Heinrich. Cache-Coherent Distributed Shared Memory : Perspectives on Its Development and Future Challenges. In *Proc. IEEE*, volume 87, pages 418–429, March 1999. (Cité page 12.)

[38] M. Hennessy and T. Regan. A process algebra for timed systems. *Information and Computation*, 117 :221–239, 1995. (Cité pages 43 et 165.)

[39] T.A Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-time Systems. *Information and Computation*, 1994. (Cité page 88.)

[40] H. Hermanns and M. Siegle. Bisimulation algorithms for stochastic process algebras and their BDD-based implementation. In *Proceedings of the 5th ARTS*, pages 244–265, 1999. (Cité pages 42 et 47.)

[41] Holger Hermanns. *Interactive Markov Chains and the Quest for Quantified Quality*. LNCS. Springer-Verlag, 2002. (Cité pages 34, 36, 39, 41, 42, 44 et 175.)

[42] Holger Hermanns, Ulrich Herzog, and Joost-Pieter Katoen. Process algebra for performance evaluation. *Science of Computer Programming*, 274 :43–87, 2002. (Cité page 41.)

[43] Holger Hermanns and Christophe Joubert. A Set of Performance and Dependability Analysis Components for CADP. In *Proc. TACAS*, volume 2619 of *LNCS*, pages 425–430. Springer-Verlag, 2003. (Cité pages 48 et 175.)

[44] Holger Hermanns and Joost-Pieter Katoen. Automated compositional Markov chain generation for a plain-old telephone system. *Science of Computer Programming*, 36 :97–127, 2000. (Cité pages 41, 42 et 43.)

[45] J. Hillston. *A Compositional Approach to Performance Modelling*. PhD thesis, Cambridge University Press, Cambridge, 1996. (Cité pages 42 et 160.)

[46] C.A.R. Hoar. Communicating Sequential Processes. *Communications of the ACM*, 21(8) :666–677, August 1978. (Cité page 26.)

[47] Torsten Hoefler, Christian Siebert, and Wolfgang Rehm. A practically constant-time MPI Broadcast Algorithm for large-scale InfiniBand Clusters with Multicast. In *Proc. IPDPS*, pages 1–8. IEEE. (Cité page 2.)

[48] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Organization for Standardization 8807, ISO, 1989. (Cité page 26.)

[49] Raj Jain. *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley and Sons, INC, 1991. (Cité pages 166, 167 et 168.)

[50] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *Proceedings of the seconde annual ACM symposium on Principales of distributed computing*, pages 228–240, 1983. (Cité page 47.)

[51] Joost-Pieter Katoen and Ivan S. Zapreev. Safe On-The-Fly Steady-State Detection for Time-Bounded Reachability. In *Quantitative Evaluation of Systems (QEST)*, pages 301–310. IEEE Computer Society, 2006. (Cité page 160.)

[52] R. Katz, S. Eggers, D. A. Wood, C. Perkins, and R. G. Sheldon. Implementing a Cache Consistency Protocol. *International Symposium on Computer Architecture*, pages 276–283, 1985. (Cité page 11.)

[53] J. G. Kemeny and J. L. Snell. *Finite Markov Chains*. Springer, 1976. (Cité page 47.)

[54] Wei keng Liao, Kenin Coloma, Alok Choudhary, Lee Ward, Eric Russell, and Sonja Tidemann. Collective caching : Application-aware client-side file caching. In *Proc. HPDC*, 2005. (Cité page 2.)

[55] Adam Kolawa and Jon Flower. Express is not just a message passing system. *Parallel Computing*, 20 :597–614, 1994. (Cité page 19.)

[56] D. Kozen. Results on the Propositional μ-calculus. *Theoretical Computer Science*, 27 :333–354, 1983. (Cité page 89.)

[57] Jean-Pierre Krimm and Laurent Mounier. Compositional State Space Generation from LOTOS Programs. In *Proceedings of the TACAS'97 Tools and Algorithms and Analysis of Systems*, 1997. (Cité page 146.)

[58] L. Lamport. What Good is Temporal Logic ? In *Proc. IFIP Congress*, pages 657–668, 1983. (Cité page 96.)

[59] Frédéric Lang. Compositional Verification using SVL Scripts. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2002 (Grenoble, France)*, volume 2280 of *LNCS*, pages 465–469, 2002. (Cité pages 48 et 146.)

[60] Frédéric Lang. EXP.OPEN 2.0 : A Flexible Tool Integrating Partial Order, Compositional, and On-the-fly Verification Methods. In *Proc. IFM'2005 Eindhoven, The Netherlands*. Springer-Verlag, 2005. (Cité page 146.)

[61] Averill CM. Law and W. David Kelton. *Simulation modeling and analysis*. Mc-GRAW HILL International Editions, 1991. (Cité pages 167 et 168.)

[62] Rusty Lusk and Ralph Butler. Portable parallel programming with P4. In *Proceedings of the Workshop on Cluster Computing. Supercomputing Computations Research Institute, Florida State University*, 1992. (Cité page 19.)

[63] Z. Manna and A. Pnueli. A Hierarchy of Temporal Properties. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 377–408. ACM New York, NY, USA, 1990. (Cité page 88.)

[64] M. Ajmone Marsan, L. Ciminiera, A. Bianco, R. Sisto, and A. Valenzano. A LOTOS extension for the performance analysis of distributed systems. In *IEEE/ACM Trans. Networking*, volume 2, pages 151–164, 1994. (Cité page 44.)

[65] Radu Mateescu. A Generic On-the-Fly Solver for Alternation-Free Boolean Equation Systems. In *Proc. of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2003*, April 2003. (Cité page 88.)

[66] Radu Mateescu and Mihaela Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Computer Programming*, 45 :255–281, March 2003. (Cité pages 47, 88 et 89.)

[67] Radu Mateescu and Damien Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *Proc. of the 15th International Symposium on Formal Methods FM'08*, 2008. (Cité page 101.)

[68] John M. Mellor-Crummey and Michael L.Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9 :21–65, February 1991. (Cité pages 74, 80 et 175.)

[69] J. F. Meyer, A. Movaghar, , and W. H. Sanders. Stochastic activity networks : Structure, behavior, and application. In *Proc. International Workshop on Timed Petri Nets*, pages 106–115, July 1985. (Cité page 160.)

[70] R. Milner. *Communication and Concurrency*. Prentice Hall, London, 1989. (Cité page 44.)

[71] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag, New York, 1980. (Cité page 26.)

[72] F. Moller and C. Tofts. A temporal calculus of communicating systems. In *Concur'90 : Theories of Concurrency - Unification and Extension, Lecture Notes in Computer Science*, volume 458, pages 401–415. Springer, Berlin, 1990. (Cité page 42.)

[73] M.F. Neuts. *Matrix-geometric Solutions in Stochastic Models - An Algorithmic Approach*. The Johns Hopkins University Press, Baltimore, MD, 1981. (Cité page 42.)

[74] X. Nicollin and J. Sifakis. An Overview and Synthesis on Timed Process Algebras. In *Lecture Notes in Computer Science, Proceeding of the 3rd International Workshop on Computer Aided Verfication*, volume 575, pages 376–398. Springer-Verlag London UK, 1991. (Cité pages 43 et 165.)

[75] M. Papamarcos and J. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proc. 11th Annual Int'l Symposium on Computer Architecture*, pages 348–354, June 1984. (Cité pages 11 et 12.)

[76] David A. Patterson and John L. Hennessy. *Computer Organization and Design*. Elsevier, 2005. (Cité pages 6 et 7.)

[77] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype RP3 : Introduction and Architecture. *International Conference on Parallel Processing*, pages 764–771, 1985. (Cité page 11.)

[78] P. Pierce. The NX/2 operating system. In *Proc. 3rd Conf. on Hypercube Concurrent Computers and Applications, ACM Press*, pages 384–390, 1988. (Cité page 19.)

[79] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Washington, DC, USA, 1977. (Cité page 88.)

[80] Ralf Reussner, Peter Sanders, Lutz Prechelt, and Matthias Muller. SKaMPI : A Detailed, Accurate MPI Benchmark. In *Proc. 5th European PVM/MPI Users Group Meeting*, volume 1497 of *LNCS*. Springer-Verlag, 1998. (Cité page 2.)

[81] N. Rico and G. von Bochmann. Performance description and analysis for distributed systems using a variant of LOTOS. In *B. Jonsson, J. Parrow, B. Pehrson (Eds.), Protocol Spec. Testing, and Ver. IX, North-Holland*, volume 2, pages 199–213, 1991. (Cité page 44.)

[82] G. Riley and M. Ammar. Simulating large networks : How big is big enough ? In *Proc. First International Conference on Grand Challenges for Modeling and Simulation*, 2002. (Cité page 159.)

[83] Larry Rudolph, Vasanth Bala, Shlomo Kipnis, and Marc Snir. Designing Efficient, Scalable, and Portable Collective Communication Libraries. In *Proceeding of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, volume 4590, pages 862–872, 1993. (Cité page 19.)

[84] S. Schneider. An operational semantics for timed CSP. *Inform. and Comput*, 116 :193–213, 1995. (Cité page 42.)

[85] J. Schot. *The role of Architectural Semantics in the formal approach towards Distributed Systems Design*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992. (Cité page 44.)

[86] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, MIT, Dept. of Electrical Engineering and Computer Science, 1995. (Cité page 113.)

[87] Park S.K. and Miller K.W. Random Number Generators : good ones are hard to find. *Communications of the ACM*, 31(10) :1192–1201, October 1988. (Cité page 162.)

[88] Alan Jay Smith. Cache Memories. *ACM Computing Surveys*, 14 :473–530, September 1982. (Cité page 10.)

[89] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994. (Cité pages 48 et 161.)

[90] Mirco Tribastone and Stephen Gilmore. A New Generation PEPA Workbench. *Process Algebra and Stochastically Timed Activities (PASTA)*, pages 1820–1845, 2006. (Cité page 160.)

[91] A. Valderrutten, O. Hjiej, A. Benzekri, and D. Gazal. Deriving queuing networks performance models from annotated LOTOS specifications. In *R. Pooley, J. Hillston (Eds.), Computer Performance Evaluation - Modelling Techniques and Tools*, pages 167–178. Edinburgh University Press, Edinburgh, 1992. (Cité page 44.)

[92] Aad J. van der Steen. Overview of recent supercomputers. Technical report, UCF/ Utrecht University, The Netherlands, March 2005. (Cité page 6.)

[93] A. W. Wilson. Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors. *International Symposium on Computer Architecture*, pages 244–225, 1987. (Cité page 12.)

[94] Bernard Ycart. *Modèles et Algorithmes Markoviens*. Springer-Verlag Berlin Heidelberg New York, 2002. (Cité page 167.)

[95] P. C. Yew, N. F. Tzeng, and D. H. Lawrie. Distributing Hot-Spot Addressing in Large-Scale Multiprocessors. *IEEE Transactions on Computers*, 36 :388–395, April 1997. (Cité page 79.)

[96] W. Yi. Real-time behaviour of asynchronous agents. In *Concur'90 : Theories of Concurrency - Unification and Extension, Lecture Notes in Computer Science*, volume 458, pages 501–520. Springer, Berlin, 1990. (Cité page 42.)

[97] W. Yi. CCS + Time = An Interleaving Model for Real Time System. In *18th International Colloquium on Automata, Language and Programming (ICALP)*, volume 510, pages 217–228. Springer-Verlag New York, Inc, 1991. (Cité pages 43 et 165.)

Liste des abréviations

ASCII	American Standard Code for Information Interchange
BCG	Binary Coded Graphs
CADP	Construction and Analysis of Distributed Processes
CC-DSM	Cache Coherent - Distributed Shared Memory
CC-NUMA	Cache Coherent - Non Uniform Memory Access
CCS	Calculus of Communicating Systems
CSP	Communicating Sequential Processes
CTMC	Continuous Time Markov Chains
DM	Débit Moyen
DSM	Distributed Shared Memory
DTMC	Discrete Time Markov Chains
EMPA	Extended Markovian Process Algebra
FAME	Flexible Architecture for Multiple Environments
IMC	Interactive Markov Chain (Chaîne de Markov Interactive)
ISO	International Organization for Standardization
LCycle	Latence d'un Cycle
LM	Latence Moyenne
LOTOS	Language Of Temporal Ordering Specification

MC	Markov Chains
MESCA	Multiple Environments on Scalable Architecture
MIMD	Multiple Instructions Multiple Data stream
MISD	Multiple Instructions Single Data stream
MPI	Message Passing Interface
MPMD	Multiple Process, Multiple Data
MRMC	Markov Reward Model Checker
NUMA	Non Uniform Memory Access
PA	Process Algebra (Algèbre de processus*)
PEPA	Performance Evaluation Process Algebra
RTPA	Real-Time Process Algebra
SIMD	Single Instruction Multiple Data stream
SISD	Single Instruction Single Data stream
SMP	Symmetric MultiProcessor
SPA	Stochastic Process Algebra (Algèbre de Processus Stochastiques)
STE	Système de Transitions Etiquetées
SPMD	Single Process, Multiple Data
TIPP	Timed Processes and Performance evaluation
TPA	Timed Process Algebras
UMA	Uniform Memory Access