THÈSE

présentée devant

l'Université de Rennes 1

Institut de Formation Supérieure en Informatique et Communication

pour obtenir

le titre de Docteur de l'Université de Rennes 1
Mention INFORMATIQUE

par

Carlos MAZIERO

Conception et réalisation d'un noyau de système réparti pour la simulation parallèle

Présentée le 31 mai 1994 devant la commission d'examen

M. :	J.P.	BANÂTRE	Président
Mme. :	B.	ROZOY	
MM. :	Ph.	MUSSI	
	X.	ROUSSET DE PINA	Rapporteurs
	Ph.	INGELS	
	R.	MARIE	
	M.	RAYNAL	

Résumé

L'évolution des besoins en puissance des simulations, due à la complexité croissante des modèles à simuler, a rendu indispensable la recherche de solutions permettant d'obtenir des gains en vitesse d'exécution et de traiter des modèles de grandes dimensions. Une des directions le plus étudiées est l'exécution des simulations sur des machines parallèles. Les stratégies de synchronisation utilisées dans un simulateur parallèle doivent garantir le respect de la relation de causalité, sans pour autant restreindre le traitement des événements causalement indépendants, sources potentielles de gain en vitesse d'exécution. Plusieurs techniques pour la distribution de simulateurs à événements discrets ont fait l'objet de recherches ces dernières années. Des schémas généraux peuvent être décelés, dont certains sont assez proches de techniques classiques de l'algorithmique répartie.

L'objectif de cette thèse est d'étudier certaines techniques de distribution de simulations à événements discrets et leur mise en œuvre sur des machines parallèles de type MIMD à mémoire répartie, et de préciser leur situation dans le contexte général de l'algorithmique répartie, en établissant des rapports entre ces techniques et d'autres méthodes de synchronisation connues. Nous avons construit un prototype de noyau de système réparti, appelé *Floria*, pour l'exécution de simulations à événements discrets. Le prototype rend possible l'expérimentation « in natura » des techniques de synchronisation étudiées, mais, de plus, il nous a permis de mener une réflexion approfondie sur les services nécessaires à la mise en œuvre de simulations réparties, et leurs conséquences sur la structure interne d'un noyau de système dédié à ce type d'application. La structure que nous proposons pour ce noyau est complètement modulaire, et permet d'isoler ses fonctionnalités dans leurs aspects liés à la distribution, la synchronisation et l'évolution du programme de simulation. Une évaluation quantitative de notre noyau est présentée, visant à mesurer les gains obtenus par la parallélisation et à étudier l'impact des diverses caractéristiques des modèles sur les performances du simulateur. Les conséquences des choix de mise en œuvre sur les performances sont aussi évaluées.

"À cause de l'anneau de canaux vides, aucun processus ne peut consommer le message qu'il a sur son entrée. L'arrivée d'autres messages ne changera par cette situation désespérée. Au contraire, les nouveaux messages vont venir s'agglutiner autour du processus, formant des conglomérats qui peu à peu pourrissent en dégageant une odeur nauséabonde. Les vapeurs et les fumées qui s'échappent de ces sites contaminés stagnent au dessus des canaux de communication. Aveuglés, les messages en transit entrent en collision. Les octets tués lors de ces accidents correspondent à des parties perdues du message..."

Michel HURFIN, dans un moment de profonde inspiration.

Table des matières

Table des figures

Introduction

La simulation est une méthode d'expérimentation très employée dans plusieurs domaines de l'activité scientifique et technologique ; elle permet de simplifier l'étude de la dynamique des systèmes et des phénomènes physiques complexes, à travers l'observation de l'évolution temporelle d'un modèle qui en reproduit le comportement. Étant donné que généralement des modèles mathématiques ou logiques peuvent être construits pour les systèmes étudiés, l'ordinateur s'avère alors être un outil de grande valeur pour la réalisation de simulations. De même, dans une simulation par ordinateur l'échelle de temps utilisé pour l'exécution du modèle est indépendante du temps réel, ce qui permet de faire évoluer le temps simulé avec une vitesse appropriée à l'observation du comportement dynamique du système étudié.

Dans le cadre de l'étude des systèmes continus (météo, thermodynamique, etc.), la modélisation est effectuée par un ensemble de variables d'état et d'équations différentielles qui représentent l'évolution de ces variables d'état de façon continue. La simulation d'un tel système consiste alors à calculer l'évolution temporelle des valeurs des variables d'état (à partir d'un état initial donné), en utilisant pour cela des algorithmes classiques de calcul numérique. Pour les systèmes discrets, le modèle décrit l'état initial du système et les règles pour passer d'un état à l'autre. Chaque occurrence d'une transition est considérée comme un événement, avec une date d'occurrence et qui peut entraîner des actions ayant une durée non nulle dans le temps simulé. Cette classe de systèmes est la seule considérée dans cette thèse.

La structure d'un simulateur séquentiel pour des systèmes à événements discrets est basée sur un échéancier contenant les événements qui auront lieu dans le futur du système, ordonnés par instants d'occurrence croissants. À la tête de cet échéancier se trouve le prochain événement à traiter ; son traitement peut engendrer la génération ou la suppression d'autres événements dans l'échéancier. L'horloge du simulateur, qui indique la date actuelle dans le temps simulé, prend toujours comme valeur le temps associé au prochain événement à traiter (celui situé à la tête de l'échéancier). Cette horloge avance de façon discrète, avec des pas de durée variable.

Dans une simulation, la propriété fondamentale à garantir est le respect de la relation de causalité entre les événements, par rapport au temps simulé. Cette relation, vérifiée dans tous les systèmes physiques réels, dit que le futur d'un système ne peut influencer son passé, en d'autres mots, que l'état du système à un instant quelconque t est indépendant de toute action pouvant se dérouler après t. Deux événements sont considérés comme causalement liés si (et seulement si) l'exécution de l'un peut influencer l'exécution de l'autre. Ainsi, tous les événements liés causalement doivent être traités dans l'ordre de leurs dates d'occurrence, pour que la relation de causalité soit

vérifiée. Un simulateur séquentiel assure le respect de cette relation par le mécanisme de l'échéancier, qui traite tous les changements d'état du système dans l'ordre de leur occurrence, qu'ils soient causalement liés ou non. Cela garantit que tout événement ayant lieu dans le système sera traité après les événements dont il dépend causalement.

L'évolution des besoins en puissance des simulations, due à la complexité croissante des modèles à simuler, a rendu indispensable la recherche de solutions permettant d'obtenir des gains en vitesse d'exécution et de traiter des modèles de grandes dimensions. Une des directions le plus étudiées est l'exécution des simulations sur des machines parallèles. Cependant, la parallélisation d'un simulateur à événements discrets séquentiel n'est pas une tâche facile. La difficulté provient du fait qu'il est difficile de déterminer quels événements peuvent être traités en parallèle sans violer la relation de causalité dans le temps simulé. Les stratégies de synchronisation utilisées dans un simulateur réparti doivent garantir le respect de la relation de causalité, sans pour autant restreindre le traitement des événements causalement indépendants, car ceux-ci peuvent logiquement être traités en parallèle et donc sont les sources potentielles de gain en vitesse d'exécution. Plusieurs techniques pour la distribution de simulateurs à événements discrets ont fait l'objet de recherches ces dernières années. Des schémas généraux peuvent être décelés, dont certains sont assez proches de techniques classiques de l'algorithmique répartie.

L'objectif de cette thèse est d'étudier certaines techniques de distribution de simulations à événements discrets sur des machines parallèles de type MIMD à mémoire répartie, et de préciser leur situation dans le contexte général de l'algorithmique répartie, en établissant des rapports entre ces techniques et d'autres méthodes de synchronisation connues. Nous avons construit un prototype de noyau de système réparti pour l'exécution de simulations à événements discrets. Le prototype rend possible l'expérimentation « in natura » des techniques de synchronisation étudiées, mais, de plus, il nous a permis de mener une réflexion approfondie sur les services nécessaires à la mise en œuvre de simulations réparties, et leurs conséquences sur la structure interne d'un noyau de système dédié à ce type d'application. La structure que nous proposons pour ce noyau est complètement modulaire, et permet d'isoler ses fonctionnalités dans leurs aspects liés à la distribution, la synchronisation et l'évolution du programme de simulation.

Cette thèse est divisée en 5 chapitres. Le premier chapitre donne une brève introduction aux techniques de modélisation et de simulation, notamment pour les systèmes à événements discrets. Le chapitre 2 étudie la distribution des simulations à événements discrets. On analyse tout d'abord les différentes approches suggérées pour la distribution de ce type de simulateur. Ensuite, la technique qui consiste à distribuer le modèle de simulation est analysée en détail, pour mettre en évidence les problèmes de synchronisation liés à sa réalisation. Enfin, les méthodes connues pour résoudre ces problèmes de synchronisation sont présentées.

Le chapitre 3 est consacré à la présentation d'un noyau de système réparti dédié à la simulation, pour des machines parallèles à mémoire répartie. Ce noyau, appelé *Floria*, utilise quelques unes des techniques de synchronisation décrites dans le chapitre antérieur ; il est utilisé comme prototype pour l'expérimentation de ces techniques. Ce noyau est divisé en couches qui réalisent des tâches distinctes : gestion de la distribution (acheminement des messages), gestion de la communication synchrone dans le

temps simulé et gestion de l'exécution du modèle. Cette dernière couche met en œuvre les services de communication et de synchronisation offerts au modèle. Des exemples d'utilisation de ces services sont donnés, visant à montrer leur adéquation à la modélisation de systèmes à événements discrets.

Le chapitre 4 présente une évaluation quantitative de notre noyau, visant à mesurer les gains obtenus par la parallélisation et à étudier l'impact des diverses caractéristiques des modèles sur les performances du simulateur. Les conséquences des choix de mise en œuvre sur les performances sont aussi évaluées.

Enfin, dans le chapitre 5 nous étudions les liens existants entre les techniques utilisées pour la simulation répartie et d'autres techniques de synchronisation pour les algorithmes répartis, notamment les synchroniseurs de réseaux. Dans le même esprit, nous analysons le rapport entre l'ordre établi par ces techniques et les autres relations d'ordre connues (causalité, *fifo*, etc.).

La conclusion expose les idées fondamentales issues de l'étude des techniques de synchronisation utilisées dans la distribution des simulations, ainsi qu'une analyse concise et critique des résultats qualitatifs et quantitatifs obtenus pour le noyau *Floria*. Pour finir nous répertorions les principales pistes de recherche qui peuvent être dégagées de la présente thèse et constituer le point de départ de futures recherches.

Chapitre 1

Modélisation et simulation

La simulation est une méthode d'expérimentation très employée dans plusieurs domaines de l'activité scientifique et technologique, pour étudier la dynamique de systèmes et phénomènes physiques complexes. Le principe de cette méthode est la construction d'un modèle réduit et simplifié qui reproduit le comportement du système réel à étudier, et l'observation du fonctionnement de ce modèle, qui permet alors d'obtenir des informations sur le comportement du système.

La simulation est en général une bonne alternative pour l'étude de systèmes sur lesquels l'expérimentation directe serait dangereuse (centrales nucléaires), trop coûteuse (aéronautique), ou même impossible (météorologie), ou lorsque les outils mathématiques connus ne sont pas capables de traiter correctement le problème.

L'ordinateur occupe une place très importante dans ce contexte. En général, des modèles mathématiques ou logiques peuvent être construits pour le système à étudier ; ces modèles sont ensuite traduits sous la forme d'un programme informatique. La puissance de calcul offerte par un ordinateur permet de traiter des modèles dont la complexité poserait des problèmes pour d'autres méthodes d'étude, et les expériences deviennent plus aisément reproductibles.

Dans une simulation par ordinateur, le déroulement du temps lié à l'exécution du modèle est lui aussi simulé. Ce temps (appelé *temps virtuel* ou *temps simulé*) n'est pas lié au temps réel, ce qui permet de le faire évoluer avec une vitesse appropriée à l'observation du comportement du système étudié. Nous verrons dans le chapitre suivant que l'avancement cohérent du temps simulé est le problème fondamental à résoudre pour la réalisation de simulations sur des machines parallèles. La suite de ce travail concerne uniquement la simulation par ordinateur.

Le travail d'expérimentation à travers la simulation comporte les étapes suivantes :

1. **comprendre le système réel** et définir les informations à obtenir sur son comportement dynamique ;
2. **construire un modèle** mathématique ou logique du système, prenant en compte seulement les caractéristiques pertinentes pour l'expérience effectuée, ce qui simplifie le modèle et son observation ;
3. représenter le modèle sous la forme d'un **programme informatique**, en utilisant pour cela un langage et une forme de représentation appropriés ;

4. **interpréter le programme de simulation** à l'aide d'un schéma d'exécution adapté ; des mesures sont alors prélevées sur le modèle en exécution ;
5. **analyser les résultats** des mesures, visant à formuler des conclusions pour le système réel simulé.

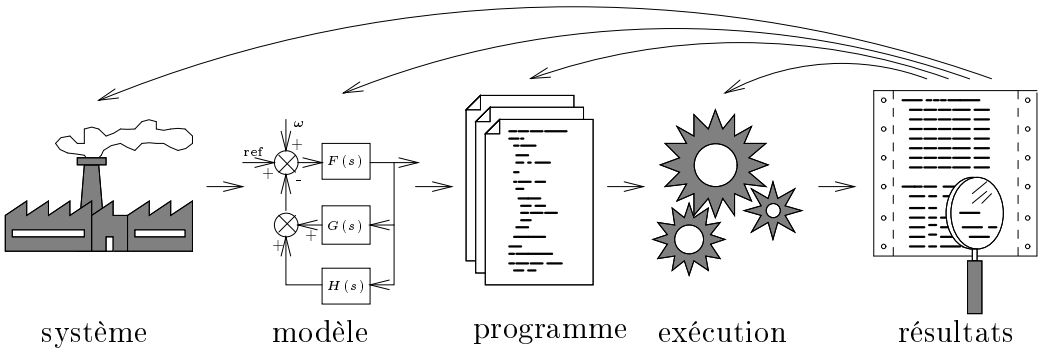

FIG. 1.1 – Étapes de la simulation

L'activité de simulation consiste alors à itérer sur ces étapes jusqu'à avoir les résultats souhaités, car les résultats d'une étape peuvent remettre en cause le travail effectué dans les étapes antérieures, comme le montre la figure 1.1. Par exemple, les mesures obtenues lors de l'exécution du programme informatique issu du modèle peuvent montrer une mauvaise formulation de celui-ci, voire même une compréhension incorrecte du système réel. Une description plus détaillée de ces étapes et de leurs interactions peut être lue dans [Pid89].

Dans cette thèse nous nous intéressons à une partie de la quatrième étape, plus précisément à la définition des schémas d'exécution adaptés à la simulation d'une certaine classe de modèles (à évolution discrète) sur un support matériel bien défini (machine parallèle MIMD à mémoire répartie). Notre regard porte aussi sur les outils nécessaires à l'écriture du programme de simulation (troisième étape), en ce qui concerne l'interface entre ce programme et les schémas d'exécution proposés.

1.1 Systèmes, modèles et programmes

Le système étudié peut être considéré comme une collection d'entités qui interagissent. Ces entités présentent des caractéristiques fonctionnelles spécifiques dans le système ; elles peuvent être des composants mécaniques ou électroniques, des êtres humains, etc. Les systèmes auxquels nous nous intéressons obéissent à deux principes [Mis86] :

- *Principe de causalité* : le futur ne peut influencer le passé. Plus précisément, l'état du système à l'instant t est indépendant de tout ce qui peut se produire à une date $t' > t$.
- *Principe de déterminisme* : le futur du système peut être déterminé à partir de son état présent et de son passé. Cela revient à dire que, à tout instant t il existe une

valeur positive ϵ telle que le comportement futur du système puisse être calculé jusqu'à $t + \epsilon$.

Ce deux principes, qui gouvernent l'évolution des systèmes réels, portent sur le temps (causalité) et la loi d'évolution du système (déterminisme). Toute modélisation devra respecter le principe de déterminisme et tout simulateur devra garantir le principe de causalité.

Dans une modélisation, il est souhaitable de préserver dans la structure du modèle le découpage du système réel en entités. La modularité rend le modèle plus facile à écrire, tester et programmer [Pid88]. Chaque entité du modèle est alors décrite à l'aide de variables d'état et de fonctions faisant évoluer ces variables. La forme de ces variables et fonctions est une conséquence du type de représentation du système que l'on souhaite effectuer :

- *Modèle continu* : dans ce type de modèle l'évolution du système est représentée de façon continue. Le comportement du système est modélisé par un système d'équations différentielles portant sur les variables d'état, qui changent de valeur de façon continue. Un exemple de cette approche est donné dans la figure 1.2, où l'on modélise le chargement d'un condensateur électrique.

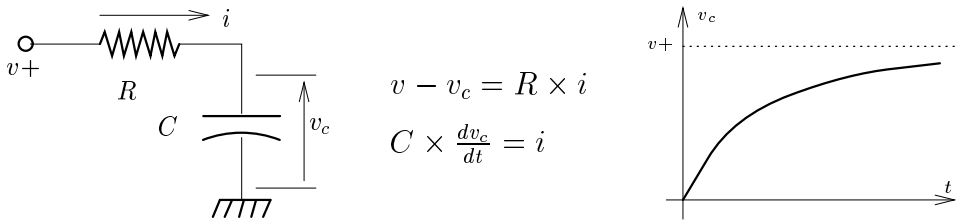

FIG. 1.2 – Modèle continu

- *Modèle discret* : ce type de modélisation est utilisé lorsque l'on souhaite décrire le comportement du système par une suite de changements d'état se produisant à des instant précis, de façon discrète (les *événements*). Ce comportement discret est représenté dans le modèle par un ensemble de transitions d'état dont les pré et post-conditions portent sur les variables d'état. L'exemple de la figure 1.3 montre une cellule d'usinage composée d'une machine-outil, d'un robot et d'un tapis roulant, et la modélisation discrète, sous la forme d'un réseau de Petri [Pet81], d'une partie de son comportement : le robot charge la machine avec une pièce qu'il reçoit sur le tapis ; une fois la pièce usinée, il la stocke dans un container.

Il faut évidemment remarquer qu'un même système peut être vu, et donc modélisé, sous une forme continue ou discrète, en fonction de l'étude que l'on veut réaliser. Ainsi, l'étude du comportement dynamique d'un train, d'un point de vue mécanique, implique la construction d'un modèle continu, dans lequel des paramètres comme le poids et la puissance des moteurs seront considérés ; les mesures portent sur le temps de freinage, l'accélération, etc. Si ce même système est étudié dans le but d'observer le flux de voyageurs, les paramètres considérés seront autres : le nombre de voyageurs, le temps d'arrêt, etc. ; les changements d'état se feront alors de manière discrète : *ouvrir_porte*, *départ*, *arrêt*, etc., ce qui implique une modélisation discrète. Dans la suite de cette thèse nous considérerons seulement les modélisations à temps discret. Bien que la simulation

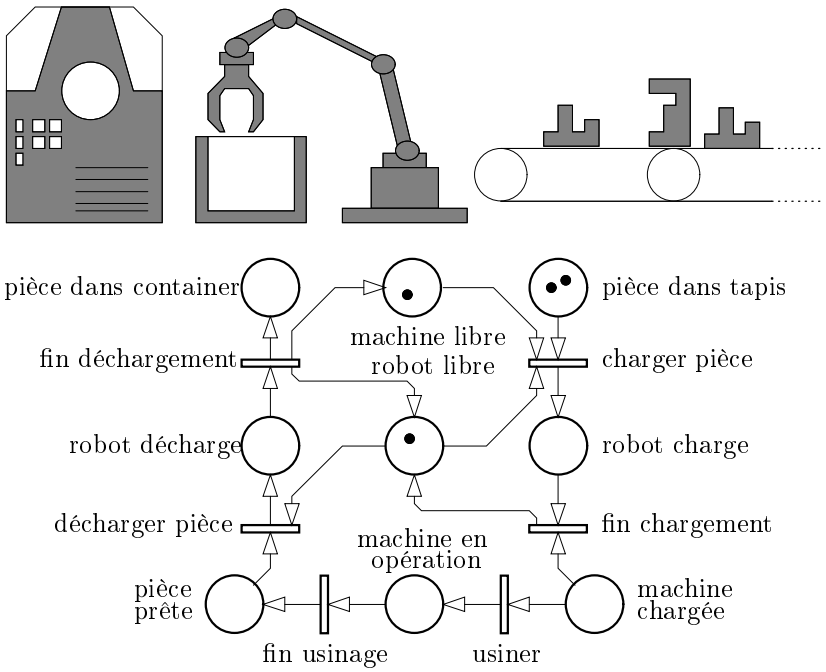

FIG. 1.3 – Modèle discret

de modèles à temps continu présente un intérêt certain, ce problème constitue un domaine de recherche à part entière, dans lequel sont utilisées des techniques d'analyse numérique (*éléments finis*, *Runge-Kutta*, etc.).

Pour l'exécution d'un modèle de simulation sur un ordinateur il est nécessaire de procéder à sa description à travers un langage informatique. Pour cela, il est possible de faire usage d'un langage de programmation universel tel que *Fortran* ou *Pascal*, ou alors d'un langage dédié à la simulation comme *Simscript* [MHC62], GPSS [Gre72] ou *Simula* [BDMN73]. Ces langages font partie d'environnements de simulation qui proposent, en plus d'un langage bien adapté à la description des modèles, des mécanismes pour la gestion du temps simulé, de files et de listes, des facilités pour pour l'analyse statistique des résultats, etc. Tout cela permet à l'utilisateur de concentrer sont attention sur la description du modèle à simuler. En ce qui concerne la description des changements d'état et des actions qui en résultent, ces langages peuvent utiliser une des approches suivantes [Ler80, Pid88, IR90] :

- *par événements* : chaque événement du système est décrit par une condition d'occurrence et par les actions à réaliser lors de son occurrence. Ces actions, qui matérialisent le changement d'état provoqué par l'événement, sont de durée nulle dans le temps simulé. Le programme est composé alors de la description des structures de données représentant l'état de chaque entité et de la description de tous les événements possibles. Cette approche est utilisé dans *Simscript* [MHC62]. La description de l'exemple donné par la figure 1.3 prend la forme suivante (t_{sim} : temps simulé) :

```
/* événement début_usinage */
si pièce_chargée alors
    démarrer_machine ;
    t_usin ← t_sim + durée_usinage ;          /* date de fin d'usinage */
fsi

/* événement fin_usinage */
si t_sim ≥ t_usin alors
    arrêter_machine ;
    envoyer pièce à robot ;
fsi
```

– *par processus* : chaque entité du modèle est représentée par un processus, qui exécute des actions matérialisant les changements d'état, l'évolution du temps simulé et les interactions avec les autres entités ; certaines de ces actions ont une durée non nulle dans le temps simulé. Les interactions entre processus peuvent s'effectuer par ressources, *i.e.* des objets passifs qui sont accédés par plusieurs processus (c'est le cas du langage *Simone* [KPH76]), par des activations et désactivations directes des processus entre eux (approche prise par *Simula* [BDMN73]), ou par échange de messages (méthode utilisé dans le langage *May* [BCM87]). Pour l'exemple de la figure 1.3, nous aurons :

```
Processus Robot :
   répéter
      attendre pièce de Tapis ;
      envoyer pièce à Machine ;
      attendre pièce de Machine ;
      envoyer pièce à Container ;
   jusqu'à fin_simulation ;

Processus Machine-Outil :
   répéter
      attendre pièce de Robot ;
      usiner pièce ;
      envoyer pièce à Robot ;
   jusqu'à fin_simulation ;
```

La description du système comme un réseau de processus avec communication par échange de messages est très proche des modèles utilisés dans l'étude de l'algorithmique répartie. Par conséquent, ce modèle s'adapte fort bien au type d'étude envisagée par cette thèse et aussi au type machine sur lequel nous avons travaillé.

1.2 Mécanismes de simulation

Une fois décrit le modèle, en termes de processus ou d'événements, il faut l'exécuter. L'exécution du modèle dépend d'une mécanique sous-jacente qui garantit le principe de causalité : un événement e daté t ne peut influencer des événements e' ayant lieu avant lui ($t > t'$). Le principe de cette mécanique est simple : à chaque pas, le simu-

lateur choisit pour l'exécution l'événement e ayant la plus petite date parmi tous les événements possibles. La causalité est ainsi garantie, car tous les événements e' dont un événement e peut dépendre causalement sont toujours traités avant lui ($t > t'$).

Tous les événements sont traités dans l'ordre de leur occurrence dans le temps simulé, au fur et à mesure de l'avancement de celui-ci. Deux façons de faire avancer le temps simulé sont alors possibles :

- *simulation dirigée par le temps* : cette technique consiste à faire progresser le temps simulé par des petits pas de durée δ. À chaque pas le simulateur vérifie l'existence d'événements ayant comme date d'occurrence la date courante t_{sim}. Il les traite et, quand plus rien ne reste à faire, il fait progresser le temp simulé. Il va de soi que la durée δ doit être judicieusement choisie : si elle est trop courte, lors de nombreuses étapes il n'y aura rien à traiter et le simulateur perd en efficacité ; par contre, avec un pas trop important il risque de « passer par dessus » certains événements, menant à une simulation incorrecte.
  Cette façon de faire avancer le temps simulé présente un intérêt lorsque la densité d'événements à traiter est régulière. Dans le cas contraire, beaucoup d'étapes seront effectués inutilement entre le traitement de deux événements, comme le montre la figure 1.4.

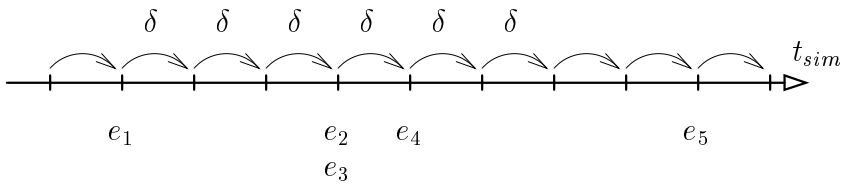

FIG. 1.4 – Simulation dirigée par le temps

- *simulation dirigée par les événements* : dans cette technique, le temps simulé avance en « sautant » d'une date d'occurrence d'événements à la date de l'événement suivant. De ce fait, après chaque progression du temps simulé il existera au moins un événement à traiter ; les étapes inutiles de la méthode antérieure sont alors éliminées. De plus, cette technique s'adapte naturellement au modèle, en faisant avancer plus ou moins rapidement le temps simulé en fonction de la densité d'événements, comme le montre la figure 1.5.

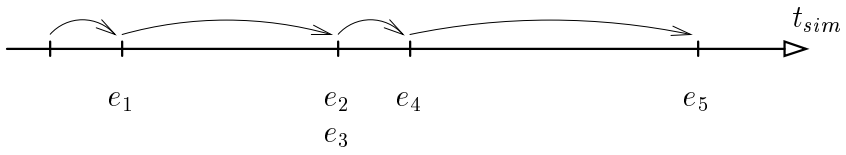

FIG. 1.5 – Simulation dirigée par les événements

Dans la technique de simulation dirigée par les événements, une structure particulière est utilisée pour la gestion des événements futurs : *l'échéancier*. Celui-ci peut être vu comme une file où sont rangés les événements futurs, dans l'ordre croissant de leurs dates d'occurrence. À la tête de cette file se trouve le prochain événement à

traiter, dont la date d'occurrence correspond à l'instant présent dans le temps simulé. Les autres événements sont considérés comme potentiels, car ils peuvent être supprimés ou modifiés pendant le traitement du premier événement. De même, le traitement d'un événement peut produire de nouveaux événements, qui sont alors insérés dans la file, dans les positions correspondantes à leurs dates d'occurrence.

L'ensemble des opérations sur l'échéancier et des structures de données concernées (dont l'échéancier lui-même) est implémenté par un *noyau de synchronisation* [Ler80], dont le mécanisme de base consiste à effectuer la boucle suivante :

répéter
  /* prendre premier événement */
  événement ← **Premier** (*échéancier*) ;
  /* mettre à jour le temps simulé */
  t_{sim} ← **date** (*événement*) ;
  traiter (*événement*) ;
jusqu'à ($t_{sim} > t_{fin\_simulation}$) ;

La mise en œuvre du noyau de synchronisation est peu influencée par le type de description employée pour le modèle. Dans un noyau adapté pour une description du modèle par événements, l'échéancier contient des événements de la forme [*exécuter* $action_i$ *à la date* t_i], ordonnées par t_i croissants. Lors du traitement de l'événement se situant à la tête de la liste, le noyau exécute le sous-programme $action_i$, qui définit le changement d'état opéré par l'événement. Par contre, l'échéancier d'un noyau de synchronisation adapté à une description du modèle par processus contiendra des événements de la forme [*réveiller* P_i *à la date* t_i]. Lorsqu'il « réveille » un processus, le noyau lui passe le contrôle et attend que le processus soit redevenu inactif ou ait finit son exécution. Cela mène à une mise en œuvre du noyau et des processus du modèle par des coroutines.

1.3 Conclusion

Dans ce chapitre nous avons introduit la simulation par ordinateur comme méthode d'expérimentation de systèmes. Cette présentation de la simulation est loin d'être exhaustive ; son seul but est de situer dans ce contexte la simulation à événements discrets et de présenter ses principes de fonctionnement. Ainsi, dans la description des étapes de simulation, la construction du modèle et le traitement des résultats de mesures n'ont guère été abordés ; une bonne description de ces étapes peut être trouvée dans [Pid88].

L'évolution des systèmes physiques est régie par deux principes : la *causalité* (le futur ne peut influencer le passé) et le *déterminisme* (le futur du système peut être déterminé à partir de son état présent et de son passé). Appliqués à la simulation, ces deux principes garantissent respectivement la sûreté de l'exécution et sa vivacité. Il est importante de remarquer le « partage de responsabilités » que nous retrouvons ici : pour que la simulation progresse, sûreté et vivacité doivent être vérifiées. La sûreté implique le respect de la causalité, ce qui doit être garanti par le support d'exécution. La vivacité repose sur le respect du déterminisme dans le modèle construit.

Lors de la modélisation, un système peut être représenté sous une forme continue ou discrète. Cette dernière, dans laquelle le comportement du système est décrit par

une suite d'événements datés, est la seule étudiée dans cette thèse. Simuler le système revient alors à interpréter le programme informatique construit à partir du modèle de ce système, en faisant avancer le temps simulé et traitant les événements au fur et à mesure de leur occurrence. Deux schémas pour réaliser cette interprétation ont été présentés : simulation dirigée par le temps ou par les événements. Ces schémas imposent un ordre total sur le traitement des événements : ils sont traités dans l'ordre strict de leurs dates d'occurrence, ce qui a pour effet un respect implicite de l'ordre causal.

Comme le respect de l'ordre causal est une condition suffisante pour garantir la correction d'un schéma de simulation, il est possible d'imaginer des schémas qui le respectent sans pour autant imposer un ordre total sur le traitement des événements. Ces schémas ne présentent pas un intérêt certain pour la simulation séquentielle, du point de vue de la vitesse de calcul, car les événements seront toujours traités de façon séquentielle, indépendamment du schéma d'exécution choisi. Toutefois cet « affaiblissement » de l'ordre de traitement des événements permet de dégager du parallélisme (à travers les événements concurrents, qui peuvent être traités simultanément) et constitue alors un point de départ pour la parallélisation des simulations.

Le chapitre suivant est consacré à l'étude de la parallélisation de simulations à événements discrets. Nous verrons que les mécanismes présentés pour la simulation séquentielle ne sont pas bien adaptés à une exécution en parallèle, ce qui rend nécessaire la proposition de nouvelles techniques pour la gestion du temps simulé.

Chapitre 2

La simulation parallèle

L'évolution des besoins en puissance des simulations, due à la complexité croissante des modèles à simuler, rend indispensable la recherche de solutions permettant d'obtenir des gains en vitesse d'exécution et de traiter des modèles de grandes dimensions. Plusieurs solutions ont été proposées pour atteindre ce but ; elles vont de l'amélioration des techniques de traitement statistique des données (la réduction de la variance des mesures se faisant alors à travers ces techniques et non plus par la répétition exhaustive de la même expérience) au câblage des parties essentielles du simulateur séquentiel classique : la gestion de l'échéancier, la génération de nombres aléatoires, etc.

La simulation d'un modèle de grandes dimensions offre en principe une quantité de parallélisme potentiel non négligeable, qui se trouve dans les événements sans lien causal et qui pourraient donc être traités em même temps. De ce fait, la parallélisation s'est avérée une voie prometteuse pour l'obtention de bonnes performances en vitesse de calcul dans les simulations.

Cependant, la parallélisation d'un simulateur à événements discrets n'est pas une tâche simple. Dans une simulation, la propriété fondamentale à assurer est le respect de la relation de causalité entre les événements, par rapport au temps simulé. Il est impérieux de garantir que tout événement ayant lieu dans le système sera traité après les événements dont il dépend causalement. Dans un simulateur séquentiel cette propriété est mise en œuvre par l'échéancier. La difficulté majeure dans la parallélisation de la gestion de l'échéancier provient du fait qu'il est difficile de déterminer quels événements peuvent être traités en parallèle sans violer la relation de causalité. Les stratégies de synchronisation utilisées dans un simulateur parallèle doivent garantir le respect de cette relation, sans pour autant restreindre le traitement des événements causalement indépendants, car ceux-ci peuvent être traités en parallèle et sont donc des sources potentielles de gain en vitesse d'exécution.

2.1 La parallélisation d'une simulation

Plusieurs techniques pour la parallélisation de simulations à événements discrets ont fait l'objet de recherches ces dernières années, dont certaines sont assez proches de techniques classiques de l'algorithmique répartie. Une étude des stratégies envisageables pour la parallélisation des simulations à événements discrets est réalisée dans [RW89].

En voici un résumé :

- *Parallélisation automatique* : un compilateur spécifique pour la machine cible détecte les parties du code pouvant s'exécuter en parallèle. À première vue c'est une approche intéressante, car la parallélisation s'effectue de façon transparente et les simulations déjà existantes peuvent être facilement parallélisées. Cependant, comme le compilateur ne prend pas en compte la structure du problème, cette approche ne permet d'exploiter qu'une faible partie de son parallélisme potentiel.
- *Distribuer les expériences* : chaque processeur de la machine exécute une simulation séquentielle indépendante du même modèle. Ainsi, plusieurs paramètres d'entrée pour le modèle peuvent être testés à la fois. Comme aucune coordination n'est nécessaire entre les processeurs, cette approche est très efficace [Hei86]. Cependant, chaque processeur doit disposer d'assez de mémoire pour stocker tout le modèle. De plus, cette approche n'est pas envisageable lorsque les paramètres d'entrée d'une simulation dépendent des résultats des autres simulations.
- *Distribuer les fonctions du simulateur* : à chaque processeur de la machine est attribuée une fonction du simulateur (gestion de l'échéancier, traitement des événements, prise de mesures, génération de nombres aléatoires, etc.) [CG88, SDC88]. Comme le nombre de fonctions d'un simulateur est limité, cette solution ne permet d'obtenir que peu d'accélération. De même, le parallélisme offert par le modèle reste inexploité, car le traitement des événements est accompli de façon séquentielle.
- *Distribuer le traitement des événements* : un échéancier global est géré, comme dans un simulateur séquentiel ; lorsqu'un processeur devient disponible, il se charge du traitement du premier événement de cet échéancier. Des mécanismes de contrôle sont nécessaires pour déterminer quels événements peuvent effectivement être traités, car l'échéancier peut être modifié par les événements en cours de traitement [Jon86]. Cette approche est particulièrement intéressante sur des systèmes à mémoire partagée, car chaque processeur peut aisément accéder à l'échéancier global.
- *Distribuer le modèle* : il est possible de représenter le modèle comme un ensemble de processus interconnectés, leur exécution étant répartie sur les processeurs disponibles. Cette approche permet d'exploiter le parallélisme potentiel du modèle, mais l'absence d'un échéancier global implique des techniques de synchronisation sophistiquées pour le maintien de la cohérence causale [CM79, JS85].

Comme ces approches ne sont pas exclusives, des combinaisons entre elles peuvent constituer des bonnes solutions pour des problèmes spécifiques. Les deux dernières approches (*i.e.* distribution des événements ou des composants du modèle) sont celles qui permettent de mieux profiter du parallélisme inhérent au modèle, et donc celles qui permettraient les meilleurs accélérations en vitesse de calcul. L'absence de structures de contrôle globales et la représentation des interactions entre processus sous la forme d'échanges de messages font de la distribution du modèle une stratégie particulièrement bien adaptée aux machines à mémoire répartie. La suite de cette étude est consacrée à cette dernière approche.

2.2 Comportement du modèle

Nous avons vu dans le chapitre 1 que le modèle de simulation peut être décrit sous la forme d'un ensemble de processus qui communiquent par échanges de messages. Les processus représentent alors les entités du modèle et les messages les interactions entre elles. Nous allons maintenant définir des règles plus précises pour le comportement du modèle, afin de pouvoir établir des stratégies de synchronisation et de les comparer.

Un processus est composé d'un *programme* ou *code*, qu'il exécute de façon séquentielle et qui représente son comportement, d'un ensemble de variables locales ou *contexte*, inaccessible aux autres processus et sur lequel il peut effectuer des calculs, et d'un ensemble de *canaux*, par l'intermédiaire desquels il interagit avec les autres processus. L'état d'un processus est donné par sa position dans son programme, les valeurs des variables de son contexte et les messages disponibles sur ses canaux, en provenance des autres processus.

L'absence de variables globales partagées par les processus dans cette proposition de comportement pour le modèle vise uniquement à simplifier le problème et rendre plus facile l'étude des algorithmes de synchronisation. La mise en œuvre de variables partagées dans un simulateur séquentiel classique ne nécessite pas de contrôle supplémentaire, car toutes les opérations de lecture et d'écriture s'effectuent dans l'ordre de leurs dates d'occurrence. La difficulté de l'implémentation de variables partagées en simulations parallèles réside dans le contrôle de l'ordre d'accès aux variables. Par exemple, toute opération d'écriture sur une variable partagée $\mathcal{X}$ à l'instant simulé t_e doit être exécutée après les opérations de lecture de $\mathcal{X}$ ayant lieu à des instants simulés $t_l < t_e$. Dans une simulation parallèle il est possible que les processus évoluent de manière asynchrone. Comme alors les accès aux variables partagées risquent de s'effectuer de façon desordonnée, il devient nécessaire d'imposer un ordre sur ces accès. Des solutions pour la mise en œuvre de variables partagées en simulations parallèles sur des machines sans mémoire commune sont présentées en détail dans [Meh93].

Par conséquent, dans nos modèles, les interactions entre processus s'effectuent uniquement par des échanges de messages, à travers de canaux de communication unidirectionnels *fifo* (*first in, first out*) non bornés. Chaque processus possède un ensemble de canaux d'entrée, sur lesquels il peut recevoir des messages, et un ensemble de canaux de sortie, sur lesquels il peut en envoyer (figure 2.1). Les messages reçus restent sur les canaux d'entrée tant qu'ils ne sont pas consommés par le processus récepteur. Les *prédécesseurs* d'un processus sont les processus directement liés à ses canaux d'entrée, desquels il reçoit des messages ; ses *successeurs* sont les processus directement liés à ses canaux de sortie, auxquels il envoie des messages.

La topologie du programme de simulation peut être statique ou dynamique. Dans le premier cas, le nombre de processus est constant et les canaux de communications sont fixes. Les processus et les canaux de communication définissent alors un graphe de communications statique, qui peut être utilisé par les algorithmes de synchronisation. Un exemple classique de ce type de système sont les réseaux de files d'attente, constitués d'un nombre fixe de serveurs (processus) et d'un nombre variable des clients (messages) qui circulent sur le réseau, par des chemins établis à priori [RW89]. Dans les systèmes à topologie dynamique, des processus peuvent être créés ou détruits, et les interactions entre eux sont arbitraires et dynamiques. Le réseau d'interconnexion n'a

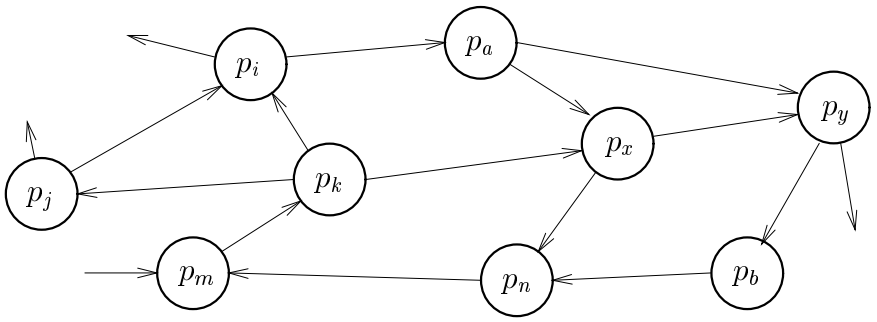

FIG. 2.1 – Réseau de processus d'un modèle de simulation

pas de structure fixe.

Dans un simulateur séquentiel, l'écoulement du temps simulé est représenté par une horloge (valeur monotone croissante) mise à jour avant chaque traitement d'événement[1]. Cette horloge unique, gérée par le noyau de synchronisation, sert de référence temporelle à tous les processus du programme de simulation. Pour garantir la cohérence de la simulation dans un contexte parallèle, les processus doivent avoir cette même perception unique du temps simulé. De plus, comme les interactions entre les entités du modèle sont en général instantanées, les transferts de messages qui les représentent lors de la simulation doivent avoir des durées de transfert nulles dans le temps simulé : un message émis à la date t dans le temps simulé sera perçu par son destinataire, sur un de ses canaux d'entrée, à cette même date t.

Dans la suite de ce chapitre nous étudierons les techniques de synchronisation employées pour la mise en œuvre de simulations parallèles prenant en compte la distribution du modèle de simulation. Des solutions synchrones et asynchrones pour des simulations dirigées par le temps ou par les événements seront abordées.

2.3 Simulation dirigée par le temps

Comme vu dans la section 1.2, cette technique consiste à faire évoluer l'horloge locale de chaque processus par des petits pas de durée δ, que nous appellerons ici *pulsations*. Un processus à l'instant t traite ses événements correspondants à cet instant, pour ensuite attendre l'avancement de son horloge à $t + \delta$.

Ce schéma d'exécution coïncide avec le concept d'*algorithme réparti synchrone* proposé par Awerbuch dans [Awe85] et developpé dans [HR88] : un algorithme réparti synchrone est composé d'un ensemble de processus s'exécutant sur un support d'exécution synchrone caractérisé par :

– Une horloge globale unique, servant de référence temporelle commune aux processus de l'algorithme synchrone. Chaque cycle de cette horloge correspond à une pulsation.

[1] Comme événements nous considérons les actions internes au processus, dus à sa logique propre, et les traitements des messages disponibles sur ses entrées, qui correspondent à des interactions avec les autres processus.

– Un service d'échange de messages dont le délai de transfert est inférieur à une pulsation d'horloge : un message émis à la pulsation p sera reçu par son destinataire pendant cette même pulsation p.

Un processus envoie au plus un message vers chaque successeur, par pulsation. Ces messages sont émis au début de chaque pulsation ; ensuite les messages concernant la pulsation actuelle sont reçus et traités (cela garantit que la réponse à un message reçu à la pulsation p ne sera pas envoyée avant la pulsation $p+1$). Lorsque l'horloge globale passe à la pulsation $p+1$, chaque processus sait que tous les messages qu'il a émis à p ont été reçus par leurs destinataires [HR88].

Pour permettre l'exécution d'algorithmes synchrones sur des supports d'exécution asynchrones, Awerbuch [Awe85] introduit la notion de *synchroniseur* : un algorithme réparti (asynchrone) construisant un support d'exécution synchrone, comme caractérisé ci-dessus, à partir d'une machine parallèle asynchrone. Plusieurs solutions pour la mise en œuvre d'un synchroniseur sont possibles [Awe85, HR88, Ada90]. Nous pouvons les classer dans deux catégories :

– *Synchroniseur fort* : les processus progressent de concert : à tout moment ils se trouvent à la même pulsation.
– *Synchroniseur faible* : les processus progressent de façon asynchrone ; chaque processus a une perception locale de l'horloge globale, pas forcément égale à celles des autres processus (ceux-ci sont donc logiquement synchrones mais peuvent être physiquement asynchrones).

Sur un synchroniseur fort, tous les processus de l'algorithme synchrone progressent à la même vitesse. Un processus ne peut passer à $p+1$ que lorsque tous les processus ont exécuté leur codes correspondants à la pulsation p et qu'ils ont reçus tous les messages envoyés à p. La mise en œuvre de ce type de synchroniseur peut être facilement effectuée à l'aide d'un processus contrôleur central, qui coordonne la mise à jour des horloges locales des processus. Une mise en œuvre entièrement répartie est aussi possible, en faisant usage de diffusions (*broadcasts*) de messages [RW89].

Le synchroniseur fort peut s'avérer utile pour la mise en œuvre d'une simulation parallèle lorsque celle-ci interagit avec le monde extérieur (acquisition de données, interactions avec l'utilisateur, etc.). Comme tous les processus sont au même instant dans le temps simulé, l'état global de la simulation est facilement obtenu. Une autre avantage de ce type de synchronisation est la complexité peu élevée en temps et nombre de messages pour son implémentation : dans une mise en œuvre avec contrôleur central, il suffit de deux transferts de messages de contrôle par processus pour que tout le système passe à la pulsation suivante (c'est à dire, $2n$ messages de contrôle et 2 unités de temps pour avancer les horloges locales des n processus). Toutefois, cette approche ne permet pas de bien exploiter le parallélisme potentiel du modèle, car deux événements potentiellement concurrents ne seront traités en parallèle que s'ils se trouvent à la même pulsation.

Avec un synchroniseur faible, la divergence entre les horloges locales (perceptions locales de l'horloge globale) de deux processus distants[2] de d dans le réseau est toujours égale ou inférieure à d (figure 2.2) : un processus à la pulsation p peut passer à $p+1$ s'il

[2]La distance entre deux processus correspond au nombre de canaux sur le chemin orienté le plus court les reliant.

a reçu tous les messages que lui ont été envoyés jusqu'à p et si tous ses prédécesseurs sont au moins à p. Ce type de synchroniseur peut être implémenté de plusieurs formes [HR88, Ada90], dont la plus simple est connue sous le nom de *synchroniseur* α. Cette mise en œuvre consiste à faire en sorte qu'un processus, dès qu'il a finit l'exécution correspondante à la pulsation actuelle, communique ce fait à tous ses successeurs. Un processus peut passer à la pulsation $p+1$ dès qu'il sait que tous ses prédécesseurs ont finit de traiter la pulsation p.

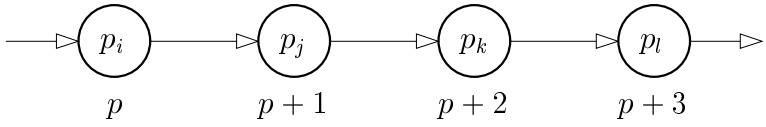

FIG. 2.2 – Synchronisation faible

Les synchroniseurs faibles permettent de mieux exploiter le parallélisme potentiel du modèle de simulation, en permettant à chaque processus de faire évoluer son horloge locale selon ses besoins. Les résultats obtenus par Adam [Ada90] semblent confirmer cela, lors de l'expérimentation d'algorithmes synchrones sur une machine parallèle sans mémoire commune. Il a pû constater que, plus le degré de synchronisation offert par un synchroniseur aux processus de l'application est fort, moins bonnes sont ses performances en temps d'exécution.

Comme pour le cas séquentiel, la simulation parallèle dirigée par le temps ne présente un intérêt que lorsque la densité d'événements à traiter est régulière. Il est important aussi de remarquer que cette méthode ne permet pas de simuler des processus ayant des réactions instantanées (envoyer à la pulsation p la réponse à un message qu'il a reçu à p), car les messages qu'un processus reçoit pendant une pulsation ne seront répondus qu'au début de la pulsation suivante.

2.4 Simulation dirigée par les événements

Dans un simulateur séquentiel dirigé par les événements, le noyau de synchronisation fait avancer le temps simulé jusqu'à la date du prochain événement, le traite et recommence le cycle sur l'événement suivant. Les événements sont donc traités selon un ordre chronologique, ce qui assure le principe de causalité établi dans le chapitre 1.

Dans un contexte parallèle, les processus se conduisent comme des simulateurs séquentiels indépendants. Chaque processus doit déterminer son prochain événement à traiter, à partir de son état local et des messages contenus dans ses canaux d'entrée. Des solutions synchrones et asynchrones sont possibles pour l'implémentation d'un tel simulateur.

Dans une mise en œuvre complètement synchrone, les processus progressent de concert, de manière semblable à celle de la simulation dirigée par le temps avec un synchroniseur fort. Chaque processus détermine la date de son prochain événement à traiter et la communique à un processus contrôleur. Celui-ci calcule le minimum des dates reçues et l'envoie de retour aux processus, qui peuvent alors avancer jusqu'à

cette date minimum. Comme pour les synchroniseurs forts, une mise en œuvre sans processus contrôleur est aussi possible, à l'aide de diffusions. Certaines améliorations sont possibles sur ce schéma de base : considérer seulement les événements d'interaction entre processus pour le calcul de la date minimum, utiliser une structure hiérarchique arborescente de processus ou du matériel spécifique pour accélérer le calcul des minima [RW89, Fil92]. Cette approche ne semble intéressante que dans des modèles avec peu d'interactions entre processus et beaucoup d'événements se produisant en même temps. D'autres auteurs [Aya89b, Lub89] proposent des solutions synchrones ayant toutefois un degré de synchronisme inférieur à celui de la mise en œuvre avec contrôle centralisé. Nous les étudierons dans la section 2.7.

Dans une mise en œuvre asynchrone chaque processus progresse de façon autonome, pour mieux tirer profit du parallélisme potentiel du modèle. Il n'existe donc pas une horloge unique, commune à l'ensemble du modèle, pour indiquer le temps simulé ; chaque processus p_i possède une horloge locale notée hl_i qui lui sert de référence dans le temps simulé. Comme à tout moment les processus peuvent se trouver à des instants différents dans le temps simulé, des décalages sont possibles entre les horloges locales, et il devient nécessaire d'estampiller chaque message émis avec sa date d'émission. Par la suite, un message m émis à la date t_m sera représenté par le couple $[m, t_m]$.

La mise à jour des horloges locales des processus doit être effectuée par le noyau de synchronisation, de façon à assurer une progression cohérente de ces horloges et donc de la simulation. Il est démontré dans [CM79, Mis86] que si chaque processus prend connaissance des messages sur ses canaux d'entrée (*i.e.* des interactions avec ses prédécesseurs) dans l'ordre de leurs dates d'émission, le principe de causalité est respecté localement, et par conséquent la simulation dans son ensemble respecte aussi ce principe. Notons l'importance de cette affirmation : à partir d'une règle de contrôle purement locale, applicable par chaque processus, il est possible de garantir le respect de la causalité par toute la simulation. Cela rend possible la désynchronisation entre les processus, sur laquelle reposent les principales techniques de simulation que nous regarderons par la suite.

Les canaux de communication entre processus sont considérés *fifo*, ce qui assure un ordre correct pour les messages circulant entre deux processus. Toutefois, les processus évoluent de façon asynchrone, et les durées réelles des transferts de messages entre processus ne sont pas définies à priori. En conséquence, un processus ayant plusieurs prédécesseurs peut recevoir des messages sur ses canaux d'entrée de façon désordonnée, pas forcément dans l'ordre croissant de leurs dates d'émission. Dans l'exemple de la figure 2.3, le processus p_i peut recevoir un message estampillé 3 en provenance de p_j ; pour une simulation correcte, ce message doit être pris en compte avant les messages m_a et m_b, déjà arrivés à p_i.

Pour assurer une évolution cohérente de la simulation, sans violer le principe de causalité, deux stratégies sont envisageables :

- *Stratégie Optimiste* : les messages sont traités par le processus dans l'ordre d'arrivée. Si une violation locale du principe de causalité est détectée, par l'arrivée d'un message $[m, t_m]$ lorsque $hl_i > t_m$, un retour en arrière de la simulation est effectué. Cette stratégie, aussi appelée *cohérente à posteriori*, est fondée sur la notion de temps virtuel énoncée par Jefferson [Jef85].
- *Stratégie Pessimiste* : cette stratégie, aussi nommée *cohérente a priori* ou *conser-*

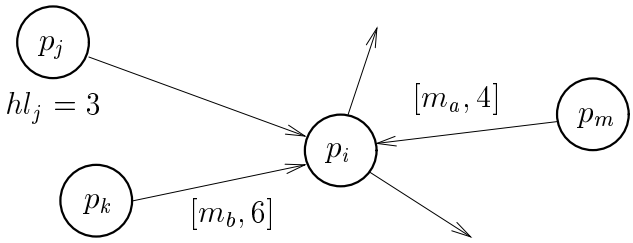

FIG. 2.3 – Cohérence causale

vative, consiste à assurer que les messages seront délivrés aux processus dans l'ordre de leurs estampilles. Ainsi, un message n'est délivré à son récepteur que s'il a déjà reçu tous les messages antérieurs à celui en question. Les mécanismes de base de cette stratégie impliquent la mise en attente des processus, ce qui peut mener à des situations d'interblocage ; cela impose l'utilisation de techniques supplémentaires pour éviter [CM79] ou détecter et résoudre [CM81, Mis86] ces interblocages.

Fondées sur ces stratégies, plusieurs méthodes ont été proposées pour la mise en œuvre de simulations parallèles. Nous les regarderons avec plus de détails dans les sections qui suivent.

2.5 Stratégie pessimiste

Les méthodes pessimistes ont été les premières à être proposées dans le cadre de la simulation parallèle [CM79, CM81]. Elles consistent à employer des schémas de synchronisation spéciaux pour assurer que les messages soient délivrés aux processus destinataires dans l'ordre de leurs estampilles.

La circulation de messages entre deux processus obéit à l'ordre *fifo* ; par conséquent, l'estampille du dernier message reçu sur un canal établit une borne inférieure pour l'estampille du prochain message à recevoir sur ce canal. Cette date minimum est nommée *temps_ canal*. Le minimum des *temps_ canal* sur tous les canaux d'entrée d'un processus définit l'estampille minimum pour le prochain message à arriver à ce processus. Tous les messages ayant une estampille inférieure ou égale à ce minimum peuvent alors lui être délivrés. Dans la figure 2.4, le minimum des *temps_ canal* est 5 ; les messages m_a, m_d et m_b peuvent être délivrés à p_i, dans cet ordre, tandis que m_c et m_e restent bloqués.

Dans ces conditions, un processus peut se trouver bloqué, en attendant l'arrivée d'autres messages pour faire avancer les *temps_ canal* et ainsi pouvoir délivrer les messages qu'il a déjà reçu. Un processus se met en attente dès que son canal d'entrée ayant le plus petit *temps_ canal* est vide. Cela peut mener à une interblocage de toute la simulation, même si le modèle simulé ne présente pas d'interblocage. La figure 2.5 illustre cette situation : à cause de l'anneau de canaux vides, aucun des processus ne peut consommer le message présent sur son entrée. L'arrivée d'autres messages ne changera pas cette situation. Deux approches sont envisageables pour résoudre ce problème : soit *éviter* l'avènement des situations d'interblocage, soit *détecter et résoudre* ces situations.

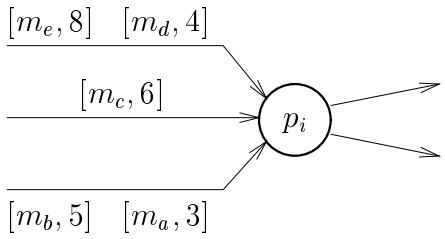

FIG. 2.4 – Livraison de messages

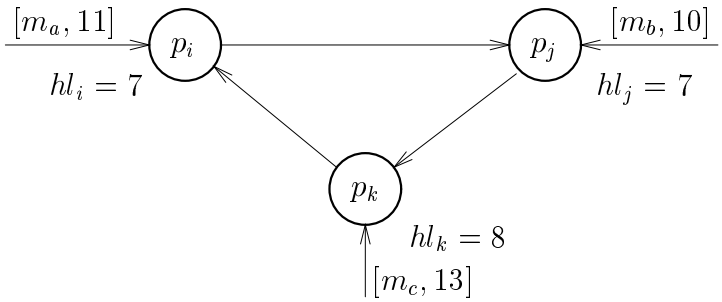

FIG. 2.5 – Interblocage dans la simulation

2.5.1 Prévention des interblocages

Cette méthode a été proposée indépendamment par Chandy et Misra [CM79] et Bryant [Bry77]. Elle fait usage de messages de contrôle, appelés *messages null*, pour éviter que des situations d'interblocage ne se produisent. Ces messages servent uniquement aux besoins de synchronisation du simulateur ; ils ne correspondent pas à des activités du modèle simulé.

Le mécanisme de base de cette méthode est assez simple : chaque fois qu'un processus envoie un message $[m, t_m]$ sur un canal de sortie (lors de l'émission, $t_m = hl_i$), il envoie simultanément sur les autres canaux de sortie des messages $[null, hl_i + \delta]$. Ces messages de contrôle ont pour but de faire avancer les *temps_ canal* des canaux de sortie, et par conséquent assurer aux processus successeurs qu'aucun message ne leur sera envoyé par le processus émetteur avant la date contenue dans les estampilles. Lorsqu'un processus reçoit un message de contrôle $[null, t_{null}]$, le *temps_ canal* du canal concerné est mis à jour ($tc = t_{null}$), et le message *null* est effacé. Le minimum des *temps_ canal* est ensuite réévalué, pour permettre la livraison des messages de la simulation arrivés. L'arrivée d'un message *null* à un processus peut faire avancer son horloge locale. Il doit alors communiquer à ses successeurs, toujours à l'aide de messages *null*, les nouvelles dates minimum pour l'envoi de messages de la simulation.

La valeur δ, ajoutée à l'estampille du message *null*, a une importance fondamentale pour le bon fonctionnement de cette méthode. En envoyant un message $[null, hl_i + \delta]$, $\delta \geq 0$, le processus communique à son successeur une *prévision* (ou *lookahead*) sur son comportement futur : il n'enverra aucun message avant δ unités de temps simulé. Cette prévision est calculée sur l'état actuel du processus, à partir de données comme la durée

minimum du traitement de chaque message, etc. Dans les modèles de réseaux de files d'attente, par exemple, les prévisions peuvent correspondre aux temps minimum de service des serveurs.

Une condition minimale doit être satisfaite pour que ce schéma de prévention d'interblocages fonctionne : pour tout circuit fermé $\mathcal{C}$ dans le graphe d'interconnexion du modèle, la somme des prévisions δ des processus sur le circuit doit être non-nulle, à tout instant dans le temps simulé [CM79] : $\forall t\ \forall \mathcal{C}\ \sum_{p_j \in \mathcal{C}}\ \delta_j(t) > 0$.

Si nous considérons l'exemple de la figure 2.5, avec des durées de traitement des messages constantes et égales à 1 (*i.e.* $\forall t\ \forall p_i\ \delta_i(t) = 1$), une des séquences possibles pour la progression du modèle serait la suivante :

hl_i	hl_j	hl_k	Action
7	7	8	p_i envoie $[null, 8]$ à p_j
7	8	8	p_j envoie $[null, 9]$ à p_k
7	8	9	p_k envoie $[null, 10]$ à p_i
10	8	9	p_i envoie $[null, 11]$ à p_j
10	10	9	p_j consomme $[m_b, 10]$

Cet exemple permet de constater que des prévisions trop petites peuvent dégrader beaucoup les performances de la méthode. Si nous avions considéré $\forall t\ \forall p_i\ \delta_i(t) = 0.001$, des milliers de messages *null* seraient nécessaires pour permettre la consommation du message m_b. Pour que cette méthode soit efficace, des prévisions de bonne qualité (aussi proches que possible de la vraie date du prochain envoi de message) sont nécessaires. Des prévisions optimales peuvent être obtenues si la durée de traitement des messages est fixe ou indépendante du type de message traité.

Un nombre élevé de messages *null* en circulation peut être constaté aussi lors de la simulation de modèles avec beaucoup de circuits et peu de messages de la simulation. Des schémas ont été suggérés pour tenter de diminuer la quantité de messages *null*, et augmenter l'efficacité de cette méthode. De Vries [DV90] propose la décomposition du réseau du modèle en sous-réseaux élémentaires, sur lesquels des algorithmes spécifiques seraient appliqués. Mühlhäuser [M88] propose l'emploi d'un algorithme réparti d'accélération, basé sur un jeton circulant dans un anneau virtuel mis en place sur l'ensemble des processeurs, pour mettre à jour plus rapidement les *temps_ canal* des canaux d'entrée des processus et ainsi éviter la prolifération de messages *null*. Misra [Mis86] et d'autres auteurs proposent des versions similaires d'une variation de la méthode des messages *null*, qui consiste à n'envoyer des mises à jour des *temps_ canal* que sur demande. Lorsqu'un processus se trouve bloqué, il sollicite du prédécesseur concerné (celui lié au canal avec le plus petit *temps_ canal*) la mise à jour du *temps_ canal*. Comme les demandes peuvent former des cycles, un mécanisme de traitement d'interblocages est ajouté.

2.5.2 Détection et résolution des interblocages

L'idée de base de cette solution est de permettre la simulation d'avancer librement. Périodiquement, le noyau de synchronisation vérifie si la simulation continue à progresser. Lorsqu'un interblocage est détecté, un algorithme de résolution est employé, pour déterminer quels processus peuvent être réactivés, et ainsi redémarrer la simulation.

La résolution des interblocages s'effectue selon un principe simple : un message peut être traité sans risquer de violer le principe de causalité seulement si aucun message ayant une estampille plus ancienne ne risque d'arriver. Ainsi, le message ayant l'estampille la plus ancienne dans tout le modèle peut toujours être délivré à son récepteur, sans avoir à attendre d'autres messages. Une solution simple pour résoudre un interblocage serait alors de déterminer le message le plus ancien dans toute la simulation et permettre à son récepteur de le consommer. Il est aussi possible d'appliquer cette solution à chaque cycle de processus interbloqués, comme celui de la figure 2.5. Toutefois, comme ce schéma reste encore peu efficace, d'autres techniques ont dû être proposées.

Pour la résolution efficace d'un interblocage, il est nécessaire de calculer, à partir des messages en attente, de la topologie du modèle, des horloges locales et des états des processus, la date minimum d'arrivée de nouveaux messages à chacun des processus du modèle. Les prévisions sur le comportement futur des processus ne sont pas essentielles au fonctionnement de cette méthode, mais peuvent en augmenter considérablement les performances [Fuj90]. Il faut aussi envisager la possibilité de détecter et résoudre un interblocage partiel, sur un sous-réseau du modèle, avant qu'il ne se propage et paralyse toute la simulation, avec des conséquences évidentes pour les performances.

L'application de cette méthode sur des systèmes à mémoire partagée s'avère simple, car un processus contrôleur peut prendre en charge la détection de l'interblocage et les calculs nécessaires au redémarrage de la simulation, sans coopération directe des autres processus. La mise en œuvre sur des machines sans mémoire commune présente plus de complexité. Les algorithmes classiques pour la détection des interblocages globaux [BT87, SMP88, Ray92] ou locaux [CJS87] dans les systèmes répartis peuvent être facilement adaptés au cas de la simulation.

Chandy et Misra [CM81] proposent un algorithme à phases pour la résolution des interblocages. Une fois l'interblocage détecté par un algorithme classique, un site contrôleur synchronise l'opération de résolution. Dans chaque phase, le rôle d'un processus consiste à envoyer à ses successeurs une mise à jour hypothétique de la date minimum de son prochain envoi de message, sans considérer ses canaux d'entrée vides. Ensuite, avec les dates reçues de ses prédécesseurs, il recalcule la sienne pour la phase suivante. Au bout d'un nombre fini de phases, les valeurs des *temps_canal* sont complètement à jour et au moins un processus peut redémarrer. Ce n'est pas une méthode très efficace, surtout dans des modèles avec peu de messages et beaucoup de circuits, où les interblocages sont fréquents. La figure 2.6 donne une idée du fonctionnement de cet algorithme ; nous avons considéré des prévisions égales à 1 pour les processus ($\delta_i = \delta_j = \delta_k = 1.0$). D'abord, chaque processus informe son successeur de la date de son prochain envoi de message, s'il ne reçoit pas de nouveaux messages (étape $\mathcal{A}$). Au cour des étapes suivantes, les dates de prochain envoi de message sont réévaluées jusqu'à ce qu'elles soient stables (étapes $\mathcal{B}$, $\mathcal{C}$, $\mathcal{D}$, etc.). Finalement, les messages m_a et m_b peuvent être délivrés à leurs destinataires, et la simulation reprend son cours.

L'approche de Grošelj et Tropper [GT91] considère l'existence de plusieurs processus du modèle par site de la machine à mémoire distribuée. Sur chaque site, le noyau de synchronisation parcourt le sous-réseau local de processus et fait avancer au maximum les *temps_canal* des canaux localisés sur le site. Cela permet de résoudre les interblocages locaux avant qu'ils ne se propagent aux autres sites. Les interblocages globaux ne sont pas explicitement détectés : lorsque tous les processus sur un processeur se

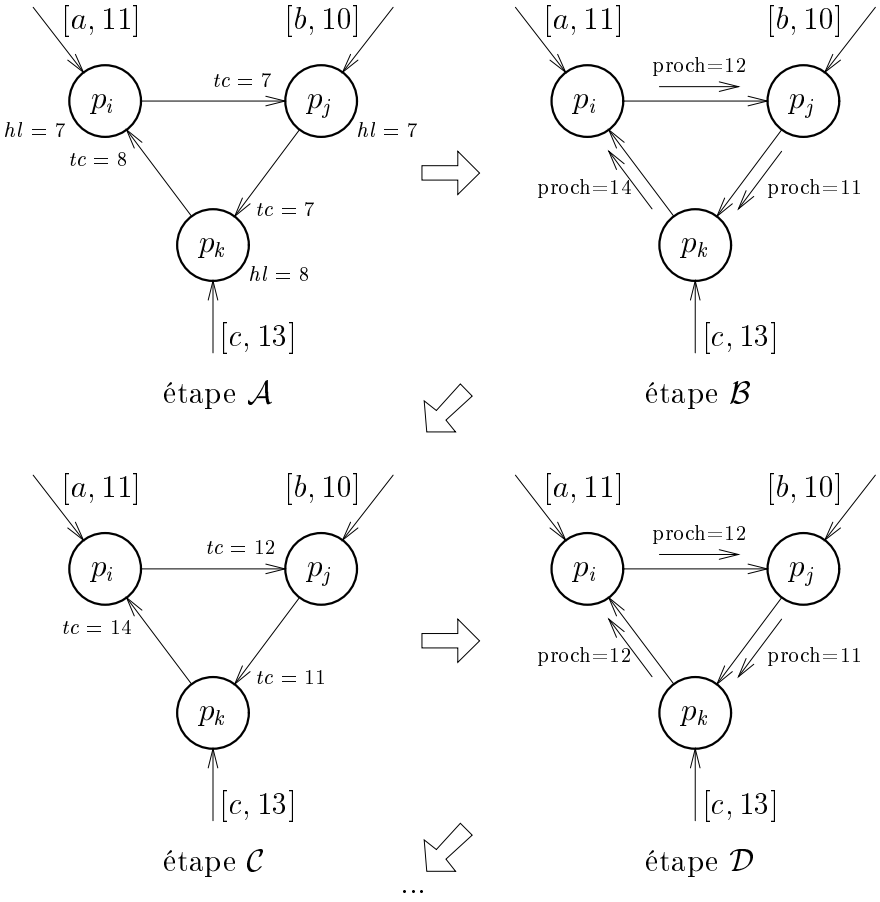

FIG. 2.6 – Résolution d'un interblocage

trouvent bloqués (et que le calcul local n'a pas réussi à en débloquer), le processeur considère qu'il y a risque d'interblocage global et lance des requêtes à ses voisins pour tenter de résoudre l'interblocage.

2.6 Stratégie optimiste

Contrairement aux méthodes pessimistes, un processus de simulation s'exécutant selon une méthode optimiste n'attend pas que l'ordre sur les messages présents dans ses canaux d'entrée soit stable pour progresser. Dès que des messages sont disponibles il les traite, dans l'ordre de leurs estampilles. Dans ces circonstances, certains messages peuvent arriver en retard à leurs destinataires, *i.e.* après la date à laquelle ils auraient dû être traités, à cause de l'asynchronisme entre processus. L'arrivée, à un processus p_i ayant une horloge locale hl_i, d'un message $[m, t_m] \mid t_m < hl_i$, constitue une violation du principe de causalité : d'autres événements appartenant au futur de ce message ($t > t_m$) ont été traités avant lui. Dans l'exemple de la figure 2.7, le processus p_i reçoit et traite correctement les messages m_a et m_b, générant respectivement les messages m'_a et m'_b. Le message m_c arrive après m_a et m_b, toutefois il devrait être traité avant eux.

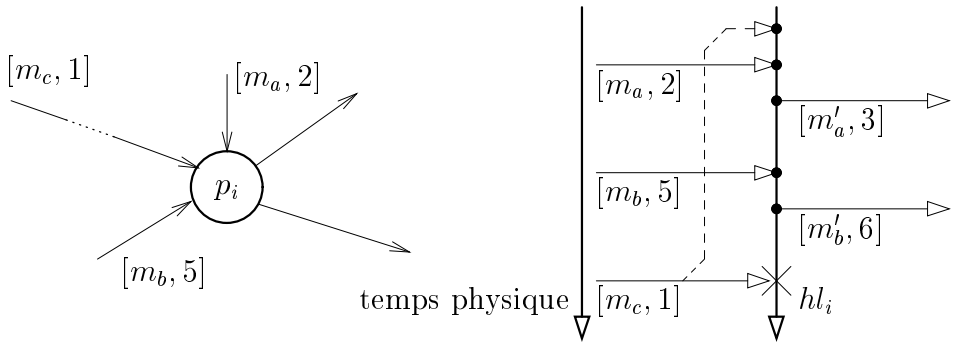

FIG. 2.7 – Violation de la causalité

Le mécanisme proposé par Jefferson [Jef85, JS85, J+87] et connu sous le nom *Time-Warp* est la plus connue et la mieux étudiée des approches optimistes. Dans cette méthode, la réparation d'une violation de la causalité due à l'arrivée d'un message $[m, t_m] \mid t_m < hl_i$, est concrétisée par le retour en arrière de l'exécution du processus, jusqu'à avoir $hl'_i \leq t_m$; la simulation est ensuite reprise, en considérant le nouveau message arrivé. Tous les traitements d'événements effectués pendant la partie incorrecte de la simulation, entre t_m et hl_i doivent être annulés, y compris les envois de messages (m'_a et m'_b, dans l'exemple de la figure 2.7).

Pour rendre possible le retour en arrière de la simulation sur un processus, son état doit être sauvegardé périodiquement, avant (ou après) chaque traitement d'événement. Les messages reçus ou envoyés doivent aussi être sauvegardés, afin de pouvoir restaurer complètement l'état du processus. Les états sont sauvegardés avec leur date, sous la forme de couples $[e_x, te_x]$, dans une liste ordonnée. Lors de l'arrivée à un processus p_i d'un message en retard $[m_r, t_{mr}] \mid t_{mr} < hl_i$, ce processus doit revenir sur son plus récent état $[e_x, te_x]$ ayant $te_x < t_{mr}$, comme le montre la figure 2.8.

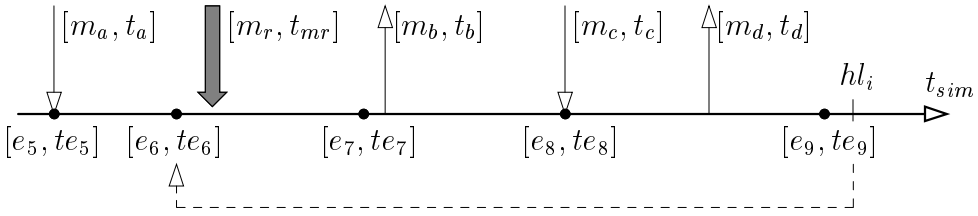

FIG. 2.8 – Retour en arrière de la simulation

Pour l'annulation des messages envoyés entre t_{mr} et hl_i, sont employés des messages de contrôle appelés *anti-messages*. Ainsi, pour l'annulation du message $[m_b, t_b]$, le processus doit envoyer l'anti-message $[-m_b, t_b]$ au même destinataire. L'arrivée, sur un canal d'entrée, d'un anti-message provoque la destruction de celui-ci et du message correspondant. Si le message à détruire a déjà été consommé par le récepteur, celui-ci revient aussi en arrière, de façon à annuler son traitement.

Deux stratégies peuvent être employées pour l'annulation des messages erronés.

La première, nommée *annulation agressive*, consiste à envoyer les anti-messages dès qu'une violation de la causalité est détectée, pour tenter d'atténuer la propagation des erreurs. Au contraire, la stratégie dite d'*annulation paresseuse* n'envoie des anti-messages que pour les messages devant effectivement être annulés. Une fois l'erreur causale détectée, le processus revient en arrière et reprend la simulation, sans envoyer des anti-messages. Pendant cette re-simulation, il compare les messages qu'il aurait dû envoyer avec ceux qu'il a envoyé pendant la simulation précédente. Les anti-messages sont alors envoyés seulement pour les messages effectivement erronés. En minimisant l'envoi d'anti-messages, cette stratégie peut éviter, pour les successeurs du processus en question, des retours en arrière sur des simulations correctes. Même si cette deuxième stratégie retarde l'envoi des anti-messages, ce qui peut favoriser la propagation des erreurs, elle semble généralement donner des meilleurs résultats que la stratégie agressive [Fuj90].

Une autre amélioration du mécanisme de base du retour en arrière est la *réévaluation paresseuse* [Fuj90]. En fait, si le traitement de l'événement fautif n'entraîne pas de modification dans l'état du processus (à part son horloge locale), la suite de la réexécution sera identique à l'exécution précédente. Dans ce cas, le processus peut défaire son dernier retour en arrière et avancer directement à l'état où il se trouvait avant l'arrivée du message en retard. Cette modification permet d'augmenter la performance de la méthode, au frais d'une plus grande consommation de mémoire pour la sauvegarde des états.

La sauvegarde des états et des messages reçus ou envoyés pendant la simulation, avec leurs dates, exige des quantités considérables de mémoire. Il est alors nécessaire d'effacer les états et les messages qui ne sont plus nécessaires à l'opération de retour en arrière. En effet, si l'on considère les horloges locales des processus et les messages non encore traités, il est possible de définir une date minimum avant laquelle aucun retour en arrière ne peut être effectué. Ce temps minimum, calculé périodiquement sur l'ensemble des processus de la simulation, est appelé *Temps Virtuel Global*, ou TVG. Tout état $[e_x, te_x] \mid te_x < TVG$ peut être oublié, de même pour les messages $[m, t_m] \mid t_m < TVG$ sauvegardés[3].

Il n'est pas nécessaire de sauvegarder l'état des processus après chaque événement ; des points de reprise plus ou moins espacés suffisent. Il faut toutefois noter que le coût d'un retour en arrière est proportionnel à l'espacement entre les points de reprise, car il faudra défaire et re-simuler sur une plus longue période. Dans la situation de la figure 2.9, un message $[m, t_m]$ arrive en retard à un processus p_i ($t_m < hl_i$), provoquant un retour en arrière au dernier état sauvegardé $[e_s, t_s]$. Le traitement du retour en arrière entre hl_i et t_m diffère de celui entre t_m et t_s :

- *Entre hl_i et t_m* : l'arrivé du message $[m, t_m]$ peut modifier l'évolution de la simulation dans cet intervalle, qui doit alors être défaite. Cela implique l'envoi d'anti-messages à d'autres processus, qui peuvent provoquer d'autres retours en arrière.
- *Entre t_m et t_s* : comme l'arrivé du message $[m, t_m]$ ne peut influencer son passé, l'évolution de la simulation dans cet intervalle ne change pas. Il est alors suffisant

[3]En réalité, il est nécessaire de garder le plus récent état $[e_x, te_x] \mid te_x < TVG$, ainsi que les messages $[m, t_m] \mid te_x \leq t_m < TVG$, pour le cas où un retour en arrière à TVG s'avérerait nécessaire.

de recalculer les états non sauvegardés, sans envoyer les messages produits ou des anti-messages. Cette phase ne provoque donc pas des retours en arrière sur d'autres processus.

Dans ces conditions le TVG ne décroît pas. Pour qu'il progresse, il faut assurer la correction du modèle du point de vue des propriétés de sûreté et de vivacité ; cela garantira alors la progression de la simulation dans son ensemble [JS85].

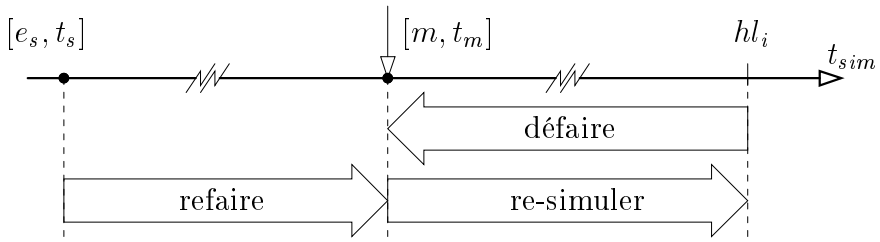

FIG. 2.9 – Traitement d'un retour en arrière

Le calcul du TVG permet de déterminer quelle partie de la simulation est définitive : toutes les actions s'effectuant à une date inférieure à TVG ne seront plus remises en cause. Cela est essentiel pour l'exécution d'actions ne pouvant pas être défaites, comme les opérations d'entrée/sortie, les calculs statistiques, etc. Une autre fonction importante du TVG est la détection de la fin de la simulation, une fois que $TVG > T_{max}$.

Le calcul du TVG, déclenché périodiquement, consiste à déterminer le minimum des horloges locales des processus et des estampilles des messages (et anti-messages) non encore traités, ou en transit au moment du calcul. Généralement ce calcul ne rend pas la valeur exacte du TVG, mais une date approximative $\tau \leq TVG$, suffisante pour l'utilisation qui en est faite Dans un système à mémoire partagée, le calcul du TVG ne présente pas de difficultés majeures. Pour les systèmes à mémoire répartie, il devient nécessaire l'utilisation d'un algorithme d'observation d'états globaux (*snapshot*), qui permet de déterminer un état global cohérent pour le système, à un moment donné [Ray92, Mat93].

Dans [CS89], Chandy et Sherman proposent une approche optimiste pour la simulation parallèle dirigée par les événements, fondée sur un mécanisme de relaxation similaire aux techniques utilisées pour la simulation de systèmes continus, et nommée *space-time simulation*. Ils considèrent la simulation comme un diagramme espace × temps, où les axes représentent respectivement les variables d'état du modèle de simulation et le temps simulé. Le but de la simulation est alors de remplir ce diagramme avec les valeurs possibles des variables d'état à chaque instant du temps simulé. Leur approche consiste à attribuer à chaque processus une région du diagramme qu'il doit remplir avec les valeurs des variables. Pour cela, le processus doit considérer les changements de comportement sur les frontières de sa région avec les régions voisines. Ces changements lui sont communiqués par les processus responsables des régions voisines, à travers des messages (cf. figure 2.10).

Le remplissage de chaque région s'effectue de façon itérative : le processus responsable reçoit les messages de ses voisins, recalcule sa région et renvoie des messages à ses

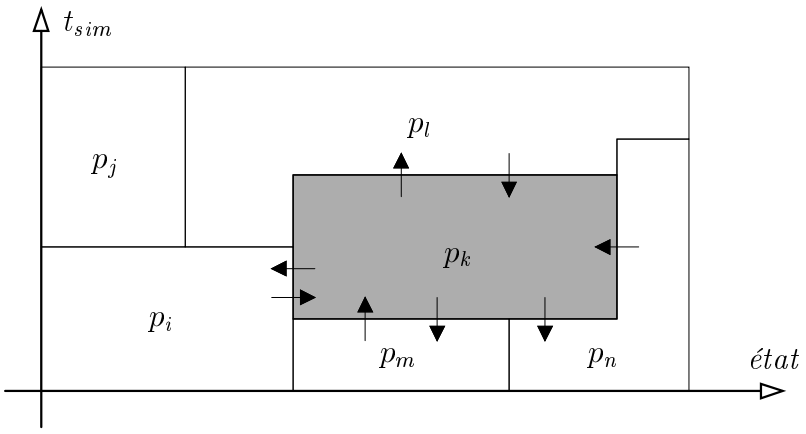

FIG. 2.10 – Régions dans le diagramme espace × temps

voisins, pour communiquer des changements d'état sur ses frontières. La simulation finit quand les modifications d'état s'arrêtent, *i.e.* quand les valeurs des régions convergent à une situation stable. Comme chaque processus ne remet en cause l'état de sa région que s'il reçoit des changement sur la frontière, et comme d'abord il recalcule le nouvel état de sa région pour ne communiquer qu'ensuite les changements à ses voisins, ce mécanisme est assez proche de celui de *Time-Warp* avec *annulation paresseuse* [Fuj90].

2.7 Solutions synchrones

Cette classe de solutions comprend notamment les propositions de Lubachevsky [Lub89] et d'Ayani [Aya89b], pour les machines à mémoire commune. Ces approches se fondent sur la notion de *distance* entre processus : la distance d_{ij} entre les processus p_i et p_j est définie comme le délai minimum de temps simulé entre un événement e_i sur p_i et son possible effet e_j sur p_j. Généralement les distances entre processus sont asymetriques ($d_{ij} \neq d_{ji}$) ; si aucune action de p_i ne peut influencer p_j, alors $d_{ij} = \infty$. Cette notion équivaut en quelque sort à celle des prévisions, dans les méthodes précédentes.

A partir de cette notion de distance, il est possible d'établir des relations de dépendance entre événements ayant lieu sur des processus distincts : deux événements estampillés $[e_i, te_i]$ et $[e_j, te_j]$ respectivement sur les processus p_i et p_j ne peuvent être exécutés en parallèle que s'ils ne peuvent être dépendants, c'est à dire, si $te_i + d_{ij} \geq te_j$ et $te_j + d_{ji} \geq te_i$. Si $te_i + d_{ij} < te_j$, e_i doit être traité avant e_j, car il peut produire des effets sur p_j avant e_j, et vice-versa. Ainsi sur la figure 2.11 : dans le cas $\mathcal{A}$ les deux événements peuvent être traités simultanément, tandis que dans le cas $\mathcal{B}$ le processus p_i devra attendre l'exécution de e_j sur p_j avant de continuer.

L'algorithme de base proposé par Ayani [Aya89b] comporte trois phases, séparées par des barrières de synchronisation :

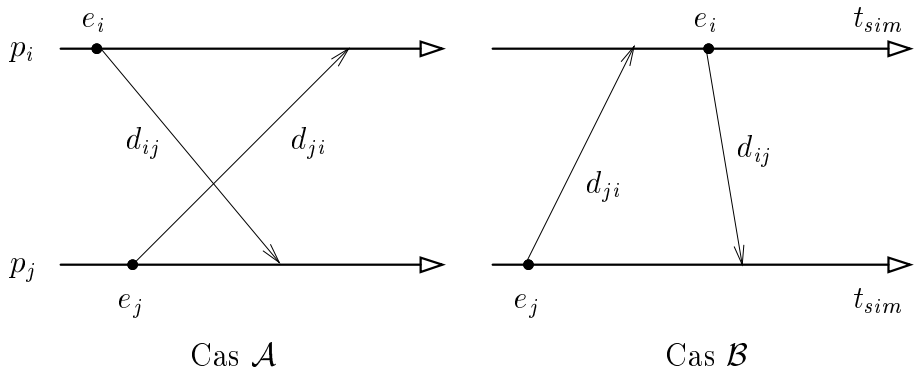

FIG. 2.11 – Dépendance entre événements

1. *Choix des candidats* : chaque processus choisit son prochain événement à traiter.
   — *Attendre les autres processus* —
2. *Invalidation* : chaque processus invalide les candidats des autres processus jugés dépendants de son candidat local, en utilisant les notions de dépendance citées ci-dessus. À la fin de cette phase, seuls les événements indépendants restent valides.
   — *Attendre les autres processus* —
3. *Exécution* : les processus ayant des événements valides les exécutent.
   — *Attendre les autres processus* —

Ces trois phases se succèdent jusqu'à la fin de la simulation. L'algorithme ne présente pas d'interblocage ; au pire, seulement l'événement avec l'estampille la plus ancienne restera valide après la phase d'invalidation, ce qui mène à une simulation séquentielle.

La phase d'invalidation requiert une forte utilisation de la mémoire commune, pour l'accès aux événements des autres processus et aux tableaux de distance entre processus, qui peuvent être dynamiques. De plus, cette phase demande un nombre élevé de comparaisons entre événements, de l'ordre de $\mathcal{O}(n^2)$. Des améliorations sont proposées par l'auteur pour traiter ces problèmes.

Un autre inconvénient de cet algorithme est de proposer au maximum un seul événement exécutable par processus, dans une phase. Dans [Aya89a], Ayani suggère le remplacement de l'événement candidat de chaque processus par une *fenêtre de temps*, à l'intérieur de laquelle tous les événements sont considérés valides. Dans la phase de choix des candidats, chaque processus p_i propose comme candidat une fenêtre de temps $[te_i, \mathcal{U}_i]$, où te_i est la date du prochain événement à traiter sur p_i et $\mathcal{U}_i$ l'extrémité supérieure de la fenêtre, initialement mise à ∞. Pendant la phase d'invalidation, p_i utilise les notions de distance et de dépendance entre événements pour faire diminuer les fenêtres des autres processus, en les « amputant » des parties qu'il juge dépendantes de son prochain événement à traiter : $\forall p_i \; \forall p_j \; \mathcal{U}_j = min(\mathcal{U}_j, te_i + d_{ij})$. À la fin de cette phase, chaque processus obtient une fenêtre d'exécution valide, qui peut contenir plusieurs événements à traiter. La figure 2.12 illustre ce mécanisme : après la phase d'invalidation, les fenêtres de temps des processus sont réduites aux zones hachurées. Tout événement à l'intérieur de ces zones peut être traité sans violer le principe de causalité.

Cette méthode est bien adaptée aux machines à mémoire physiquement partagée,

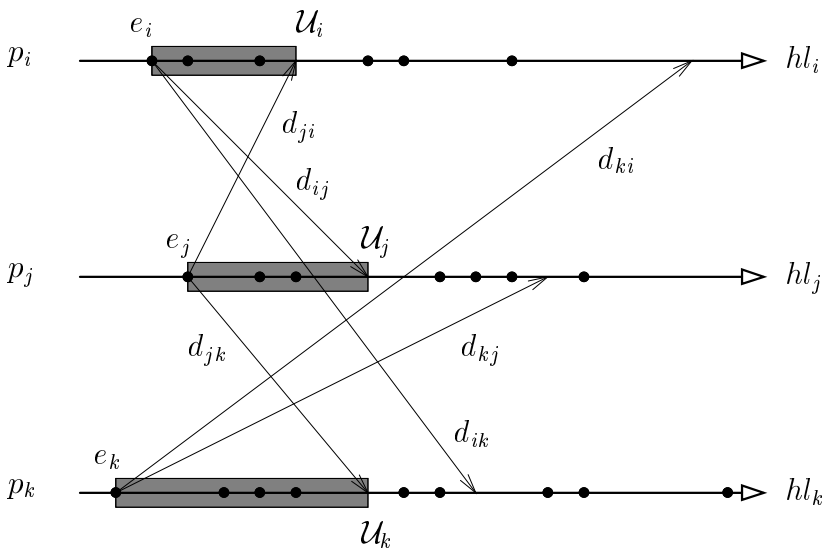

FIG. 2.12 – Fenêtres de temps valides

car la phase d'invalidation exige l'accès intensif à des données appartenant aux autres processus. Des essais de l'algorithme de base sur une machine à mémoire répartie offrant un service de mémoire virtuelle partagée [LP91] ont apporté des résultats peu encourageants [Had92]. Ces résultats sont en partie dus à la charge imposée par les protocoles de maintien de la cohérence de la mémoire virtuelle commune, mais la raison principale des mauvaises performances est sans doute l'excès de synchronisation exigé par cette méthode. Des bonnes performances peuvent toutefois être obtenues, dans des cas particuliers [Lub89, Aya91].

2.8 Approches hybrides

Plusieurs propositions ont été effectuées dans le sens de conjuguer les approches optimistes et pessimistes. Le point commun de ces propositions est de partir d'une approche pessimiste et permettre un certain degré d'optimisme aux processus. Dans [Meh91], Mehl suggère l'utilisation de calculs *spéculatifs* pour accélérer les méthodes pessimistes de simulation parallèle. Le principe de base est de permettre à un processus bloqué par l'algorithme de livraison de messages le traitement optimiste de ses messages d'entrée. Le processus effectue ses calculs sur une copie de son contexte et ne détruit pas les messages d'entrée consommés. Les messages produits par cette exécution optimiste sont retenus à l'intérieur du processus. Si l'avancement des *temps_ canal* des canaux d'entrée démontre la véracité de l'hypothèse optimiste, les messages produits sont envoyés, les messages consommés sont effacés et l'état optimiste devient effectif. Le cas échéant, le processus reprend son exécution normale. Cette stratégie semble particulièrement intéressante lorsque la charge de calcul dispensée au traitement des événements est élevée.

Le mécanisme proposé par Steinman [Ste91] combine la simulation dirigée par le

temps à un comportement local optimiste. La durée des pulsations est variable ; chaque processus définit une fenêtre de temps à l'intérieur de laquelle l'exécution est estimée causalement correcte. Il est permis aux processus d'avancer au delà de la date maximum définie par la fenêtre mais, comme dans la méthode antérieure, les messages produits attendent la vérification de respect de la causalité pour être envoyés. Les processus ayant trop avancé (situation détectée par l'arrivée d'un message en retard) effectuent des retours en arrière locaux, sans conséquence pour les autres processus.

2.9 Parallélisme potentiel

Bien évidemment, l'emploi d'un simulateur parallèle ne permet d'obtenir des gains en vitesse de calcul que pour des modèles présentant du parallélisme potentiel. Il est intéressant de connaître le degré de parallélisme offert par un modèle, pour deux raisons : d'une part, pour savoir si l'utilisation d'un simulateur parallèle peut apporter des gains en vitesse, et d'autre part pour permettre de dimensionner la machine aux besoins du calcul. En fait, comme le degré de parallélisme d'un modèle n'est jamais infini, le simulateur n'utilisera de façon efficace qu'un nombre limité de processeurs de la machine parallèle. L'ajout de processus supplémentaires ne fera pas accélérer davantage l'exécution. La mesure du degré de parallélisme offert peut servir aussi à l'étude de la performance des simulateurs parallèles, *i.e.* de leur capacité d'exploiter ce parallélisme.

Berry et Jefferson [BJ85] décrivent une méthode de mesure du parallélisme potentiel à partir de l'analyse de la trace d'une simulation séquentielle du modèle. Dans cette méthode, la trace de l'exécution séquentielle est analysée dans le but d'en déceler des événements causalement liés et ainsi constituer des chemins de dépendance causale (deux événements sont liés causalement s'ils ont lieu sur le même processus ou s'ils représentent l'envoi et la réception d'un même message).

En attribuant à chaque traitement d'événement et à chaque transfert de message une durée (dans le temps réel), le temps minimum nécessaire à la simulation du modèle sera donné par la durée du chemin causal le plus long. L'exécution représentée par le diagramme de la figure 2.13 illustre cette méthode. En faisant l'hypothèse que le transfert d'un message (indiqué par une flèche) et son traitement (indiqué par un rectangle) consomment chacun une unité de temps de calcul, l'exécution représentée par le diagramme n'exigerait pas moins de 9 unités de temps de calcul, *i.e.* la durée du chemin causal le plus long, en gris sur le diagramme. Un simulateur séquentiel demandera 11 unités de temps de calcul pour cette même exécution (tous les événements seront traités de façon séquentielle ; les transfert de messages sont considérés de durée nulle, car ils s'effectuent sur un même processeur). De cela nous pouvons conclure que l'accélération maximum permise par cette exécution est de $(11 - 9) \div 11 \approx 18\%$, peu importe le nombre de processeurs utilisés.

Lorsque la complexité du modèle ou de son comportement empêche l'analyse du chemin critique sur une trace de l'exécution, la méthode proposée par Righter et Walrand [RW89] peut s'avérer utile. Elle consiste à effectuer cette analyse « à la volée », *i.e.* à calculer la longueur maximale des chemins de dépendance causale en même temps que l'exécution de la simulation parallèle. Pour cela, chaque message ou événement est étendu d'une deuxième estampille qui transporte la durée (dans le temps réel) du

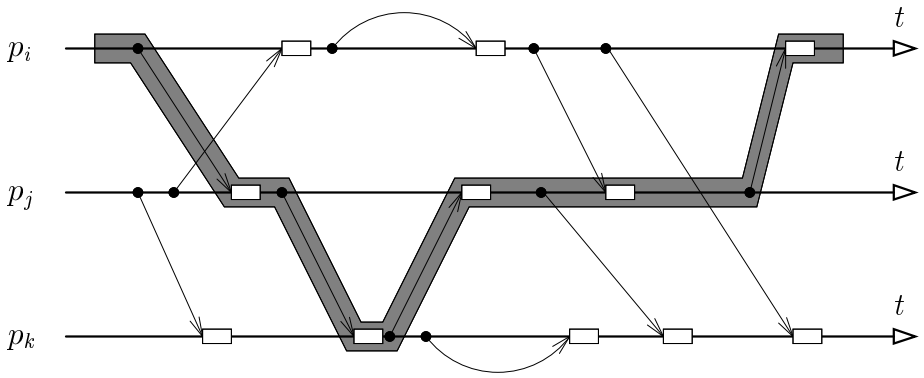

FIG. 2.13 – Mesure du parallèlisme potentiel

chemin causal le plus long qui mène à l'événement ou au message en question. Ces estampilles sont calculées en fonction des estampilles des messages déjà traités par le processeur, et des durées de transfert et de traitement des messages. Les derniers événements traités dans la simulation auront alors comme estampille la durée du chemin causal le plus long, *i.e.* la durée minimum de temps de calcul nécessaire pour la simulation de ce modèle.

La figure 2.14 détaille le fonctionnement de cette méthode, en considérant l'émission d'un message $[m, t_m]$ par un processus émetteur p_e et sa réception et traitement par le récepteur p_r. La durée de transfert des messages est donnée par δ_t et la durée de leur traitement (dans le temps réel) par δ_c. Si tr, déterminé à partir des traitements antérieurs, est l'instant au plus tôt auquel le message $[m, t_m]$ aurait pû être émis, il sera reçu par p_r à $tr + \delta_t$. Étant donné tr_0 l'instant au plus tôt de la fin du traitement du dernier événement avant le message $[m, t_m]$, la date de fin de son traitement sera $tr_1 = max(tr_0, tr+\delta_t)+\delta_c$. Cette date servira à estampiller le prochain envoi de message (m') et à mettre à jour la valeur tr_0, pour la suite du calcul.

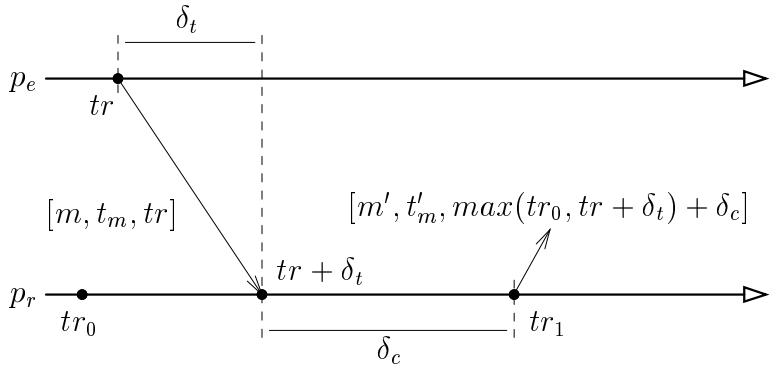

FIG. 2.14 – Mesure « à la volée » du parallèlisme potentiel

Le lecteur pourra constater la proximité entre ce mécanisme et celui des horloges logiques de Lamport [Lam78], utilisées pour le calcul d'une approximation de la relation

de causalité entre les événements d'une exécution d'un algorithme réparti. C'est en réalité un mécanisme similaire, dans lequel les incréments unitaires sont remplacés par des incréments exprimés en termes de durée de calcul ou de durée de transfert de messages.

2.10 Conclusion

Dans ce chapitre nous avons présenté les principales propositions pour la parallélisation de simulations à événements discrets. Notre attention s'est concentré sur la voie de parallélisation la plus prometteuse, la distribution du modèle de simulation. Dans cette voie, nous avons aussi bien examiné les stratégies de simulation parallèle dirigées par le temps que celles dirigées par les événements. Dans le cadre des stratégies dirigées par les événements, nous avons regardé en détail quatre approches : pessimiste avec prévention d'interblocages, pessimiste avec détection et résolution d'interblocages, optimiste *Time-Warp* et synchrone.

Les méthodes optimistes ont été plus évaluées que les pessimistes ; les résultats trouvés montrent que les meilleures performances des simulateurs parallèles sont généralement obtenues avec des telles méthodes [J+87, Fuj87, RW89, Fuj90, Pre90]. Ces méthodes montrent d'autres caractéristiques intéressantes, comme l'utilisation implicite des prévisions, et une adaptation plus facile des mécanismes de synchronisation aux modèles à topologie dynamique. Le plus grand handicap de ces méthodes semble être la sauvegarde des états, pour rendre possible les retours en arrière. Ce mécanisme est très gourmand en mémoire, et il ne semble être facilement et efficacement mis en œuvre que pour des processus ayant des états avec une structure simple, comme les réseaux de files d'attente [IM92, IMR93].

Les méthodes pessimistes présentent des besoins en mémoire nettement inférieurs à ceux des méthodes optimistes, et sont plus facilement adaptables à des modèles variés. Toutefois, leurs performances sont assez liées à la qualité des prévisions qui peuvent être fournies par le modèle [Wag91, IMR92, MR92]. Même l'approche de détection et résolution des interblocages, qui peut a priori fonctionner sans l'apport des prévisions, y est fort sensible [Fuj90]. Parmi les méthodes conservatives, la prévention des interblocages semble être la mieux adaptée aux machines sans mémoire commune. Étant données les difficultés imposées par la détection et la résolution des interblocages dans un contexte entièrement réparti, la méthode de détection et résolution d'interblocages semble mieux adaptée aux machines à mémoire partagée.

Pessimiste ou optimiste ? La réponse à cette question n'est absolument pas évidente, car elle dépend sensiblement des caractéristiques du modèle à simuler et de la machine choisie pour l'exécution. Si le modèle à simuler présente de bonnes possibilités de prévision, ou si la sauvegarde des états des processus peut poser des problèmes de volume ou complexité, une approche conservative est la plus indiquée. Par contre, si les prévisions sont de mauvaise qualité, une méthode optimiste peut donner des résultats plus intéressants, mais il faut tenir compte des difficultés pour la sauvegarde des états.

Le prochain chapitre exposera la mise en œuvre d'un noyau de système réparti dédié à la simulation de systèmes à événements discrets, sur le principe de la distribution du modèle. Ce noyau cherche offrir aux processus du modèle des primitives de communi-

cation et de synchronisation performantes, pour permettre leur évolution dans le temps simulé.

Chapitre 3

Un noyau pour la simulation répartie

L'objetif de ce travail de thèse est la construction d'un noyau de système réparti, nommé *Floria*, dédié au support de simulations à événements discrets. Notre but principal est de réfléchir sur la structure d'un tel noyau, dans ses aspects liés à la distribution du programme de simulation et à sa synchronisation, pour offrir un support d'exécution indépendant des modèles à simuler. Nous cherchons à faire en sorte que l'interface offerte aux applications soit homogène et, dans la mesure du possible, indépendante des techniques de synchronisation employées. Les services offerts par ce noyau doivent rester assez généraux pour supporter la réalisation d'environnements correspondant à des langages ou des types de simulation divers.

Nous cherchons aussi à garantir une indépendance de la structure du noyau vis-à-vis des algorithmes de synchronisation employés, pour que l'expérimentation et l'évaluation de techniques de synchronisation pour la simulation répartie puissent être aisément effectuées. Dans le cadre de la simulation, une des principales raisons pour l'emploi de machines parallèles est le gain en vitesse que cela peut apporter. Pour cette raison, la mise en œuvre de ce noyau doit aussi prendre en compte l'efficacité des exécutions.

Pour la mise en œuvre du noyau *Floria* nous nous sommes restreints aux méthodes dites pessimistes (section 2.5). Au moment où nous avons fait ce choix, les méthodes optimistes (*Time Warp*) avaient fait l'objet de plusieurs réalisations et d'évaluations poussées, avec des résultats positifs [J+87]. Par contre, les méthodes pessimistes avaient été peu évaluées ; parmi les quelques travaux sur ce sujet on peut citer [Fuj87, Fuj88b, Fuj88a] où sont décrites des expériences sur des mises en œuvre d'algorithmes conservatifs sur un multiprocesseur à mémoire commune, et [MÖ88] qui présente un système de simulation répartie basé sur une méthode pessimiste. C'est pourquoi il nous a semblé intéressant d'investiguer cette voie en réalisant un système complet. De plus, l'efficacité des méthodes optimistes est très fortement liée à celle du retour en arrière, c'est à dire en fin de compte à des mécanismes de gestion de la mémoire, alors que nous nous intéressons plus aux algorithmes de contrôle et de synchronisation. Enfin, si des solutions efficaces pour le retour en arrière ont été proposées pour des applications particulières comme les files d'attente [Che87], ce mécanisme nous semble plus difficilement généralisable.

Comme machine cible pour la mise en œuvre du noyau, nous avons choisi l'hypercube iPSC/2 d'INTEL, dans une configuration à 32 nœuds. Chaque nœud de cette machine est constitué d'un processeur i386, d'un contrôleur de communications et de 4

Mo de mémoire locale. Un processeur hôte contrôle le chargement des applications et la récupération des résultats. Le système d'exploitation installé sur cette machine est un sous-ensemble d'UNIX, auquel ont été rajoutées des primitives pour la synchronisation et l'échange asynchrone et synchrone de messages entre processeurs. Ce service d'échange de messages est *fifo*, fiable et permet de considérer le réseau d'interconnexion entre processeurs comme étant complètement maillé. Le noyau *Floria* a été développé en utilisant le langage $\mathcal{C}$, et comporte environ 3000 lignes de code source.

Dans le présent chapitre nous décrivons d'abord l'environnement de simulation proposé aux applications, sous la forme de services offerts par le noyau. Ensuite, la mise en œuvre du noyau est détaillée. Finalement, des possibilités d'utilisation du noyau *Floria* sont exposées.

3.1 Un environnement de temps virtuel

Comme nous avons vu dans les chapitres antérieurs, dans une simulation les synchronisations ne se font pas par rapport au temps physique, mais dans un temps dit « simulé » ou « virtuel », dont l'évolution suit des règles spécifiques et sans lien direct avec le temps physique. Les machines multiprocesseur dont nous disposons sont synchronisées par rapport au temps physique, les rendant donc inadéquates à l'exécution directe des simulations. Le noyau *Floria* vise à construire, à partir d'une machine réelle asynchrone, une machine virtuelle ayant des caractéristiques adéquates pour supporter l'exécution de programmes de simulation.

Les applications (programmes de simulation) concernées par l'utilisation du noyau *Floria* ont une structure similaire à celle présentée dans la section 2.2, *i.e.* un ensemble de processus connectés par des canaux de communication unidirectionnels *fifo* selon une topologie quelconque. Par souci de simplicité, nous nous sommes restreints aux modèles à topologie statique. L'adaptation du noyau aux modèles à topologie dynamique implique l'ajout cohérent (du point de vue de la synchronisation dans le temps simulé) de nouveaux canaux et processus, problème auquel nous ne nous sommes pas intéressés pour le moment.

Dans la suite de cette section, nous présenterons l'environnement de temps virtuel mis à disposition des processus du programme de simulation par le noyau *Floria*.

3.1.1 La machine virtuelle offerte aux processus

Les processus du programme de simulation voient le noyau comme étant une machine virtuelle synchronisée par rapport au temps simulé. Les processus de l'application évoluent logiquement dans un temps virtuel dont ils ont tous la même perception, représentée par une horloge virtuelle globale *hv*. Le synchronisme logique des processus, qui doit permettre d'ordonner tous les événements se produisant dans le système, peut être synthétisé par la propriété suivante :

> $\mathcal{P}_1$: **Unicité du temps virtuel** : *le temps virtuel est unique et progresse (logiquement) à la même vitesse pour tous.*

Alors que les processus exécutent des actions qui ont une certaine durée (positive ou nulle) dans le temps virtuel, leurs interactions (*i.e.* transferts de messages) sont

caractérisées par la propriété suivante :

> $\mathcal{P}_2$: **Synchronisme des communications** : *dans le temps virtuel tout message émis à la date t est reçu par son destinataire à cette même date t.*[1]

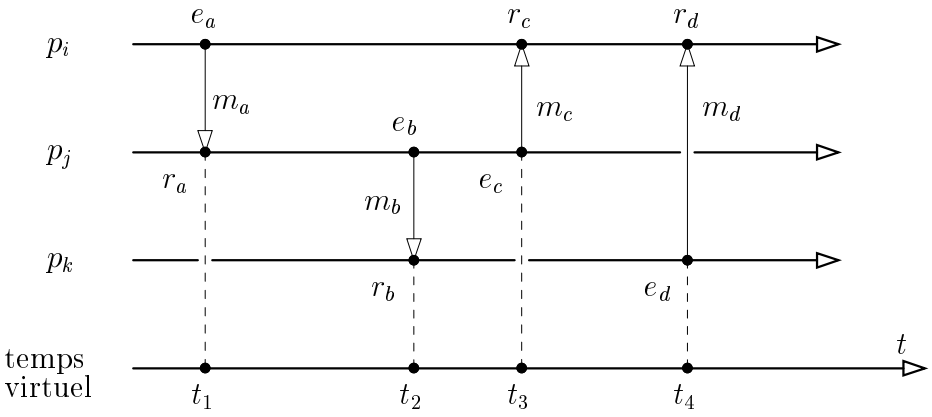

FIG. 3.1 – Communications synchrones dans le temps virtuel

Dans ce modèle les communications sont donc synchrones au sens de la propriété $\mathcal{P}_2$, comme montre la figure 3.1 (pour un message m_x, e_x indique son émission et r_x sa réception). Par contre, pour ne pas restreindre à un cadre trop étroit les applications qui seront supportées par le noyau, nous considérons que les processus sont asynchrones au sens suivant : si un message est reçu à la date t, son destinataire n'est pas obligé de le consommer à cette même date. Cela définit la propriété $\mathcal{P}_3$:

> $\mathcal{P}_3$: **Asynchronisme des processus** : *lorsqu'un message arrive à un processus (à la date t), celui-ci peut le consommer immédiatement ou à une date ultérieure ; la consommation effective n'est définie que par le comportement du processus (*i.e. *son programme).*

La propriété $\mathcal{P}_3$ est matérialisée de la façon suivante : tout message arrivé à un processus et non consommé est considéré comme « connu » par ce processus, puisque celui-ci peut, s'il le souhaite, les utiliser. Les messages reçus par un processus p_i et qu'il n'a pas encore consommés sont stockés, dans leur ordre d'arrivée et avec leurs dates d'émission, dans une file appelée $Fconnus_i$. La figure 3.2 représente l'état de la

[1] À la place de $\mathcal{P}_2$, la propriété $\mathcal{P}_{2'}$ suivante pourrait être considérée :

> $\mathcal{P}_{2'}$: **Synchronisme des communications** : *lorsqu'un message m est émis à la date virtuelle t, son émetteur précise la durée virtuelle δ_m associée à sa transmission ; le message m est reçu à la date virtuelle $t + \delta_m$ par son destinataire.*

Les propriétés $\mathcal{P}_2$ et $\mathcal{P}_{2'}$ sont équivalentes en ce qui concerne la puissance d'expression. En prenant $\forall m\ \delta_m = 0$, la propriété $\mathcal{P}_{2'}$ se réduit à $\mathcal{P}_2$. Si l'on considère $\mathcal{P}_2$, il est toujours possible de rajouter entre un processus émetteur et le destinataire associé un processus intermédiaire qui sert de tampon et y conserve le message m pendant la durée δ_m ; la propriété $\mathcal{P}_{2'}$ est alors obtenue entre l'émetteur et le destinataire.

file $Fconnus_i$ d'un processus p_i à l'instant t, lorsque celui-ci n'a consommé aucun des messages qu'il a reçus.

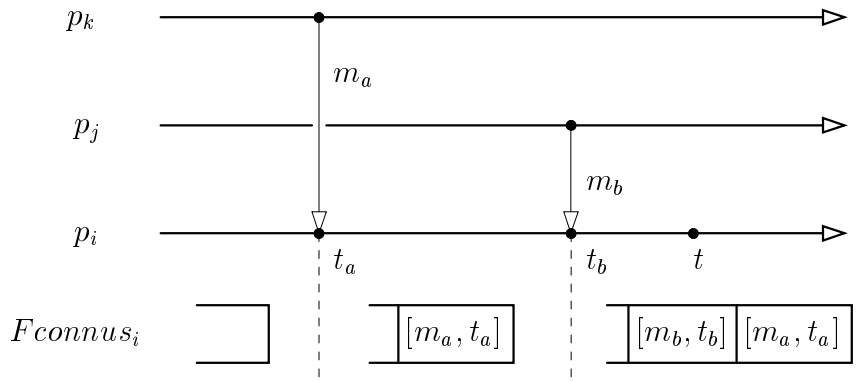

FIG. 3.2 – États de la file $Fconnus_i$

L'état d'un processus p_i de l'application, à un instant quelconque, est caractérisé par l'état de son contexte, la valeur de l'horloge globale hv et le contenu de sa file $Fconnus_i$.

3.1.2 Interface du noyau

Le noyau *Floria* est vu par chaque processus comme une machine offrant le temps virtuel défini par les propriétés $\mathcal{P}_1$, $\mathcal{P}_2$ et $\mathcal{P}_3$, définies dans la section antérieure. Ce temps virtuel est utilisé par les processus du programme de simulation à travers des services offerts par le noyau ; ces services, matérialisés par des primitives qui peuvent être appelées par les processus, sont les suivants :

- Lecture de l'horloge virtuelle hv, qui donne l'instant présent du processus dans le temps virtuel (cette lecture est effectuée par la primitive `Temps`).
- Attente jusqu'à ce que l'horloge hv ait atteint une valeur particulière t (à l'aide de la primitive `AttendreTemps(`t`)`).
- Attente, avec échéance à la date t, de l'arrivée d'un nouveau message dans $Fconnus_i$. Il s'agit de l'analogue dans le temps virtuel du service d'attente d'événement avec échéance, utilisé dans la plupart des systèmes temps-réel. Ce service est implémenté par la primitive (`AttendreMsg(`t`)`).

En plus des services directement liés au temps virtuel et à son avancement, d'autres services sont mis a disposition des processus. Pour accéder au contenu de sa file de messages connus $Fconnus_i$ (*i.e.* aux messages reçus et non consommés), le processus peut faire appel aux primitives suivantes :

- `Vide` : retourne la valeur `vrai` si $Fconnus_i$ est vide.
- `Premier` : retourne une référence sur le plus ancien message présent dans $Fconnus_i$, ou `nil` si la file est vide.
- `Dernier` : idem, pour le message le plus récent.
- `Prochain(`m`)` : idem, pour le message qui suit celui qui est référencé m dans la file.

- `Antérieur(`m`)` : idem, pour le message qui précède celui qui est référencé m.
- `Prendre(`m`)` : enlève de la file le message référencé par m.
- `Effacer(`m`)` : détruit le message référencé par m.

Les processus du programme de simulation se servent de primitives du noyau pour la création et l'envoi des messages qu'ils s'échangent. Chaque message est composé d'un corps, qui transporte des données appartenant à l'application, et d'un en-tête, géré par *Floria*, qui contient des informations génériques sur le message : type, date d'envoi (dans le temps simulé), canal d'arrivée. Ces informations sont mises à disposition des processus par primitives suivantes :

- `NouveauMsg(type)` : crée un nouveau message du type donné et rend un pointeur sur lui.
- `Envoyer(`m`,`cs`)` : envoi d'un message m sur le canal de sortie cs du processus.
- `MsgTemps(`m`)` : rend la date d'émission du message référencé m.
- `MsgCanal(`m`)` : rend le numéro du canal sur lequel le message m est arrivé.
- `MsgType(`m`)` : rend le type du message référencé m.

Pour la définition du programme de simulation, il est nécessaire d'établir le comportement de chacun des processus qui le composent et la topologie du réseau de canaux qui les relie. Les deux primitives du noyau *Floria* utilisées pour accomplir la description du programme sont les suivantes :

- `Processus(IP,S,C,NE,NS,TP)` : crée un nouveau processus sur le site `S` de la machine parallèle, avec `NE` canaux d'entrée et `NS` canaux de sortie. `IP` est l'identificateur numérique du nouveau processus, et le code de la procédure `C` définit son fonctionnement ; le noyau alloue au processus une pile d'exécution de taille `TP` octets.
- `Connexion(PE,CS,PR,CE)` : connecte le canal de sortie numéro `CS` du processus émetteur `PE` au canal d'entrée numéro `CE` du processus récepteur `PR`. Ainsi tout message envoyé par `PE` sur son canal de sortie `CS` sera déposé par le noyau sur le canal d'entrée `CE` du processus `PR`.

Une fois créés et leurs connexions effectuées, les processus n'ont pas besoin de connaître ni l'identité ni l'emplacement (sur la machine parallèle) des processus auxquels ils sont reliés. Chaque processus ne garde que l'identité de ses canaux d'entrée et de sortie, car les transferts de messages sont pris en charge par le noyau, qui s'occupe de résoudre les problèmes d'acheminement des messages.

Deux primitives supplémentaires peuvent être utilisées par les processus du programme de simulation. La première, `Prévision(`δ`)`, informe le noyau de la prévision δ du processus pour la date de son prochain envoi de message (cf. section 2.5.1). Finalement, la primitive `Fin` permet au processus de finir son exécution. En outre le noyau permet à chaque processus d'effectuer des manipulations sur son contexte (ses variables locales). Les actions exécutées par un processus sont de durée nulle dans le temps virtuel, à l'exception des services d'attente sur horloge ou sur message. Ces deux services, de par leur définition, « consomment » du temps virtuel.

3.2 L'architecture du noyau

Nous avons envisagé la mise en œuvre du noyau *Floria* sur une machine parallèle de type MIMD à mémoire répartie. Dans ce type de machine, chaque processeur dispose d'un espace mémoire exclusif et de canaux de communication lui permettant d'envoyer et de recevoir des messages des autres processeurs. La topologie du réseau de communications peut être quelconque ; en général le système d'exploitation propre à la machine fournit un service d'échange de messages permettant à un processeur d'envoyer des messages à n'importe quel autre processeur. L'échange de messages, avec des durées de transfert arbitraires mais finies, constitue la seule forme d'interaction entre les processeurs. Nous considérons que le service de transfert de messages offert par la machine est *fifo* et sans défaillances ; si cela n'est pas le cas, ces propriétés peuvent être assurées à l'aide de protocoles appropriés.

La réalisation, sur ce type de machine, d'un noyau offrant la notion de temps virtuel à une application sous la forme d'un réseau de processus communicants pose trois problèmes majeurs :

- D'abord, il faut distribuer l'application sur une machine ayant un nombre donné de processeurs et une topologie fixée. Comme le nombre de processus du programme de simulation peut être nettement supérieur au nombre de processeurs de la machine parallèle, plusieurs processus peuvent être mis sur le même processeur, et des processus voisins peuvent se trouver sur des processeurs distincts. Cette distribution doit être masquée par le noyau, de façon à rendre les échanges de messages et le partage des processeurs complètement transparents aux processus du programme de simulation.
- De par sa définition, la machine virtuelle offre à chaque processus de l'application un service de transfert de messages instantané dans le temps simulé (propriété $\mathcal{P}_2$). Ce service garantit qu'un message émis par un processus à une date t dans le temps simulé sera perçu par son destinataire à cette même date t. Pour accomplir cette tâche, le noyau doit estampiller les messages envoyés par les processus avec leurs dates d'émission, et contrôler leur livraison aux processus destinataires, en faisant usage des méthodes de synchronisation présentées dans le chapitre précédent.
- Finalement, il faut contrôler l'évolution des processus de façon à faire respecter les propriétés $\mathcal{P}_1$ et $\mathcal{P}_3$ du temps virtuel. La propriété $\mathcal{P}_1$ assure aux processus une vision unique et cohérente du temps virtuel, sous la forme d'une horloge virtuelle commune à l'ensemble du programme de simulation. Comme les interactions entre processus se limitent à l'échange de messages, il faut assurer que, en recevant des messages, un processus ne puisse détecter qu'un autre processus se trouve à un autre instant du temps virtuel. Pour cela, chaque processus doit prendre connaissance des messages qui lui sont destinés dans l'ordre de leurs dates d'émission. La propriété $\mathcal{P}_3$ assure à chaque processus l'accès aux messages qui lui ont été envoyés (qui appartiennent donc à son passé) et qu'il n'a pas encore consommés (conforme section 3.1.1). Pour assurer cette propriété, les messages destinés a un processus sont placés dans une file ($Fconnus_i$), dont l'accès doit être contrôlé par le noyau. Il doit assurer au processus l'accès à tous les messages qui lui ont été envoyés dans son passé (dont les estampilles sont inférieures à sa

vision de l'horloge globale *hv*) et à aucun des messages appartenant à son futur (dont les estampilles sont supérieures à sa vision de *hv*).

Ces trois problèmes sont à peu près indépendants, il est donc possible de les résoudre séparément. De plus, ils se structurent d'une façon hiérarchique : pour contrôler l'évolution des processus, il est nécessaire d'assurer des communications instantanées dans le temps simulé ; pour mettre en œuvre ce service de communications il est nécessaire de régler les problèmes de distribution et d'acheminement des messages.

Voyons cela sous un autre angle : en résolvant les problèmes liés a la répartition des processus sur les sites, nous construisons une machine virtuelle, toutefois encore asynchrone, sur laquelle les processus peuvent s'échanger des messages sans connaître l'identité ni la localisation de leurs interlocuteurs. En rajoutant des mécanismes de contrôle appropriés pour synchroniser ces transferts de message, nous obtenons une machine virtuelle avec des communications instantanées dans le temps simulé. Finalement, en construisant les primitives présentées dans la section 3.1.2 à partir de ce service de transfert instantané de messages, et en réglant les problèmes de partage de chaque processeur, nous construisons autant de machines virtuelles synchronisées par rapport au temps simulé que de processus du programme de simulation, obtenant ainsi un environnement adéquat à l'exécution du programme de simulation.

Ces constatations nous ont amené à structurer le noyau en trois couches superposées, qui correspondent chacune à un des problèmes énoncés ci-dessus : le *contrôle de la distribution*, qui gère l'acheminement des messages entre les processus et offre aux couches supérieures un service d'échange de messages indépendant de la localisation des processus ; le *contrôle des communications dans le temps virtuel*, qui construit un service de transfert de messages synhronisé par rapport au temps simulé et le *contrôle de l'évolution des processus* (figure 3.3), qui met en œuvre les primitives énoncées dans la section 3.1.2, pour permettre aux processus du programme de simulation d'évoluer de façon cohérente dans le temps simulé.

Cette conception modulaire de la structure du noyau (figure 3.3) facilite l'exécution de modifications et de tests sur les méthodes de synchronisation utilisées, et permet d'offrir aux processus de l'application une interface homogène et indépendante des techniques utilisées dans la mise en œuvre du noyau, ainsi que de la topologie de la machine multiprocesseur utilisée. La fonction qu'accomplit chacune des couches du noyau *Floria* et les services qu'elles offrent seront détaillés dans les sections qui suivent.

3.3 Contrôle de la distribution

Dans une simulation répartie sur une machine multiprocesseur, chaque site de la machine prend en charge l'exécution d'un groupe de processus. L'application ne doit pourtant pas avoir à s'occuper des problèmes liés à cette répartition des processus, en particulier l'acheminement des messages. Le rôle de cette couche du noyau est de résoudre ces problèmes, permettant aux processus du programme de simulation d'être indépendants de leur répartition sur la machine réelle sur laquelle ils s'exécutent.

Lors de la définition du réseau de processus, à travers les primitives `Processus` et `Connexion` (section 3.1.2), cette couche du noyau mémorise le graphe d'interconnexion des processus et leur localisation dans la machine multiprocesseur, pour offrir aux

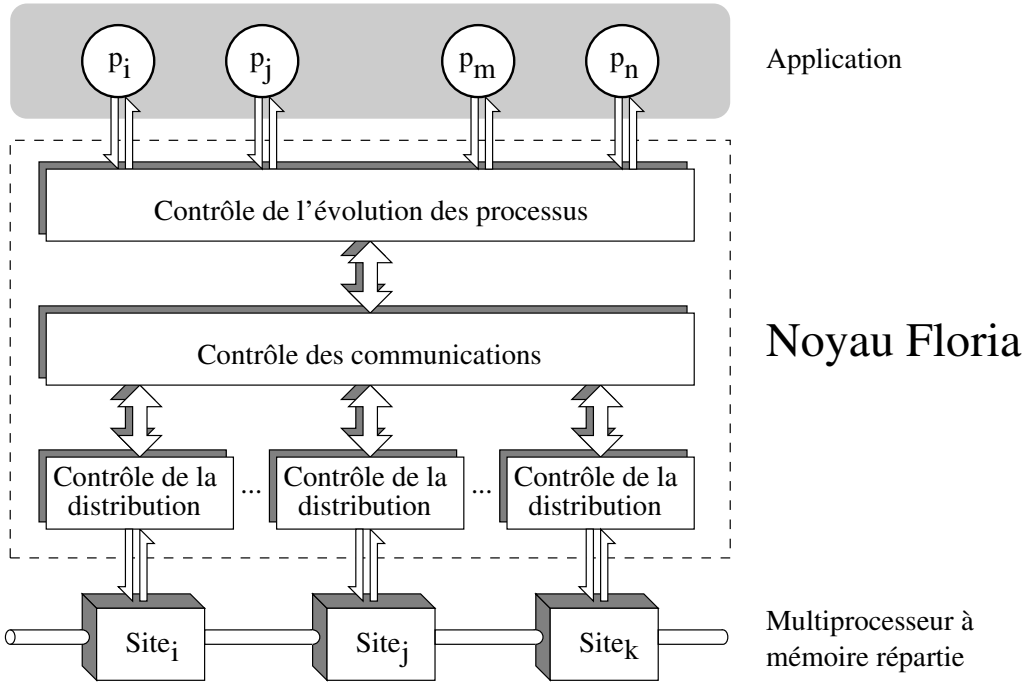

FIG. 3.3 – Structure du noyau *Floria*

couches supérieures du noyau (et par conséquent au programme de simulation) la vision d'un site abstrait unique, sur lequel se trouvent tous les processus de l'application. Le problème d'acheminement des messages est ainsi résolu, car un processus peut émettre des messages sur ses canaux de sortie sans connaître ses destinataires ni savoir sur quels processeurs de la machine ils se trouvent.

Le placement des processus est effectué manuellement et statiquement, lors de leur création, par l'intermédiaire de la primitive `Processus`. En fait, nous avons concentré nos efforts sur les deux couches supérieures (communication et évolution des processus) ; les problèmes liés au placement optimal des processus [AP88, MT91] ou à la migration des processus pour l'équilibre de la charge de calcul [RJ90, Gla92] constituent des domaines de recherche à part entière et n'ont pas été abordés dans nos travaux.

La figure 3.4 montre un exemple du service offert par cette couche lors d'un transfert de message : le processus p_i, sur le site s_i, envoie le message m sur son canal de sortie cs_a. En recevant ce message, la couche de distribution du noyau constate que le canal de sortie sur lequel il a été envoyé (cs_a) est relié au canal d'entrée ce_b du processus p_j, et que celui-ci se trouve sur le site s_j. Le message est alors envoyé au site s_j, où il est reçu par la couche de distribution du noyau et ensuite déposé sur le canal d'entrée ce_b de son destinataire, p_j.

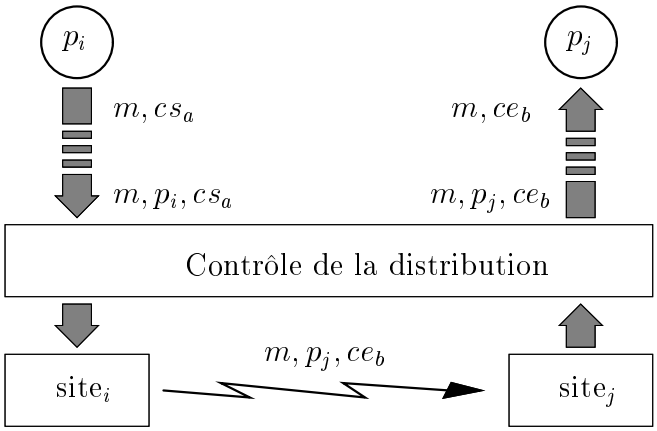

FIG. 3.4 – Le service de distribution de messages

3.4 Contrôle des communications dans le temps virtuel

Cette couche offre un service de transfert de messages dans le temps virtuel, pour faire en sorte que tout message émis par un processus à la date virtuelle t soit délivré à son destinataire à cette même date virtuelle (propriété $\mathcal{P}_2$).

Comme nous avons vu dans la section 2.4, pour garantir que l'évolution du temps simulé sur un processus soit cohérente, et qu'ainsi le principe de causalité soit respecté, les messages arrivés au processus doivent lui être délivrés dans l'ordre de leurs dates d'émission. Ainsi, le processus ne peut alors « remonter le temps virtuel » en recevant un message émis à t et puis un autre émis à t' avec $t' < t$. Pour cela, cette couche met à disposition de la couche supérieure du noyau les messages arrivés à chaque processus, dans l'ordre de leurs dates d'émission.

Les sections suivantes décrivent en détail les services offerts par le contrôle des communications et leur mise en œuvre.

3.4.1 Services offerts

Les processus du programme de simulation sont exécutés par le noyau de façon asynchrone dans le temps réel. Pour que les messages envoyés à un processus soient reçus dans l'ordre correct, ils sont estampillés lors de leur envoi ; quand ils arrivent au destinataire, ils sont stockés dans une file de messages arrivés, en respectant l'ordre de leurs estampilles d'émission. Toutefois, l'ordre des messages dans cette file n'est pas forcément stable. En effet, des nouveaux messages peuvent arriver et s'intercaler entre les messages déjà présents dans la file.

Nous avons vu dans la section 2.5 que, pour un processus p_i donné, le minimum des *temps_canal* tce_{ij} de ses canaux d'entrée ce_{ij} indique l'estampille minimum du prochain message à arriver à ce processus. Cette limite inférieure définit alors une *horloge d'entrée* he_i pour le processus, qui indique la date jusqu'à laquelle le contenu de sa file de messages arrivés est stable : tout nouveau message arrivé aura une estampille

supérieure ou égale à he_i (figure 3.5). Comme l'horloge d'entrée est déterminée à partir des *temps_ canal* des canaux d'entrée du processus ($he_i = min_j(tce_{ij})$), elle peut évoluer à chaque arrivée de message ou à chaque mise à jour des *temps_ canal*.

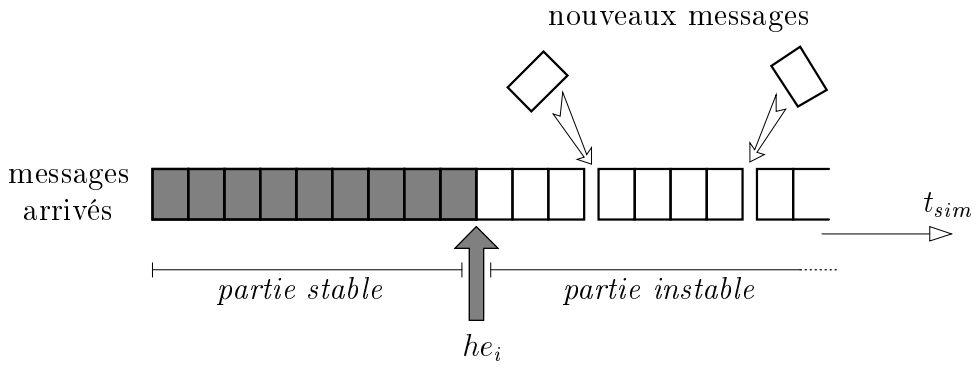

FIG. 3.5 – File de messages arrivés

L'évolution d'un processus dépend de l'arrivée de messages sur ses canaux d'entrée. Ainsi, la couche de contrôle de l'évolution des processus doit être informée lors de ces arrivées, pour assurer l'exécution des processus qui attendent des messages. Toutefois, il n'est utile d'informer la couche supérieure que si le message arrivé peut être délivré au processus, c'est à dire, s'il se trouve dans la partie stable de la file de messages arrivés. Ainsi, le contrôle des communications informe la couche supérieure seulement lors des modifications de la partie stable de cette file, soit par l'avancement de l'horloge d'entrée he_i, soit par l'arrivé d'un message ayant une estampille égale à he_i (dans ce cas, he_i ne change pas de valeur, mais le message vient se mettre à la fin de la partie stable de la file, ce qui la modifie).

Comme nous utilisons une méthode conservative, les processus doivent informer périodiquement le noyau des prévisions sur leur comportement futur. Lorsqu'un processus indique au noyau qu'il n'enverra pas de messages avant la date simulée t, cette information est passé au contrôle des communications pour que les synchronisations nécessaires soient effectuées (émission de messages *null*, dans le cas de la méthode pessimiste de prévention d'interblocages).

Finalement, un service d'envoi de messages instantané dans le temps simulé est offert. Les messages reçus de la couche de contrôle de l'évolution des processus sont aussitôt transférés au contrôle de la distribution, pour leur acheminement.

Pour résumer, la couche de contrôle des communications met à disposition du contrôle de l'évolution des processus les services suivants :

- Envoi de messages instantané dans le temps simulé.
- Stockage des messages arrivés, dans l'ordre de leurs dates d'émission.
- Maintien à jour de l'horloge d'entrée associée à la file de messages arrivés.
- Signalisation des modifications dans la partie stable de la file de messages arrivés.
- Traitement des informations concernant les prévisions des processus.

La figure 3.6 illustre les interactions entre le contrôle des communications et le contrôle de l'évolution des processus.

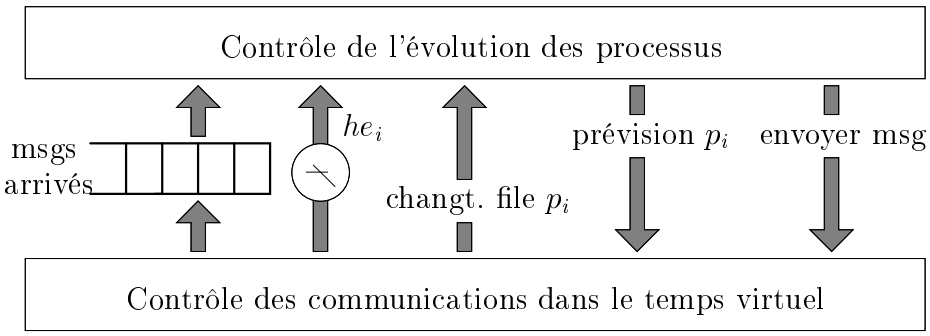

FIG. 3.6 – Services offerts para le contrôle des communications

3.4.2 Mise en œuvre des communications

Pour la mise en œuvre des services de communication et de synchronisation offerts par cette couche, nous avons utilisée la méthode de prévention d'interblocages avec des messages *null*, présentée dans la section 2.5.1. Cependant, contrairement à la proposition initiale de Chandy et Misra [CM79], dans notre implémentation les messages *null* ne sont pas automatiquement envoyés après chaque envoi de message par un processus de l'application.

Tant qu'il s'exécute, un processus peut émettre des messages de la simulation. Un message *null* émis sur un canal peut être aussitôt suivi d'un message de la simulation, ou d'un autre message *null*. Dans ce cas, l'envoi du premier message *null* a été inutile, puisque son unique fonction était de mettre à jour le *temps_canal* du canal sur lequel il a été émis, tâche qui sera aussi bien accomplie par le message qui lui succède.

Pour cette raison, nous avons préféré de ne pas émettre des messages *null* après chaque émission de message par un processus, mais uniquement lorsque l'exécution du processus ne peut plus poursuivre. Ainsi seuls les messages *null* strictement nécessaires seront envoyés, ce qui évite la circulation d'un trop grand nombre de messages de contrôle. L'on pourrait penser que l'envoi différé des messages *null* peut avoir des conséquences néfastes sur les performances des simulations, car les processus bloqués vont recevoir avec du retard les messages qui peuvent les débloquer. Cette constatation n'est pas fausse, mais des tests que nous avons réalisés ont montré que cette perte est largement compensée par l'importante réduction du nombre de messages *null* à produire, transmettre et traiter.

Pour mettre en œuvre la technique des messages *null*, la couche de contrôle des communications prend en charge les tâches suivantes :

- Estampillage des messages envoyés par les processus : lorsqu'un message est envoyé, il reçoit comme estampille la date affichée par l'horloge locale du processus émetteur.
- Synchronisation d'un processus : demandé par la couche de contrôle de l'évolution des processus, lorsque celle-ci interrompt l'exécution du processus. Dans la mise en œuvre actuelle, cette synchronisation consiste simplement en l'envoi des messages *null* nécessaires (en considérant une prévision δ_i sur les canaux de sortie) :

```
Synchroniser (p_i) :
   ∀cs_ij canal de sortie de p_i faire :   /* vérifier tous les canaux de sortie */
      si tcs_ij < hl_i + δ_i alors          /* si le temps_ canal n'est pas à jour */
         envoyer [null, hl_i + δ_i] sur cs_ij ;        /* envoyer message null */
      fsi
   fin
```

- Mise à jour des *temps_ canal* des canaux d'entrée et de sortie : lorsqu'un message est émis ou reçu sur un canal, son *temps_ canal* est mis à jour avec la date contenue dans l'estampille du message.
- Mise à jour de l'horloge d'entrée he_i de chaque processus, au fur et à mésure qu'évoluent les *temps_ canal* des canaux d'entrée.
- réception et filtrage des messages reçus : une fois les mises à jour des *temps_ canal* effectuées, les messages *null* sont effacés et ceux de la simulation sont mis à disposition du contrôle de l'évolution des processus (*i.e.* déposés dans la file de messages arrivés du processus destinataire).

3.5 Le contrôle de l'exécution des processus

La couche de contrôle de l'exécution des processus est la seule accessible directement par le programme de simulation. Elle met en œuvre, à partir des services offerts par les couches inférieures, les primitives présentées dans la section 3.1.2 et construit donc la machine virtuelle offerte à chaque processus de l'application. Pour cela, elle doit accomplir trois fonctions distinctes :

- L'organisation sur un processeur : comme chaque processeur de la machine parallèle peut prendre en charge l'exécution de plusieurs processus du modèle, il est nécessaire de gérer l'utilisation du processeur, pour que son partage soit équitable.
- le contrôle de l'évolution de chaque processus : les processus peuvent évoluer en fonction de leur comportement interne et des messages qui leur arrivent. Il faut prendre en compte les messages arrivés à chaque processus (mises à disposition de cette couche par le contrôle des communications) et de leurs horloges d'entrée pour gérer leur évolution.
- la mise en œuvre, à partir des services offerts par les couches inférieures, des primitives offertes aux processus (conforme la section 3.1.2), notamment celles qui mettent en œuvre la file *Fconnus*, à partir des messages arrivés et des horloges locales et d'entrée de chaque processus.

3.5.1 Organisation sur un processeur

Comme le nombre de processus de l'application est *a priori* plus élevé que celui des processeurs, plusieurs processus doivent être placés sur le même processeur. Il devient alors nécessaire de trouver des schémas pour le contrôle de l'exécution des processus sur un processeur. Une première solution possible consiste à considérer qu'il existe sur chaque processeur un simulateur séquentiel classique qui règle l'évolution des processus placés sur ce processeur ; il n'y a alors qu'une seule horloge virtuelle par processeur et les algorithmes de synchronisation vus dans la section 3.4 ne s'appliquent qu'entre processeurs. Une autre solution consiste à maintenir une horloge virtuelle par processus,

et à appliquer les algorithmes de synchronisation entre tous les processus, qu'ils soient ou non situés sur le même processeur. Cette deuxième solution, *a priori* plus coûteuse, permet en fait de mieux profiter du parallélisme potentiel du modèle. Il est en effet possible que sur un processeur, compte tenu du graphe des processus et des interactions possibles avec des processus placés sur d'autres processeurs, des messages n'ayant pas la plus petite estampille, parmi tous les messages en attente sur le processeur, puissent être traités par leurs processus destinataires.

Le changement de contexte entre processus UNIX standards, effectué a chaque fois que la possession du processeur est transférée d'un processus à l'autre, est une opération coûteuse, vu le nombre de paramètres à prendre en compte et les dimensions des contextes des processus. Comme le nombre de processus dans une simulation répartie peut être élevé et les fréquences d'activations et désactivations de processus assez importantes, nous avons préféré mettre en œuvre ces processus comme des processus légers (*lightweight processes*). A l'époque où nous avons entrepris le développement du noyau *Floria*, cette facilité n'était pas disponible sur la machine cible : aucun micro-noyau mettant en œuvre des processus légers, comme MACH ou CHORUS [Bla90, ZGMB84] ne nous était disponible sur l'iPSC/2. Nous avons donc développé un gestionnaire de processus légers, intégré au noyau. De cette façon, tous les processus de l'application placés sur un même site sont mis à l'intérieur d'un seul processus UNIX et partagent le même espace d'adressage. Le transfert du processeur d'un processus léger à l'autre implique seulement la sauvegarde des registres du processeur (le contexte de chaque processus léger est situé dans sa pile, non accessible aux autres processus).

Les processus se trouvant sur un processeur doivent alterner dans la possession du processeur. Un processus en exécution peut se bloquer, s'il attend des nouveaux messages pour poursuivre, et laisser le processeur à un autre processus capable de s'exécuter. Un processus peut aussi finir volontairement son exécution, ou peut être réactivé par l'arrivé de nouveaux messages sur ses canaux d'entrée. Les états possibles dans le cycle de fonctionnement d'un processus p_i du programme de simulation sont les suivants (figure 3.7) :

- *Bloqué* : il attend l'arrivée de nouveaux messages ou la mise à jour de he_i pour devenir activable ; ces deux événements seront signalés par la couche de contrôle des communications.
- *Activable* : il peut s'exécuter, mais doit attendre que le processeur soit disponible. Les processus activables sont rangés dans une file dont la tête correspond au prochain processus à activer. La couche de contrôle des communications signale au contrôle de l'évolution des processus lorsque l'état des canaux d'entrée d'un processus bloqué est changé, permettant sa réactivation. Un processus ainsi réactivé est aussitôt placé à la fin de la file de processus activables.
- *Actif* : c'est le processus en cours d'exécution, celui qui détient le processeur. Il peut se bloquer en attendant l'arrivée de nouveaux messages, ou terminer son exécution (par l'appel à la primitive `Fin` du noyau, ou lorsque $hl_i > t_{max}$). Si le processus actif ne possède pas de canaux d'entrée, il ne se bloquera jamais dans une attente de message. Alors, pour permettre l'exécution des autres processus activables, il retourne à l'état activable après chaque émission de message.
- *Terminé* : le processus a complètement fini son exécution ; sa prévision de prochain envoi de message δ_i est alors fixée à $+\infty$ pour informer à ses successeurs qu'il

n'enverra plus de messages.

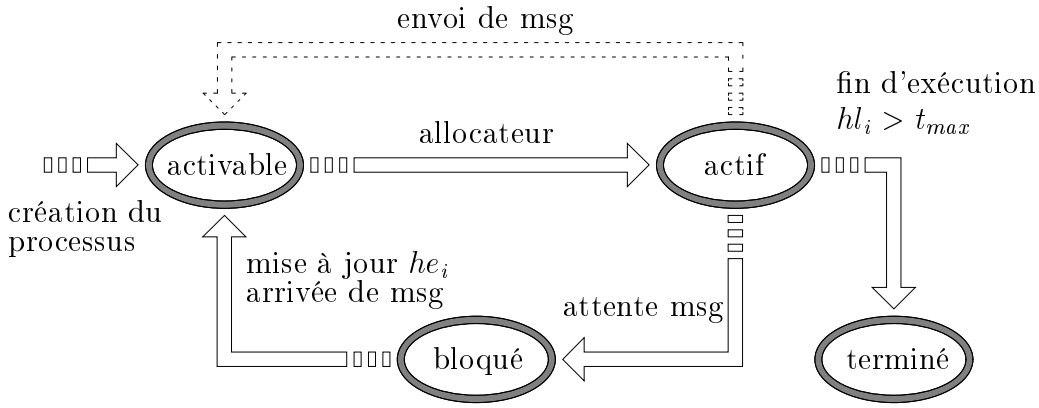

FIG. 3.7 – Cycle d'exécution d'un processus

Sur chaque site de la machine parallèle, plusieurs processus de l'application peuvent s'exécuter ; un allocateur gère l'exécution de ces processus. Le code ci-dessous présente la structure simplifiée de cet allocateur.

```
Allocateur :
   répéter
      proc ← tête (file_ activables) ;          /* prochain processus à activer */
      activer (proc) ;                          /* donner le processeur à proc */
      /* attendre la fin de l'exécution de proc */
      si hl_proc > t_max alors
         proc.état ← terminé ;                  /* fin d'exécution pour le processus */
         δ_proc ← +∞ ;
      fsi
      Synchroniser (proc, hl_proc, δ_proc) ;
   jusqu'à ∀ p_i ∈ Site hl_i > t_max ;          /* jusqu'à la fin de la simulation */
```

L'allocateur se charge simplément d'activer le processus qui se trouve à la tête de la file de processus activables, une fois que le processeur est devenu disponible. Dès que le processus actif se trouve bloqué, il arrête son exécution, et l'allocateur demande à la couche en dessous qu'il soit synchronisé (*i.e.* que des messages *null* soient envoyés sur ses canaux de sortie, dans la mise en œuvre actuelle).

D'après le code ci-dessus, la condition testée par l'allocateur pour vérifier la fin de la simulation porte uniquement sur les processus qui se trouvent sur son site. Il est facile de montrer que le test de cette condition locale est suffisant : lorsque l'horloge locale hl_i d'un processus p_i dépasse la date t_{max} de fin de simulation, il passe à l'état *terminé*, est sa prévision δ_i est fixée à $+\infty$. Ainsi, lors de sa synchronisation, des messages $[null, +\infty]$ seront envoyés sur tous ses canaux de sortie. Cela équivaut à déconnecter le processus terminé du réseau, car chaque message $[null, +\infty]$ reçu sur un canal d'entrée met à $+\infty$ son *temps_ canal*, de sorte que le canal n'est plus considéré pour le calcul de l'horloge d'entrée du récepteur.

La fin de l'exécution d'un processus est une propriété stable, qui ne peut être changée par une arrivée de message. Ainsi, dès que tous les processus sur un site dépassent t_{max},

il ne lui reste rien d'autre à faire que d'attendre que les autres sites finissent l'exécution de leurs processus. La simulation est terminée lorsque tous les sites ont fini d'exécuter leurs processus. Ceci est détecté à l'aide d'un algorithme simple de tour d'un jeton sur un anneau virtuel établi sur l'ensemble des sites [Ray92].

3.5.2 Gestion de l'évolution des processus

L'horloge globale hv, qui donne l'instant actuel dans le temps simulé, est mise en œuvre sous la forme d'une horloge hl_i locale à chaque à chaque processus p_i, qu'il peut consulter et qui avance au fur et à mesure de son exécution. Une synchronisation forte des toutes les horloges locales ($\forall i, j\ hl_i = hl_j$) serait très pénalisante du point de vue de l'efficacité, pour ne pas permettre de bien exploiter le parallélisme potentiel des modèles à simuler. Nous avons donc choisi de faire évoluer les processus de façon asynchrone, en permettant des décalages entre ces horloges ($hl_i \neq hl_j$). Il devient alors nécessaire d'assurer que, lors des communications, les processus ne puissent pas constater de divergences entre leurs horloges locales, de façon à que la propriété $\mathcal{P}_1$ soit toujours garantie. Pour cela, les messages arrivant au processus doivent lui être délivrés dans l'ordre de leurs dates d'émission.

La couche de contrôle des communications fournit, pour chaque processus, une file contenant les messages arrivés, dans l'ordre de leurs estampilles. Elle gère aussi l'horloge d'entrée du processus, qui indique la date minimum d'arrivée de nouveaux messages, c'est à dire, la date jusqu'à laquelle l'ordre des messages dans la file est bien défini. Cette horloge permet de définir quelle portion de la file de messages arrivés peut être mise à disposition du processus sans risque de réception d'un message ayant une estampille plus petite.

Nous pouvons aussi classer les messages arrivés à un processus, par rapport à son horloge locale : une partie de cette file correspond aux messages se trouvant dans le futur du processus (dont les dates d'émission sont supérieures à l'horloge locale du processus : $[m, t_m] \mid t_m > hl_i$), qui lui ont été envoyés par des processus plus avancés dans le temps simulé, et auxquels il n'a pas accès. Le restant des messages appartient à son passé et ira alors constituer la file *Fconnus*, accessible au processus à l'aide des primitives présentées dans la section 3.1.2, dont la mise en œuvre sera détaillée par la suite. La figure 3.8 illustre le classement des messages présents dans la file de messages arrivés d'un processus, d'après son horloge locale et son horloge d'entrée.

Il devient alors clair que l'évolution d'un processus est intimement liée à celle de ses canaux d'entrée. En d'autres termes, l'avancement de l'horloge locale hl_i d'un processus est dépendant de l'avancement de son horloge d'entrée he_i. Pour que le processus ne puisse accéder qu'aux messages dont l'ordre est assuré (autrement dit, pour que l'ordre des messages dans la file *Fconnus* soit assurée), l'horloge locale d'un processus ne doit en principe jamais dépasser son horloge d'entrée. C'est une des tâches du contrôle de l'évolution des processus d'assurer que la relation $\forall i\ hl_i \leq he_i$ soit toujours vérifiée.

Toutefois, il se peut qu'un processus n'ait pas besoin de connaître tous les messages émis vers lui jusqu'à son instant présent pour avoir un comportement correct. Comme exemple, prenons le cas assez courant d'un serveur de réseau de files d'attente avec comportement *fifo* : tant qu'il existe des clients non consommés sur ses canaux d'entrée, il peut les traiter sans prendre connaissance des nouveaux clients arrivés, c'est à dire,

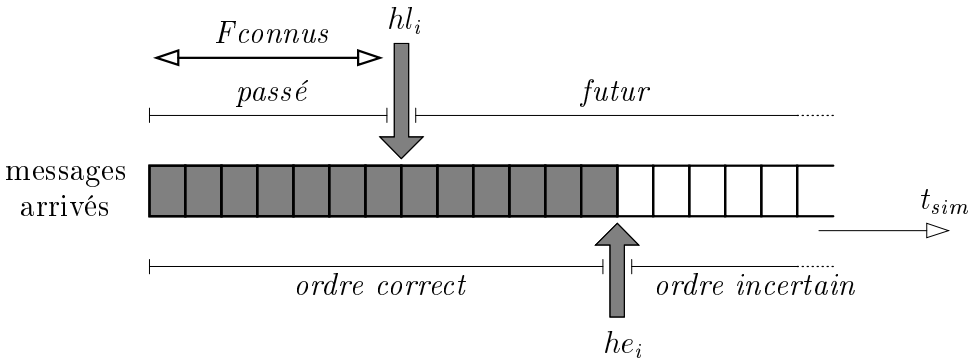

FIG. 3.8 – Classement des messages arrivés

sans connaître l'état complet de ses canaux d'entrée. La seule condition à satisfaire dans ce cas est que l'horloge d'entrée doit être supérieure à l'estampille du prochain client à traiter, pour assurer qu'aucun autre client n'arrivera avec une estampille inférieure à celle du prochain client.

Cet exemple montre que le blocage systématique des processus, en attendant $he_i \geq hl_i$, peut introduire des excès de synchronisation coûteux et inutiles. Pour éviter ce problème, le noyau autorise alors les processus à avoir $hl_i > he_i$, introduisant ainsi des inconsistances dans la file $Fconnus$ (*i.e.* certains messages émis dans le passé de p_i lui sont encore inconnus) lorsque cela ne peut mettre en défaut les propriétés $\mathcal{P}_1$ et $\mathcal{P}_2$; le blocage du processus ne devient nécessaire que s'il tente d'utiliser ces messages.

La section suivante montre comment la mise en œuvre des primitives du noyau prend en compte ces possibilités de relâchement de la synchronisation sur le temps virtuel.

3.5.3 Mise en œuvre des primitives

Les primitives présentées dans la section 3.1.2 permettent aux processus du programme de simulation l'accès à la machine virtuelle construite par le noyau. Elles sont mises en œuvre à partir des services offerts par les couches inférieures. Certaines de ces primitives ont une mise en œuvre trivial, relevant uniquement de détails d'implémentation, et donc ne présentent pas d'intérêt particulier. D'autres, par contre, sont fortement dépendantes des choix effectués dans les sections précédentes, en ce qui concerne les aspects à la syncronisation des processus. C'est le cas des primitives d'attente sur l'horloge (`AttendreTemps`) ou sur message (`AttendreMsg`) et des primitives d'accès à la file de messages connus (`Premier`, `Dernier`, etc.). Dans cette section nous regarderons en détail la mise en œuvre de ces primitives.

Regardons d'abord la primitive d'attente sur horloge : en principe, un processus p_i qui effectue `AttendreTemps(`t`)` devrait d'abord attendre que son horloge d'entrée ait évoluée jusqu'à t ($he_i \geq t$) pour ensuite attribuer à son horloge locale hl_i la valeur t et poursuivre son exécution. Toutefois, comme nous l'avons vu dans la section précédente, il n'est pas toujours nécessaire à un processus d'attendre que le contenu de ses canaux d'entrée soit entièrement connu pour continuer son exécution. Donc, la mise

en œuvre de l'appel `AttendreTemps(`t`)` sera effectuée par la simple attribution de la valeur t à l'horloge locale du processus ($hl_i \leftarrow t$), sans obliger le processus à attendre la resynchronisation entre ses horloges locale et d'entrée.

Rappelons que, par sa définition, la file $Fconnus_i$ contient, à l'instant simulé t, tous les messages émis vers p_i avant ou à la date t et non encore consommés. Avec cette mise en œuvre de `AttendreTemps(`t`)`, cette propriété de $Fconnus_i$ peut être mise en défaut. Toutefois, elle peut toujours être rétablie de façon à ce que cette incohérence temporaire ne soit jamais perceptible par le processus p_i. Cette incohérence temporaire sera éliminée dans la mise en œuvre des primitives qui permettent au processus l'accès aux messages connus, comme le montre l'analyse des deux situations possibles pour les horloges locale et d'entrée d'un processus p_i, après un appel `AttendreTemps(`t`)` :

- $hl_i < he_i$: tous les messages $[m, t_m]$ appartenant au passé du processus ($t_m \leq hl_i$) non encore retirés par `Prendre(`m`)` sont réellement disponibles dans $Fconnus_i$; le processus peut donc les accéder et les consommer s'il le désire. Cette situation corresponde à celle présentée dans la figure 3.8.
- $hl_i \geq he_i$: une partie des messages relatifs au passé de p_i (les messages $[m, t_m]$ dont il est destinataire tels que $he_i < t_m \leq hl_i$) n'est pas réellement disponible, car l'ordre de la file $Fconnus_i$ n'est pas entièrement déterminée (la partie de cette file dont l'ordre est inconnu est représentée par la zone blanche de $Fconnus$ dans la figure 3.9). Seuls les messages $[m, t_m]$ tels que $t_m \leq he_i$ sont réellement disponibles dans $Fconnus_i$.
  Comme nous l'avons vu dans la section 3.5.2, le blocage de $_i$ lorsque $hl_i \geq he_i$ n'est pas toujours nécessaire. p_i peut continuer son exécution mais, lors d'éventuels appels aux primitives d'accès à $Fconnus_i$ (`Premier`, `Dernier`, etc.) il peut être bloqué, s'il tente d'accéder à la partie non encore déterminée de $Fconnus_i$ (partie blanche de cette file, dans la figure 3.9). S'il est bloqué, il reprendra son exécution lorsque la couche de communication dans le temps virtuel lui aura délivré le message attendu : la propriété associée à la définition de $Fconnus_i$ sera alors à nouveau vérifiée.

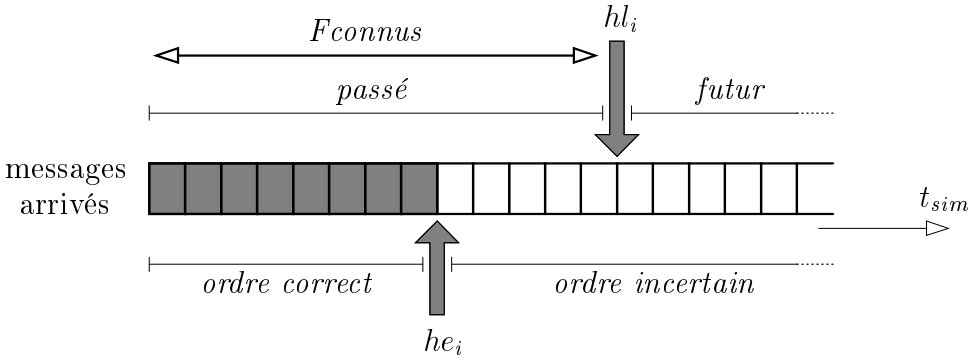

FIG. 3.9 – Attente sur horloge avec $hl_i > he_i$

Donc la mise en œuvre des primitives d'accès à la file de messages connus du processus doit toujours garantir, pour l'accès à un message $[m, t_m]$, que $he_i \geq t_m$. Le cas échéant, le processus est bloqué en attendant que cette condition soit vérifiée pour le

message voulu.

Il nous reste encore à examiner l'implémentation de la primitive d'attente sur message. Un appel à la primitive `AttendreMsg(`t`)` rend au processus un nouveau message, en dehors de ceux qu'il connaissait déjà (qui avaient donc $t_m \leq hl_i$), ou rien, si aucun message n'est arrivée avant la date t d'échéance de l'attente. L'horloge locale du processus doit donc être avancée jusqu'à la date d'arrivée de ce nouveau message, pour que le processus en prenne connaissance, ou à t, le cas échéant.

Ainsi, l'exécution d'une attente sur message par un processus p_i implique toujours l'avancement de son horloge locale hl_i. Elle peut aussi imposer ou non un blocage du processus, suivant qu'il existe ou non un message m disponible sur un canal d'entrée, doté d'une estampille t_m inférieure à la date d'échéance t :

- $\exists[m, t_m] \mid hl_i < t_m \leq he_i$: dans ce cas le plus ancien de ces messages $[m, t_m]$ est intégré à la file $Fconnus_i$, hl_i est avancée à t_m et le processus reprend son exécution.
- $\neg\exists[m, t_m] \mid hl_i < t_m \leq he_i$: le processus est bloqué ; il ne reprendra son exécution que lorsqu'un nouveau message sera délivré par la couche de contrôle des communication, ou quand hl_i sera égal à t.

On en déduit que, immédiatement après une attente sur message, les horloges locale et d'entrée du processus qui l'a exécutée sont toujours resynchronisées, c'est à dire, la condition $he_i \geq hl_i$ est toujours vérifiée.

3.6 Utilisation du noyau

Le noyau *Floria* n'est pas le support d'exécution d'un langage particulier, ni la mise en œuvre d'un type particulier de modèle de simulation. L'objectif est de fournir un ensemble de primitives suffisamment générales pour supporter la réalisation d'environnements correspondant à des langages ou des types de simulation différents. C'est en ce sens que nous parlons de noyau.

Cet objectif est-il atteint ? La réponse à cette question, parce que qualitative, est difficile à donner. Nous avons étudié la mise en œuvre d'un certain nombre de modèles classiques qui laisse penser que les primitives de *Floria* permettent effectivement une mise en œuvre simple et raisonnablement efficace de modèles variés.

Considérons, pour illustrer l'utilisation du noyau *Floria*, la mise en œuvre d'une instruction hypothétique offerte par un langage à l'utilisateur : l'instruction *prendre_ lifo(m)*, qui délivre au processus p_i qui l'invoque le dernier message arrivé. Cette instruction peut être mise en œuvre par la séquence suivante d'appels aux primitives du noyau :

```
prendre_lifo(m) :
   si Vide alors                  /* si Fconnus est vide, attendre l'arrivée d'un message */
      AttendreMsg(∞)
   fsi ;
   m ← Prendre (Dernier) ;                              /* prendre le dernier msg arrivé */
```

Les primitives fournies par le noyau *Floria* permettent de définir des services d'attente de message satisfaisant un prédicat ou appartenant à un type donné, avec ou sans échéance, analogues à ceux proposées dans [BCM87]. La mise en œuvre de ce type

d'extension est simple avec la primitive d'attente de message fournie par *Floria* : il suffit de boucler sur une attente de message et de tester les messages reçus jusqu'à ce qu'un d'entre eux puisse satisfaire le prédicat donné, où la date d'échéance définie soit arrivée.

Dans les sections suivantes nous présentons des exemples plus élaborés de l'utilisation de *Floria* pour la modélisation de serveurs de files d'attente. Nous pouvons mettre en œuvre les serveurs par des processus et les clients par des messages. La file associée à chaque serveur est alors mise en œuvre par la file *Fconnus* du processus. Cette correspondance est certes assez simplifiée, mais permet toutefois de montrer les possibilités d'utilisation des primitives de *Floria* pour la modélisation de réseaux de files d'attente. Les primitives du noyau *Floria* utilisées dans ces modélisations sont présentées dans la partie 3.1.2.

3.6.1 Mise en œuvre de serveurs *fifo* et *lifo*

Dans un réseau de files d'attente, un serveur avec comportement *fifo* (sans préemption) prend chaque client dans l'ordre de son arrivée à sa file d'entrée, traite sa requête pendant un certain temps et l'envoie à la sortie, pour ensuite chercher le prochain client. La durée du traitement de chaque client peut fixe, aléatoire ou dépendante du client à traiter.

Ci-dessous nous présentons la mise en œuvre d'un serveur ayant ce comportement, à l'aide des primitives du noyau *Floria*. Dans cette mise en œuvre, la durée de traitement de chaque client est aléatoire, indépendante du client à traiter (fonction $\mathcal{F}_{aleat}$). Comme le temps de traitement de chaque client est indépendant du client à traiter, il est alors possible de définir la prévision du prochain envoi de message comme étant au minimum le prochain temps de service (si sa file d'entrée est vide, le serveur peut avoir à attendre que des nouveaux clients arrivent, et sa prochaine émission pourra s'effectuer encore plus tard).

```
Serveur fifo :
   répéter
      t_serv ← F_alat ;                          /* temps de service pour le prochain client */
      Prévision (t_serv) ;                       /* prévision du prochain envoi */

      si Vide alors AttendreMsg(∞) fsi ;
      Client ← Prendre (Premier) ;               /* recevoir prochain client à traiter */

      AttendreTemps (t_serv) ;                   /* traiter le client et l'expédier */
      Envoyer (Client, Sortie) ;
   jusqu'à Temps ≥ t_max ;                       /* fin de la simulation */
```

Pour la mise en œuvre d'un serveur *lifo*, qui traite toujours le dernier client disponible, il suffit de remplacer l'appel de la primitive `Premier` par `Dernier`, qui rend le dernier message arrivé.

3.6.2 Mise en œuvre de serveurs *quantum*

Dans un serveur *quantum* les clients sont servis à tour de rôle par tranches de temps de durée fixe (appelées *quantum*), l'ordre des services étant de type *fifo*. Une solution directe mais inefficace pour la mise en œuvre de ce serveur consisterait à réécrire tout le mécanisme de gestion du serveur à l'intérieur d'un processus. La solution adoptée est la suivante : quand le serveur finit le *quantum* d'un client, celui-ci peut être renvoyé à l'entrée du serveur en utilisant un canal auxiliaire (canal *retour* – figure 3.10) ; si le client requiert encore du temps de traitement, il sera alors ordonné avec les nouveaux clients arrivés sur le canal *entrée*. Une fois terminé son traitement, il abandonne le serveur par le canal *sortie*.

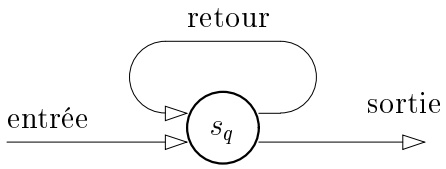

FIG. 3.10 – Serveur *quantum*

Le code décrit ci-dessous représente le comportement du serveur. La variable *sur_ retour*, qui compte le nombre de clients sur le canal *retour*, est utilisée pour le mécanisme de calcul des prévisions à fournir au noyau. Comme le serveur ne se bloque que par l'absence de clients (donc en particulier quand le canal *retour* est vide), il n'est pas utile de calculer une bonne prévision tant qu'il y aura des clients sur ce canal ($sur\_\ retour \neq 0$). Quand le canal *retour* est vide la prévision est donnée par le minimum entre le prochain temps de service (t_{proch}) et la durée du *quantum* (t_{quant}). L'attribut t_{rest} de chaque client indique le temps de traitement qui lui reste à effectuer. La primitive `MsgCanal(`m`)` indique le canal d'où provient le message.

```
Serveur quantum :
   t_proch ← F_alat ;                          /* temps de service pour le premier client */
   sur_retour ← 0 ;

   répéter
      si sur_retour = 0 alors                  /* établir une prévision */
         Prévision (min (t_proch, t_quant))
      sinon Prévision (0.0) fsi ;

      si Vide alors AttendreMsg(∞) fsi ;
      client ← Prendre (Premier) ;             /* prendre prochain client à traiter */

      si MsgCanal (client) = retour alors
         sur_retour ← sur_retour −1
      sinon
         client.t_rest ← t_proch ;             /* temps de service du nouveau client */
         t_proch ← F_alat ;
      fsi ;

      t_serv ← min (client.t_rest, t_quant) ;  /* traiter et expédier le client reçu */
      AttendreTemps (t_serv) ;
      client.t_rest ← client.t_rest - t_serv ;
      si client.t_rest = 0.0 alors
         Envoyer (client, sortie)
      sinon
         Envoyer (client, retour) ;
         sur_retour ← sur_retour +1 ;
      fsi ;
   jusqu'à Temps ≥ t_max ;                     /* fin de la simulation */
fin
```

3.7 Conclusion

La réalisation du noyau *Floria* nous a permis d'abord de préciser les choix architecturaux qui peuvent se poser lors de la mise en œuvre répartie d'applications liées à un temps virtuel, telles que la simulation. Un des caractères principaux de l'architecture de *Floria* est la séparation en couches bien délimitées pour la gestion de la distribution, de la communication dans le temps virtuel et de l'évolution des processus. Sur ce point elle se distingue nettement de la structure d'autres simulateurs, comme PARSEVAL [RM91], un simulateur de réseaux de files d'attente développé sur un réseau de transputers. Cette séparation permet de bien isoler le contrôle des contraintes de causalité, qui a fait l'objet des principales études sur la simulation répartie, et pour lequel des solutions algorithmiques particulières ont été proposées [CM79, Mis86]. Il est ainsi possible de facilement essayer telle ou telle variante de ces algorithmes. Cette séparation permet également d'avoir un produit qui n'est pas lié à une classe d'applications particulière (réseau de file d'attente, etc.).

Pour la mise en œuvre du contrôle des communications, nous avons utilisée la mé-

thode des messages *null* (section 2.5.1). La structure modulaire du noyau *Floria* permet cependant d'utiliser d'autres méthodes conservatives de synchronisation, comme la détection et résolution d'interblocages, ou de proposer des améliorations sur la méthode utilisée, comme des mécanismes d'accélération pour éviter les cycles de messages *null* répétitifs [M88], ou l'utilisation de la mémoire partagée pour rendre plus agile la synchronisation entre les processus implantés sur le même site [GT91, LG93].

L'impossibilité de tirer parti des propriétés du modèle dans l'algorithme de gestion de la communication, à cause de cette séparation en couches, pouvait faire craindre une dégradation des performances du simulateur. La définition de l'interface externe du noyau *Floria*, et sa mise en œuvre dans la couche de contrôle de l'exécution des processus, permettent cependant d'éviter cet écueil. Nous obtenons à la fois une interface générale et souple, et une mise en œuvre qui se révèle efficace quand le comportement des processus le permet (processus *fifo*), comme nous le verrons dans le chapitre suivant.

Chapitre 4

Les performances du noyau *Floria*

Dans le chapitre 3 nous avons décrit la mise en œuvre d'un noyau de système destiné à supporter des simulations réparties. Nous avons présenté des possibilités d'utilisation du noyau, et le pouvoir d'expression des primitives qu'il fournit. Nous allons maintenant mener une évaluation détaillée de son comportement sur un plan quantitatif. Trois questions se posent alors :

- D'abord, et c'est le point essentiel, la technique employée permet-elle de gagner un temps d'exécution appréciable en distribuant les simulations ?
- Quel est l'impact, sur les performances du noyau, des choix de mise en œuvre effectués ?
- Quel est l'impact des caractéristiques du modèle de simulation sur les performances ? Pour quels types de modèle le noyau se comporte-t-il le mieux ?

Peu de mesures ont été effectuées sur l'exécution de méthodes de sysnchronisation pessimistes sur des machines parallèles sans mémoire commune. Il est important d'en connaître le potentiel d'accélération. Il est aussi nécessaire de mesurer l'efficacité d'utilisation de la machine parallèle, c'est à dire, le rapport entre l'accélération obtenue et le nombre de processeurs utilisés.

Le noyau *Floria* ne met pas en œuvre seulement une méthode de synchronisation pessimiste. Il fait usage aussi de techniques de gestion des processus et de contrôle d'accès aux messages reçus qui lui sont propres (section 3.5). Les performances du schéma choisi pour le contrôle des accès aux messages reçus semblent particulièrement sensibles à la politique d'accès des processus à ces messages. Nous avons cherché à mesurer cette influence, en jouant sur cet aspect du comportement des processus.

Dans la section 2.5.1 nous avons vu que les performances des méthodes pessimistes sont liées à la qualité des prévisions que peuvent fournir les processus sur leur comportement futur. Fujimoto présente dans [Fuj88a] des résultats confirmant cette hypothèse pour des machines parallèles à mémoire partagée. Nous avons repris ce type d'expérience dans notre cadre d'étude, en évaluant l'influence de ce paramètre dans des situations diverses.

Dans la suite de ce chapitre nous décrivons d'abord le cadre dans lequel nous avons effectué nos expériences, pour ensuite présenter les résultats obtenus. Le résultat d'une expérience peut a priori dépendre d'un grand nombre de facteurs, et il n'est pas facile dans ce cas d'en tirer des leçons. C'est pourquoi nous n'avons pas choisi des modèles réels mais plutôt des modèles schématiques réguliers, dans le but de pouvoir mieux

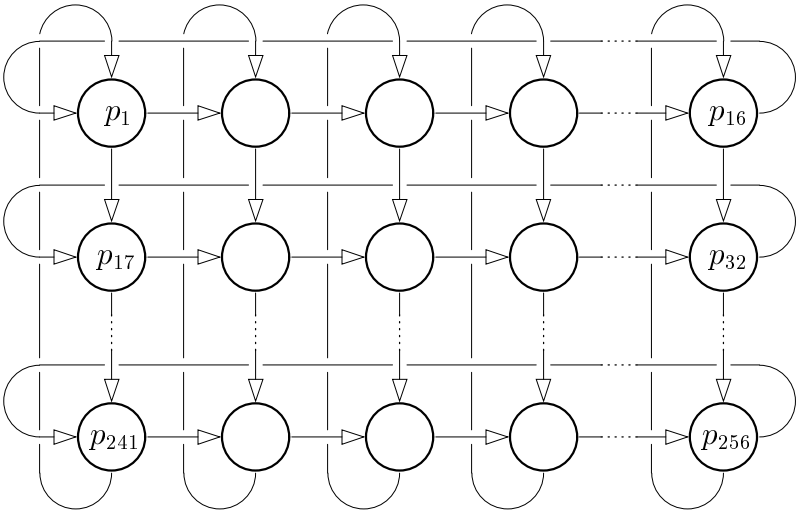

FIG. 4.1 – Topologie du tore

contrôler les facteurs intervenant dans chaque expérience.

4.1 Le cadre de l'étude

La mise en œuvre de *Floria* et les expériences ont été effectuées sur l'iPSC/2 d'INTEL, un multiprocesseur à mémoire distribuée avec un réseau de connexion de type hypercube. Chaque processeur est doté de 4 *Mo* de mémoire locale. Le système d'exploitation sur les nœuds, une version réduite d'UNIX, gère les communications entre les sites de façon à fournir une machine virtuelle complètement interconnectée.

Nous disposons d'une machine ayant 32 processeurs. Il est possible pour un utilisateur de n'utiliser qu'un sous-ensemble de ces processeurs ; les processeurs alloués ne sont pas partagés avec d'autres utilisateurs. Ceci nous a permis de faire varier le nombre de processeurs sur lesquels nous exécutions nos expériences. Le système de communication de l'iPSC/2 assure un contrôle de flux sur les petits messages (moins de 100 octets), ce qui est le cas dans nos expériences. Le nombre de liens de communication séparant deux nœuds a peu d'influence sur le temps de transfert d'un message d'un nœud à l'autre.

Nous avons étudié principalement deux réseaux : d'une part un tore de 16×16 processus (figure 4.1), dans lequel chaque processus à deux canaux d'entrée et deux canaux de sortie, et d'autre part un réseau à topologie variable noté *rev* (figure 4.2), constitué d'une matrice 16×16 de processus, dans laquelle il est possible de faire varier le nombre de canaux d'entrée (et donc de sortie) de chaque processus (dans un réseau à k entrées, noté rev_k, le processus $p_{i,j}$ émet vers les processus $p_{i+1,j+1}$, $p_{i+2,j+1}$, ..., $p_{i+k,j+1}$, tous les indices étant calculés modulo 16).

Ces réseaux comportent un nombre assez important de processus (256). Il nous semble que c'est effectivement le type de modèle à étudier, les simulations intéressantes à distribuer étant forcément de grandes dimensions et présentant un haut dégré de

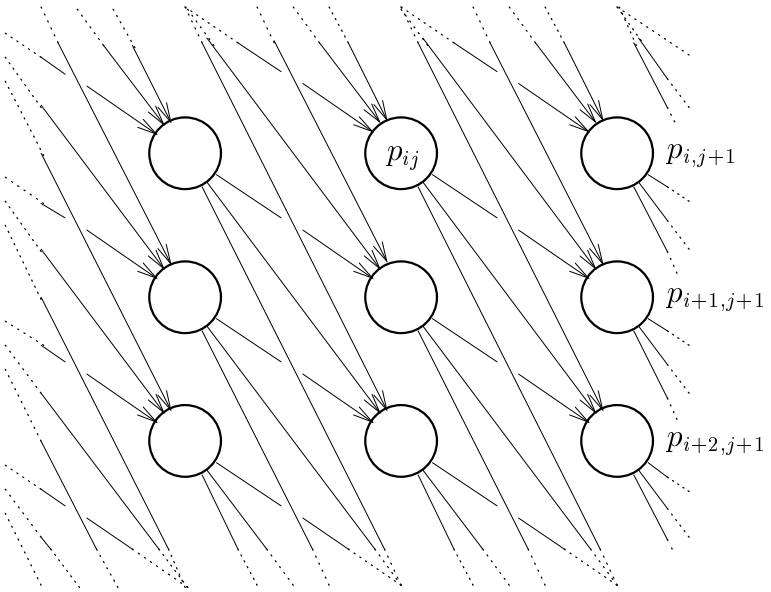

FIG. 4.2 – Topologie du réseau rev_3

parallélisme potentiel (section 2.9).

Nous avons choisi des réseaux réguliers pour simplifier l'analyse des résultats des expériences. En particulier l'utilisation de réseaux symétriques permet de limiter les effets du placement des processus sur les processeurs. Les processus sont placés sur les processeurs par ligne, le nombre de lignes placées sur un site dépendant du nombre de processeurs utilisés. Pour éviter les effets liés au placement une ligne n'est jamais éclatée sur deux processeurs, c'est pourquoi nous n'avons pas fait d'expériences avec plus de 16 processeurs.

Dans une expérience tous les processus du modèle ont un comportement semblable, conforme au schéma ci dessous, où *nbmsg* indique le nombre de messages initiaux, *charge* la charge physique de calcul et t_{max} la date de fin de simulation.

```
Processus du modèle :
   pour i ← 1 jusqu'à nbmsg faire          /* envoyer les nbmsg messages initiaux */
      Msg ← NouveauMsg (type_ msg) ;
      Envoyer (Msg, Sortie) ;
   fait ;
   répéter
      t_serv ← 1.0 + exp (9.0) ;          /* prochain temps de service et prévision */
      Prévision (Q * t_serv) ;
      si Vide alors AttendreMsg(∞) fsi ;   /* prendre le message suivant */
      Msg ← Prendre ( Premier | Dernier | ... ) ;
      AttendreTemps (t_serv) ;            /* attendre t_serv unités de temps virtuel */
      pour i ← 1 jusqu'à charge faire      /* charge physique de calcul */
         charge de calcul unitaire ;
      fait ;
      Envoyer (Msg, Sortie) ;             /* envoyer le message sur une des sorties */
   jusqu'à Temps ≥ t_max ;
   Fin ;
```

Le nombre de messages initiaux, qui fixe le nombre de messages circulant dans le système puisqu'il n'y a pas création de nouveaux messages, est un paramètre des expériences. Les messages sont émis sur les canaux de sortie de manière régulière, un canal après l'autre circulairement.

Nous avons étudié deux politiques pour fixer l'ordre de traitement des messages, *fifo* (le plus ancien message reçu est traité d'abord) et *lifo* (le plus récent message reçu est traité d'abord). Nous nous sommes limités à ces deux types de service parce qu'ils représentent deux situations extrêmes pour notre noyau : *fifo* est la plus favorable (le prélèvement d'un message ne provoque un blocage que s'il n'y a pas de message), *lifo* est la plus défavorable (il faut toujours attendre que que $he_i \geq hl_i$ pour prélever le dernier message reçu, comme vu dans la section 3.5.3).

Le temps de service correspond à la durée dans le temps virtuel du traitement d'un message, il est donné par un tirage d'une variable aléatoire de loi $a + exp\ (b) \mid a, b > 0$[1]. Il y a deux raisons au choix de cette loi : d'abord, le temps de service minimum est différent de zéro, pour éviter des prévisions nulles et donc la possibilité d'interblocage (la section 2.5.1) ; le temps de service maximum n'est pas limité, ce qui permet des décalages importantes entre les horloges des processus, pour en renforcer le comportement asynchrone. Pour chaque processus le germe de la suite aléatoire est différent, et la suite obtenue diffère donc d'un processus à l'autre, ce qui évite des synchronisations trop fortes et peu réalistes entre processus.

Le paramètre *charge de calcul* permet de faire varier le temps physique passé à traiter un message. La durée d'un calcul physique élémentaire est de l'ordre de 0.6 millisecondes ; cette valeur est du même ordre de grandeur que la durée de transfert d'un message court (taille ≤ 100 octets) sur l'iPSC/2.

La prévision mesure la capacité d'un processus à connaître son comportement futur, en ce qui concerne ses émissions de messages. Si au temps virtuel t un processus fixe la valeur de sa prévision sur un canal de sortie à une valeur δ, cela signifie qu'il assure ne pas émettre de message de la simulation sur ce canal avant $t + \delta$. Dans nos expériences

[1] la fonction $exp(x)$ rend une valeur aléatoire de distribution exponentielle négative et moyenne x.

la prévision est toujours la même sur tous les canaux de sortie d'un processus. Plus la valeur proposée est proche de la valeur réelle, plus la prévision est de bonne qualité. Pour pouvoir faire évoluer de manière plus fine ce paramètre nous utilisons comme prévision un pourcentage de la prévision maximum, *i.e.* du prochain temps de service, ce qui définit la qualité $\mathcal{Q}$ de la prévision ($0.0 < \mathcal{Q} \leq 1.0$).

4.2 Stabilité des résultats des expériences

Toute évaluation quantitative doit impérativement assurer la bonne qualité des mesures prélevées. Pour cela, il est nécessaire de répéter un nombre significatif de fois chaque mesure et d'évaluer la confiance dans les résultats obtenus, par des méthodes statistiques. Nous considérons deux sources de variation sur les résultats d'expériences, pour un ensemble de paramètres donné :

- Les variables aléatoires intervenant dans le modèle : ceci a évidement un effet sur le résultat de la simulation (le comportement du modèle), mais aussi sur le comportement du noyau (l'évolution différente du temps virtuel conduit à des situations de synchronisation différentes).
- Les délais variables de communication entre les processeurs de la machine : ils peuvent, bien sûr, avoir un effet sur le comportement du noyau, mais ils peuvent également influer sur le comportement de certains modèles. En effet, comme *Floria* permet de délivrer un message émis au temps simulé t sans que tous les messages émis à cette date soient reçus, des délais de communications différents peuvent conduire à des ordres d'arrivée des messages différents et donc à des comportements du modèle différents.

Nous avons d'abord étudié la stabilité des résultats de certaines expériences quand tous les paramètres, y compris les germes des séries pseudo-aléatoires, sont identiques. Dans ce cas, les seules variations sont dues aux délais de communication de l'iPSC/2. Les modèles avec des processus *fifo* donnent des résultats moins stables que ceux des modèles avec des processus *lifo* : en effet pour des processus *lifo* les temps de communication n'ont aucun effet sur le comportement du modèle puisque un message émis à la date simulée t ne peut être délivré que si tous les messages émis à t ont été reçus. Dans tous les cas l'effet moyen sur le comportement du modèle est très faible : l'intervalle de confiance à 95% est inférieur à $\pm 3\%$ de la valeur moyenne, comme l'indique le tableau 4.1 qui donne ce pourcentage pour le temps de simulation, le nombre de messages de la simulation et le nombre de messages *null*. Toutes ces mesures ont été effectuées sur des simulations réparties sur 4 processeurs.

Nous avons également étudié les résultats des expériences faites avec les mêmes paramètres, mais avec des séries pseudo-aléatoires différentes. Là encore nous avons une bonne stabilité des résultats des expériences : l'intervalle de confiance à 95% est toujours inférieur à $\pm 3\%$ de la valeur moyenne, comme l'indique le tableau 4.2 pour quelques expériences.

réseau	type	charge calcul	nombre msgs	prév.	confiance temps	confiance msgs simul	confiance msgs *null*
tore	*fifo*	0	1	min	1,61%	0,27%	1,95%
tore	*lifo*	0	1	min	0,71%	0,00%	0,66%
rev_4	*fifo*	0	1	min	2,49%	0,27%	2,60%
rev_4	*fifo*	0	4	min	1,35%	0,13%	1,39%
rev_4	*lifo*	0	1	min	0,81%	0,00%	0,74%

TAB. 4.1 – Largeur de l'intervalle de confiance en % de la moyenne

réseau	type	charge calcul	nombre msgs	prév.	confiance temps	confiance msgs simul	confiance msgs *null*
tore	*fifo*	0	1	min	1,85%	0,43%	2,50%
tore	*lifo*	0	1	min	1,88%	0,27%	1,53%
rev_4	*fifo*	0	1	min	2,21%	0,37%	2,61%
rev_4	*fifo*	0	4	min	1,72%	0,29%	2,41%
rev_4	*fifo*	4	1	min	2,75%	0,32%	2,69%
rev_4	*lifo*	0	4	min	1,53%	0,23%	1,67%
rev_4	*lifo*	0	1	min	1,80%	0,27%	1,86%

TAB. 4.2 – Largeur de l'intervalle de confiance en % de la moyenne

4.3 Comparaison avec un simulateur séquentiel

Pour évaluer le gain apporté par une simulation distribuée sur n processeurs il faut en principe la comparer avec la simulation du même modèle effectuée sur un simulateur séquentiel. Nous ne disposons pas d'un simulateur séquentiel pouvant s'exécuter sur un nœud de l'iPSC/2.

Comme d'une part nous disposons d'un logiciel de traitement de réseaux de file d'attente QNAP2 [PV84], incluant un simulateur séquentiel et s'exécutant sur une station de travail, et que d'autre part nous pouvons exécuter *Floria* sur cette station, nous avons voulu calibrer nos expériences en comparant l'exécution de la simulation du même modèle sur une station de travail, en utilisant soit QNAP2 soit *Floria*.

Le tableau 4.3 regroupe les résultats de quelques unes de ces expériences sur un réseau *rev* de 256 processus (16×16) : la colonne $\#msg$ donne le nombre de messages par processus, $\#can$ donne le nombre de canaux d'entrée par processus, *prév* indique quelle prévision a été utilisée. Les temps de simulation (t_{Qnap2} et t_{Floria}) sont donnés en secondes, et la différence entre eux est donnée par $\%Diff = (t_{Floria} - t_{Qnap2})/t_{Qnap2}$. Nous avons constaté que sur les réseaux étudiés l'exécution de *Floria* sur un monoprocesseur est la plupart du temps plus rapide que l'exécution de QNAP2 ($\%Diff$ négatif). Les expériences 2 et 14 correspondent à des situations très pénalisantes pour la méthode de synchronisation mise en œuvre dans *Floria*, comme nous le verrons par la suite (beaucoup de canaux d'entrée, peu de messages et une prévision de qualité minimale).

Un certain nombre de facteurs peuvent expliquer cette bonne performance de *Floria* : d'une part QNAP2 peut faire bien d'autres choses que la simulation séquentielle (résolution analytique par diverses méthodes) et le simulateur n'est pas forcément très

expér.	type	#msg	#can	prév.	t_{Qnap2}	t_{Floria}	%*Diff*
1	*fifo*	1	2	min	108.9	99.0	-9.1
2	*fifo*	1	4	min	108.3	687.0	534.3
3	*fifo*	2	2	min	164.0	59.0	-64.0
4	*fifo*	2	4	min	165.8	185.0	11.6
5	*fifo*	4	2	min	214.3	45.0	-79.0
6	*fifo*	4	4	min	214.5	87.0	-59.4
7	*fifo*	1	2	max	108.9	49.0	-55.0
8	*fifo*	1	4	max	108.3	72.0	-33.5
9	*fifo*	2	2	max	164.0	42.0	-74.4
10	*fifo*	2	4	max	165.8	69.0	-58.4
11	*fifo*	4	2	max	214.3	42.0	-80.4
12	*fifo*	4	4	max	214.5	57.0	-73.4
13	*lifo*	1	2	min	109.1	150.0	37.5
14	*lifo*	1	4	min	108.3	635.0	486.3
15	*lifo*	2	2	min	163.8	109.0	-33.5
16	*lifo*	2	4	min	164.2	219.0	33.4
17	*lifo*	4	2	min	212.7	97.0	-54.4
18	*lifo*	4	4	min	212.7	173.0	-18.7
19	*lifo*	1	2	max	109.1	52.0	-52.3
20	*lifo*	1	4	max	108.3	75.0	-30.7
21	*lifo*	2	2	max	163.8	51.0	-68.9
22	*lifo*	2	4	max	164.2	71.0	-56.8
23	*lifo*	4	2	max	212.7	53.0	-75.1
24	*lifo*	4	4	max	212.7	82.0	-61.4

TAB. 4.3 – Comparaison entre QNAP2 et *Floria* / modèle à 16 × 16 processus

optimisé ; d'autre part les dimensions du modèle utilisé sont assez importantes, ce qui est pénalisant pour un simulateur séquentiel. Effectivement la simulation d'un modèle plus petit (16 processus) est plus favorable à QNAP2, comme le montrent les résultats sur le tableau 4.4.

Il est alors possible de conclure que *Floria* est efficace dans l'exécution des modèles lourds à traiter par des simulateurs séquentiels. Cela peut être attribué à la difficulté de gestion de l'échéancier dans un simulateur classique. Dans *Floria*, l'échéancier global cède sa place à des échéanciers locaux à chaque processus, de taille beaucoup plus réduite et donc plus faciles à gérer.

Compte tenu de ces résultats, pour les expériences ultérieures nous avons évalué les accélérations en rapportant une simulation avec *Floria* sur n processeurs de l'iPSC/2 à la simulation du même modèle avec *Floria* sur un seul processeur de l'iPSC/2.

4.4 Les expériences réalisées

Le tableau 4.5 donne les caractéristiques détaillées de chacune des expériences réalisées, *i.e.* les valeurs choisies pour les paramètres considérés variables lors de chaque expérience. Nous y ferrons référence dans les sections ultérieures.

expér.	type	#msg	#can	prév.	t_{Qnap2}	t_{Floria}	%*Diff*
25	*fifo*	1	2	min	4.67	19.0	306.9
26	*fifo*	1	4	min	4.59	42.0	815.0
27	*fifo*	2	2	min	5.86	8.0	36.5
28	*fifo*	2	4	min	5.98	26.0	334.8
29	*fifo*	4	2	min	6.64	5.0	-24.7
30	*fifo*	4	4	min	6.84	13.0	90.1
31	*fifo*	1	2	max	4.67	6.0	28.5
32	*fifo*	1	4	max	4.59	11.0	139.7
33	*fifo*	2	2	max	5.86	5.0	-14.7
34	*fifo*	2	4	max	5.98	8.0	33.8
35	*fifo*	4	2	max	6.64	4.0	-39.8
36	*fifo*	4	4	max	6.84	5.0	-26.9
37	*lifo*	1	2	min	4.39	26.0	492.3
38	*lifo*	1	4	min	4.39	44.0	902.3
39	*lifo*	2	2	min	5.74	16.0	178.7
40	*lifo*	2	4	min	5.93	34.0	473.4
41	*lifo*	4	2	min	6.57	13.0	97.9
42	*lifo*	4	4	min	6.73	26.0	286.3
43	*lifo*	1	2	max	4.39	9.0	105.0
44	*lifo*	1	4	max	4.39	12.0	173.3
45	*lifo*	2	2	max	5.74	8.0	39.4
46	*lifo*	2	4	max	5.93	11.0	85.5
47	*lifo*	4	2	max	6.57	7.0	6.5
48	*lifo*	4	4	max	6.73	10.0	48.6

TAB. 4.4 – Comparaison entre QNAP2 et *Floria* / modèle à 4×4 processus

4.5 Accélération obtenue par la distribution

L'accélération a été calculée comme étant, pour un modèle donné, le rapport entre le temps d'exécution de *Floria* sur 1 processeur et le temps d'exécution sur n processeurs. La figure 4.3 donne les résultats obtenus sur le tore 16×16, avec des serveurs *fifo* et un indice de charge de calcul physique variant de 0 à 16 (expérience 1). Dans tous les cas une accélération de l'exécution est obtenue quand le nombre de processeurs augmente, cette accélération étant plus forte quand la charge de calcul physique est plus élevée.

L'efficacité d'utilisation de la machine, obtenue en divisant l'accélération par le nombre de processeurs utilisés, diminue avec le nombre de processeurs, comme le montre la figure 4.4. Évidemment, plus il existe des processeurs pour effectuer un calcul coopératif, moins efficacement chacun sera utilisé, à cause des coûts de communication et de synchronisation. Mais en fait pour la simulation distribuée, destinée à rendre réalisables des simulations de grandes dimensions, c'est bien la possibilité de diminuer le temps global de simulation qui est importante, même au prix d'une mauvaise utilisation des processeurs.

Cette possibilité d'accélération se retrouve pour d'autres modèles, comme le montrent les figures 4.5 (avec les mêmes conditions que précédemment, mais des serveurs *lifo* - expérience 2) et 4.6 (*rev* à 2, 3, 4 ou 8 canaux d'entrée - expérience 3).

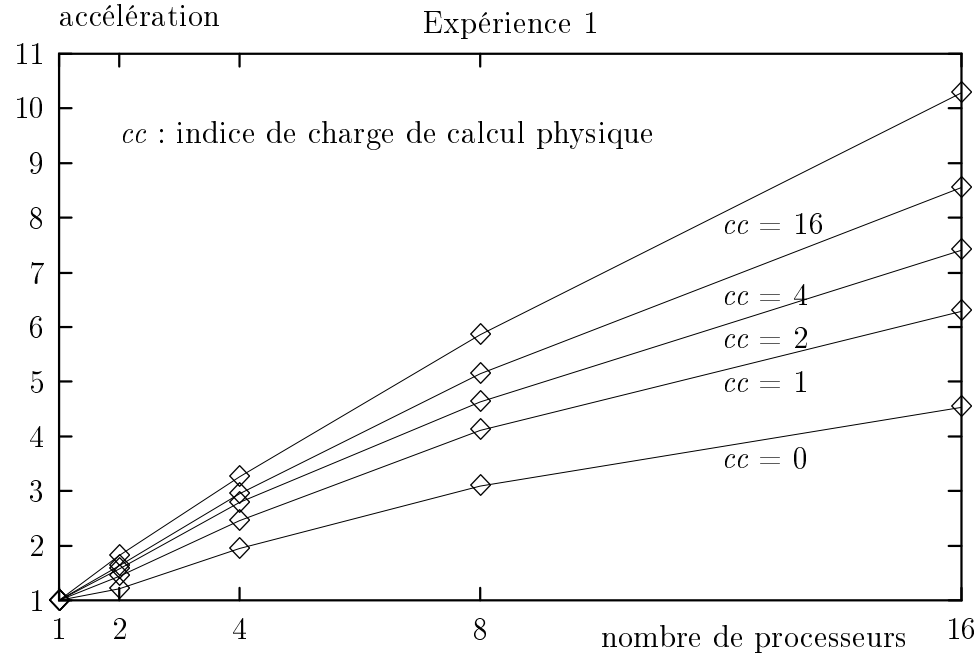

FIG. 4.3 – Accélération = $\mathcal{F}$ (nombre de sites) pour le tore, processus *fifo*

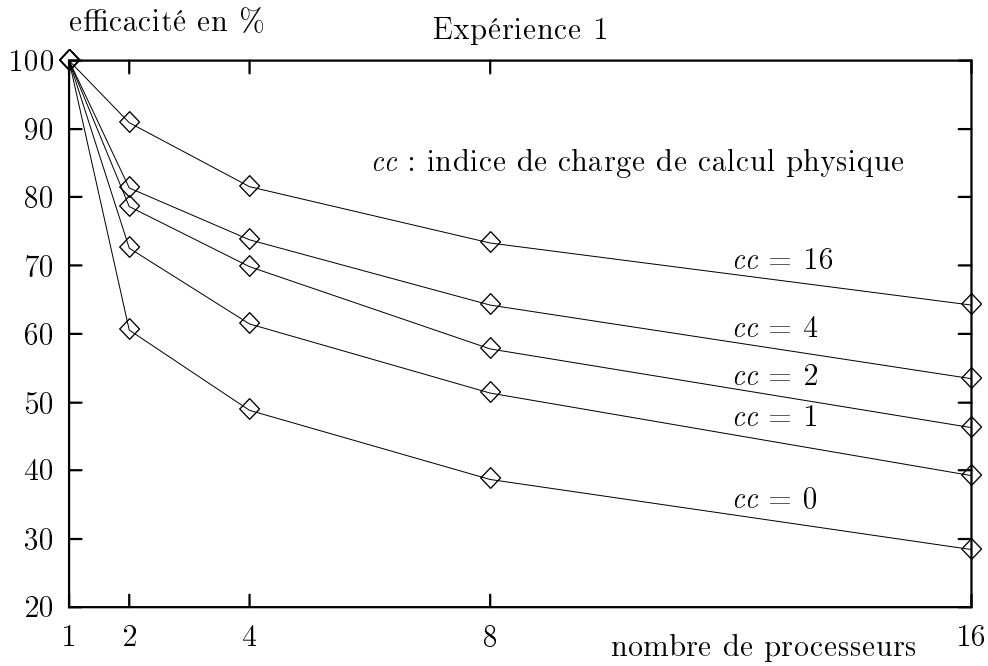

FIG. 4.4 – Efficacité = $\mathcal{F}$ (nombre de sites) pour le tore, processus *fifo*

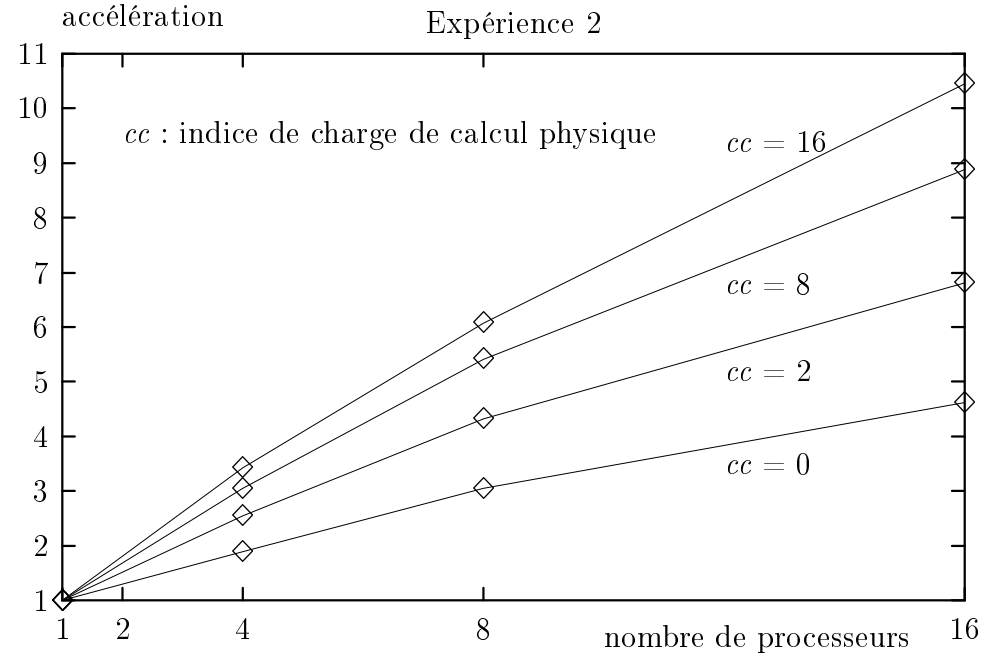

FIG. 4.5 – Accélération = $\mathcal{F}$ (nombre de sites) pour le tore, processus *lifo*

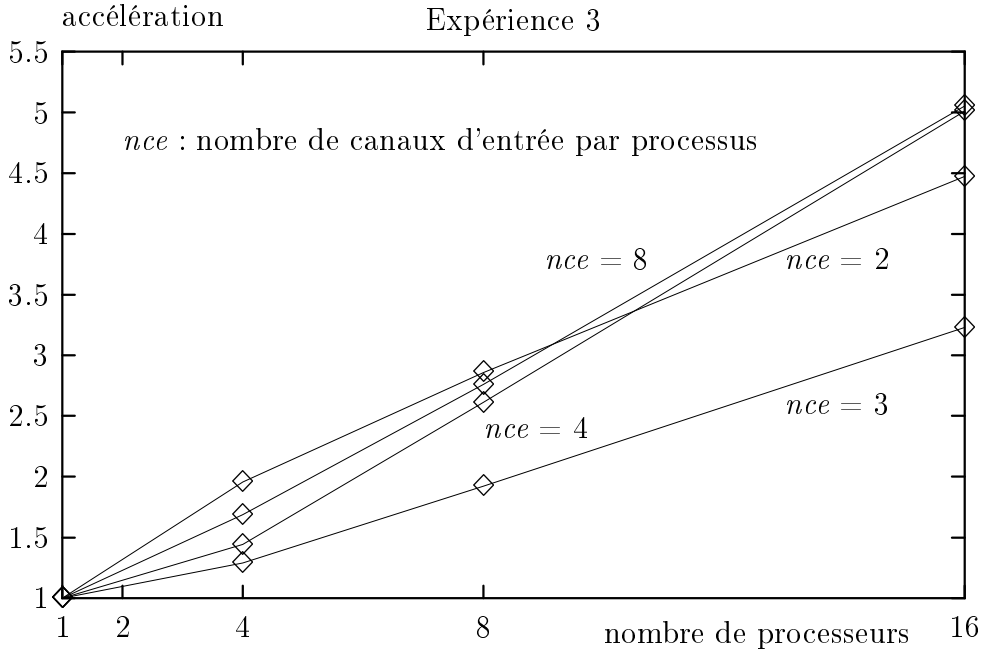

FIG. 4.6 – Accélération = $\mathcal{F}$ (nombre de sites) pour le *rev*, processus *lifo*

expér.	réseau	nombre canaux	nombre nœuds	type processus	charge calcul	nombre msgs	prév.
1	tore	2	1 à 16	*fifo*	0 à 16	1	min
2	tore	2	1 à 16	*lifo*	0 à 16	1	min
3	*rev*	2-3-4-8	1 à 16	*fifo*	0	1	min
4	tore	2	4-8-16	*fifo/lifo*	0 à 64	1	min
5	*rev*	4	4-8-16	*fifo*	0 à 32	1	min
6	tore	2	8	*fifo/lifo*	0	1	min
7	tore	2	8	*fifo/lifo*	0	1-2-3-4	min
8	tore	2	8	*fifo/lifo*	0	1-2	min à max
9	*rev*	2-3-4-5	8	*fifo/lifo*	0	1-2-4	min
10	tore	2	8	*fifo/lifo*	0	1-2	min à max
11	*rev*	3-5	8	*fifo*	0	1-4	min à max
12	*rev*	3-5	8	*lifo*	0	1-4	min à max
13	tore	2	8	*fifo/lifo*	0	1-2-3-4	min et max
14	*rev*	2-3-4-5	8	*fifo*	0	1-2-3-4	min
15	*rev*	2-3-4-5	8	*lifo*	0	1-2-3-4	min
16	*rev*	1 à 15	16	*fifo/lifo*	0	1	min et max

TAB. 4.5 – Caractéristiques des expériences

4.6 Influence de la charge de calcul

Nous allons étudier ici plus précisément l'effet de la charge de calcul physique sur le comportement d'une simulation. Intuitivement, plus les processus du modèle ont de calcul à faire, plus la distribution de la simulation doit être intéressante. Nous retrouvons effectivement ce résultat, par exemple sur la figure 4.7 qui donne l'évolution de l'efficacité en fonction de la charge de calcul physique, pour un tore 16×16 (expérience 4).

Si l'augmentation de l'efficacité est très rapide au début, elle se stabilise au delà d'une certaine charge. Dans l'expérience de la figure 4.7 l'efficacité optimale est pratiquement atteinte avec un indice de charge de 16 (ce qui correspond à un temps physique de traitement d'un message de l'ordre de 9 *ms*). La figure 4.8 donne pour la même expérience, dans le cas des serveurs *fifo*, le pourcentage de temps passé dans le noyau par rapport au temps total de simulation. Le pourcentage de temps passé dans le système diminue rapidement quand la charge physique commence à augmenter, pour se stabiliser ensuite. Ce phénomène n'est pas dû à une évolution du nombre de messages *null*, qui n'est pas sensible à l'augmentation de la charge, comme l'indique la figure 4.9, donnant le pourcentage de messages *null* par rapport au nombre total de messages échangés. La diminution du pourcentage de temps passé dans le système quand la charge commence à augmenter est probablement dûe à la récupération partielle des temps d'attente pour effectuer le travail supplémentaire dû au modèle. Mais cette récupération ne peut être complète, et il arrive un moment ou l'augmentation de la charge n'améliore plus les performances.

Un comportement similaire est retrouvé sur le réseau à nombre variable d'entrées, comme indiqué sur la figure 4.10, qui donne l'efficacité en fonction de la charge pour

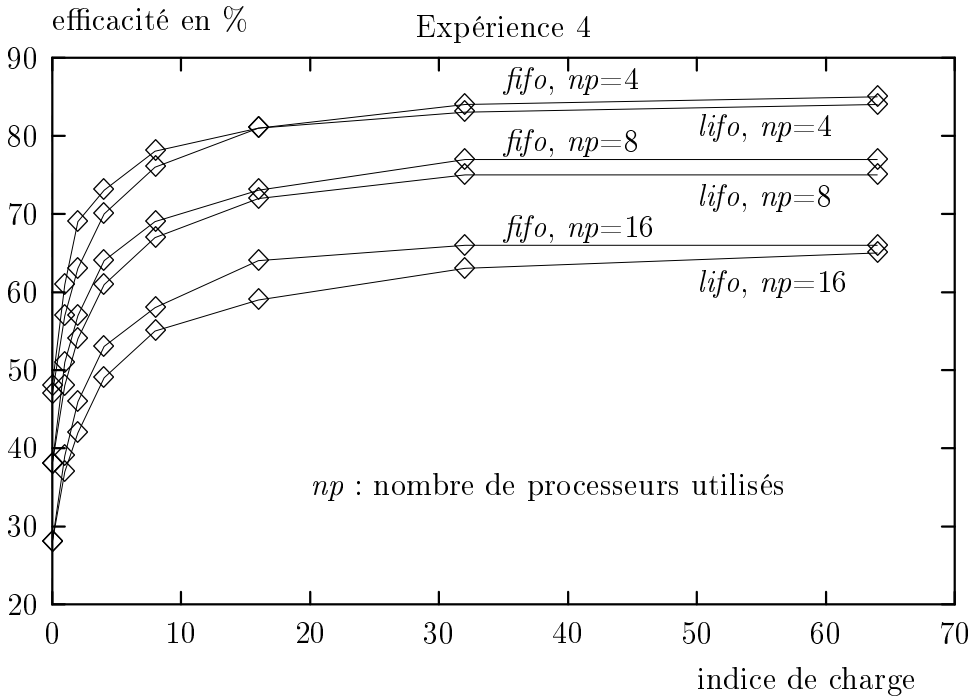

FIG. 4.7 – Efficacité = $\mathcal{F}$ (charge de calcul) pour le tore

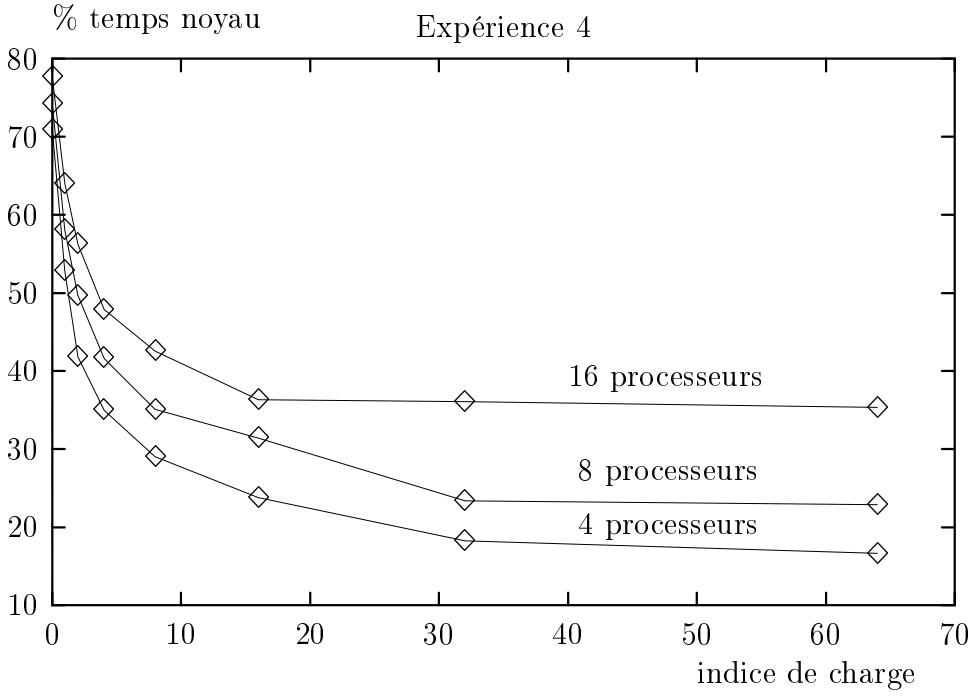

FIG. 4.8 – % temps noyau = $\mathcal{F}$ (charge de calcul) pour le tore, processus *fifo*

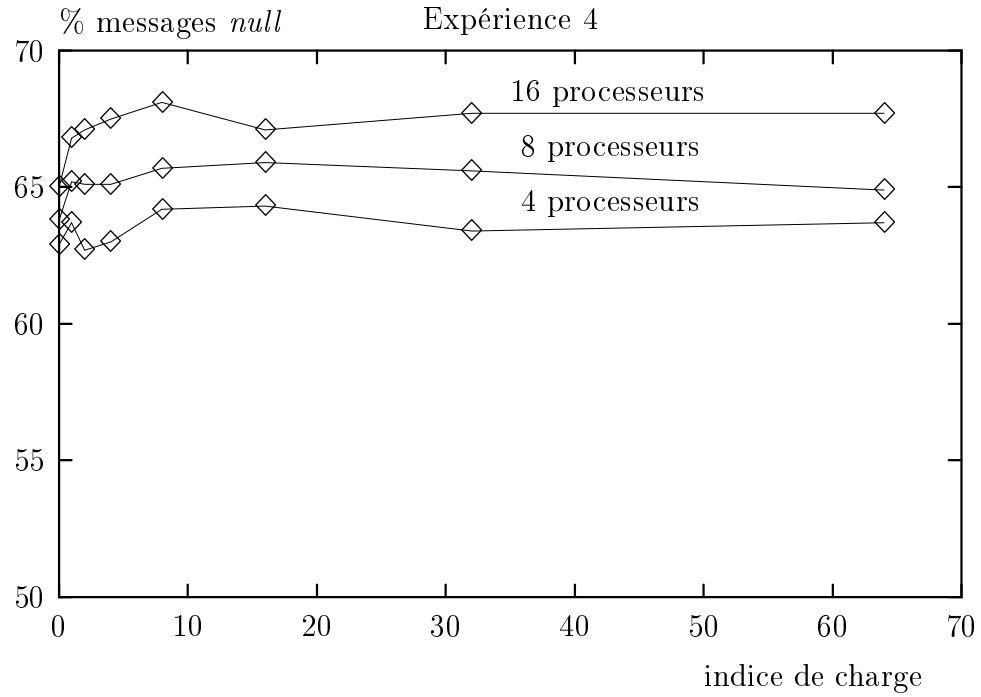

FIG. 4.9 – % messages $null = \mathcal{F}$ (charge de calcul) pour le tore, processus *fifo*

un réseau à 4 canaux d'entrée par processus *fifo* (expérience 5).

4.7 Influence de la politique de service

Nous étudions ici l'influence du comportement des processus du modèle simulé sur les performances du noyau. Nous avons étudié deux types de comportement : des serveurs *fifo* et des serveurs *lifo*. Dans chaque expérience tous les processus sont du même type. Nous évaluons, tous paramètres étant égaux par ailleurs, la différence entre les durées d'exécution pour des serveurs *fifo* et *lifo*, grâce à l'indicateur $\delta_{lf} = (t_{lifo} - t_{fifo})/t_{fifo}$.

La courbe 4.11 donne la valeur de δ_{lf} en fonction de l'indicateur de charge physique, pour un tore sur 8 processeurs (expérience 6). La différence, importante pour les charges faibles, devient négligeable quand la charge augmente. Cela confirme les résultats obtenus sur l'influence de la charge physique : les temps d'attente plus importants pour les serveurs *lifo* expliquent l'écart important quand la charge est faible, cet écart diminue quand le travail supplémentaire du modèle est pris sur ces temps d'attente.

La courbe 4.12 donne la valeur de δ_{lf} en fonction du nombre de messages par processus, pour un tore sur 8 processeurs (expérience 7). Elle montre que l'écart entre les deux comportements extrêmes est plus important quand le nombre de messages augmente. Cela indique clairement que la désynchronisation introduite entre les horloges locale hl_i et d'entrée he_i de chaque processus (comme vu dans la section 3.5) a été bénéfique, permettant au noyau de mieux tirer profit du comportement des processus : le fait de ne bloquer un processus que s'il tente effectivement d'utiliser un message manquant conduit à diminuer considérablement le nombre de blocages pour des serveurs *fifo* quand le nombre de messages augmente.

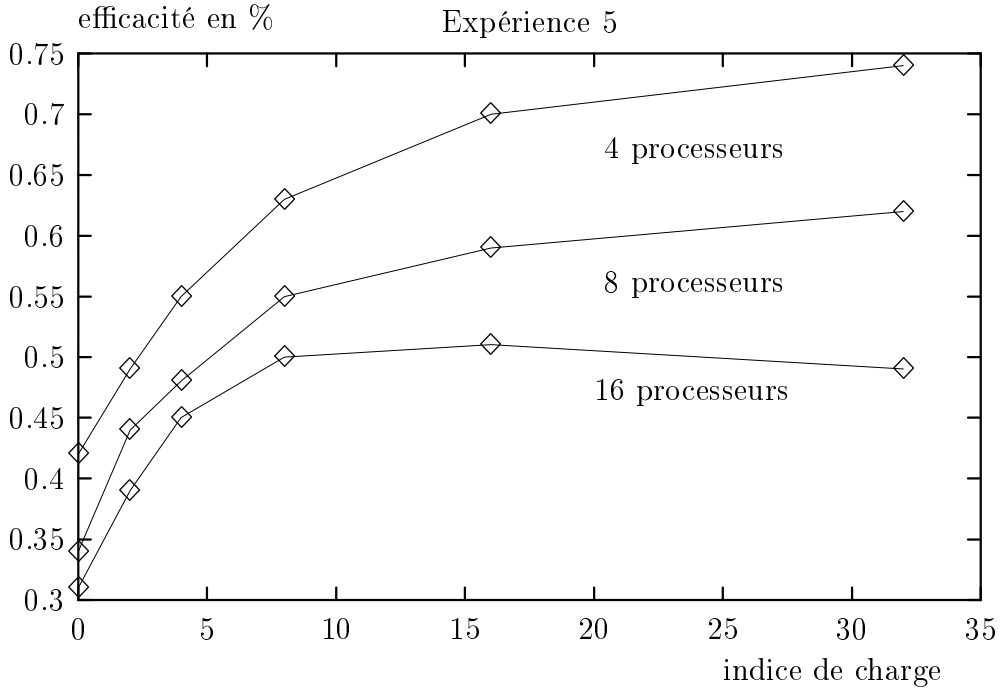

FIG. 4.10 – Efficacité = $\mathcal{F}$ (charge de calcul) pour le *rev* à 4 canaux d'entrée

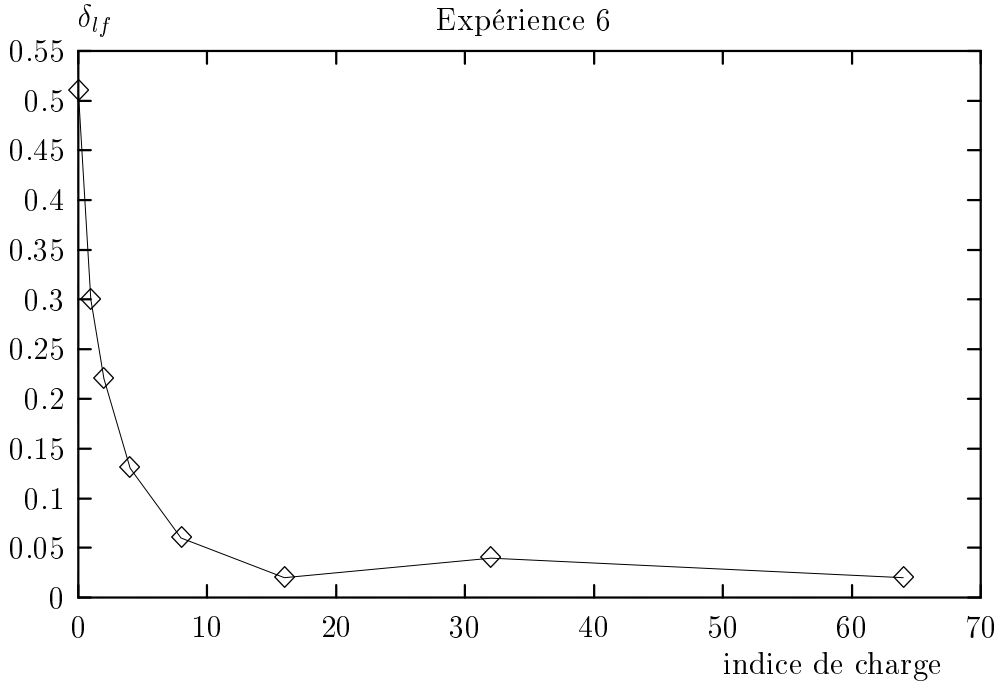

FIG. 4.11 – δ_{lf} = $\mathcal{F}$ (charge de calcul) pour le tore

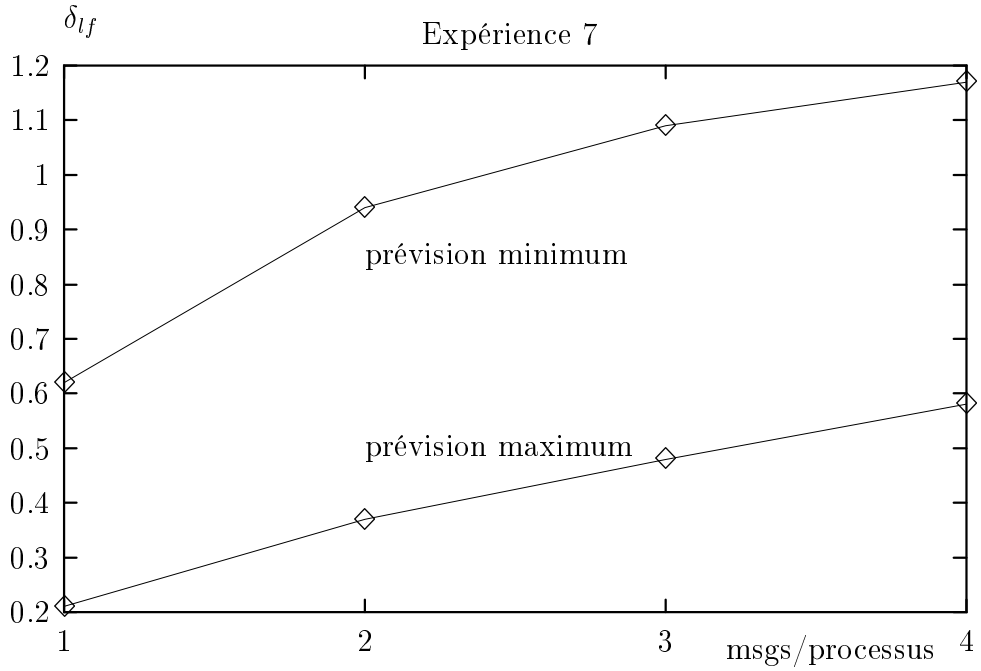

FIG. 4.12 – $\delta_{lf} = \mathcal{F}$ (nombre de messages) pour le tore

La courbe 4.13 donne la valeur de δ_{lf} en fonction de la qualité $\mathcal{Q}$ de la prévision, pour un tore sur 8 processeurs (expérience 8). Une bonne prévision diminue l'écart entre le comportement des processus *fifo* et *lifo*.

La courbe 4.14 donne la valeur de δ_{lf} en fonction du nombre de canaux d'entrée pour le *rev* sur 8 processeurs (expérience 9). Nous y retrouvons l'augmentation de la différence entre les serveurs *fifo* et *lifo* quand le nombre de messages en circulation augmente. Le comportement vis à vis du nombre de canaux d'entrée des processus dépend du nombre de messages : quand il y a un message par processus l'augmentation du nombre de canaux d'entrée est très pénalisante, quel que soit le type de serveur, et la différence entre les deux types de serveurs diminue ; par contre quand le nombre de messages augmente le phénomène évoqué précédemment est retrouvé et les serveurs *fifo* sont avantagés.

4.8 Influence de la prévision

Nous nous intéressons ici plus particulièrement aux effets de la prévision sur le comportement des simulations. Une étude faite par Fujimoto [Fuj88a] sur les méthodes de simulation distribuées conservatives indiquait que la qualité de la prévision est un paramètre influençant fortement l'efficacité de la simulation. Cette étude avait été effectuée sur un multiprocesseur à mémoire partagée (de type BBN BUTTERFLY) et il est intéressant de regarder si ses résultats restent valables avec une mémoire distribuée. Dans son étude, Fujimoto prenait toujours comme prévision le temps de service minimum, et il évaluait la qualité de la prévision par le rapport *prévision / (temps moyen de service)*. Pour faire varier cette qualité il utilisait plusieurs lois différentes, de même moyenne.

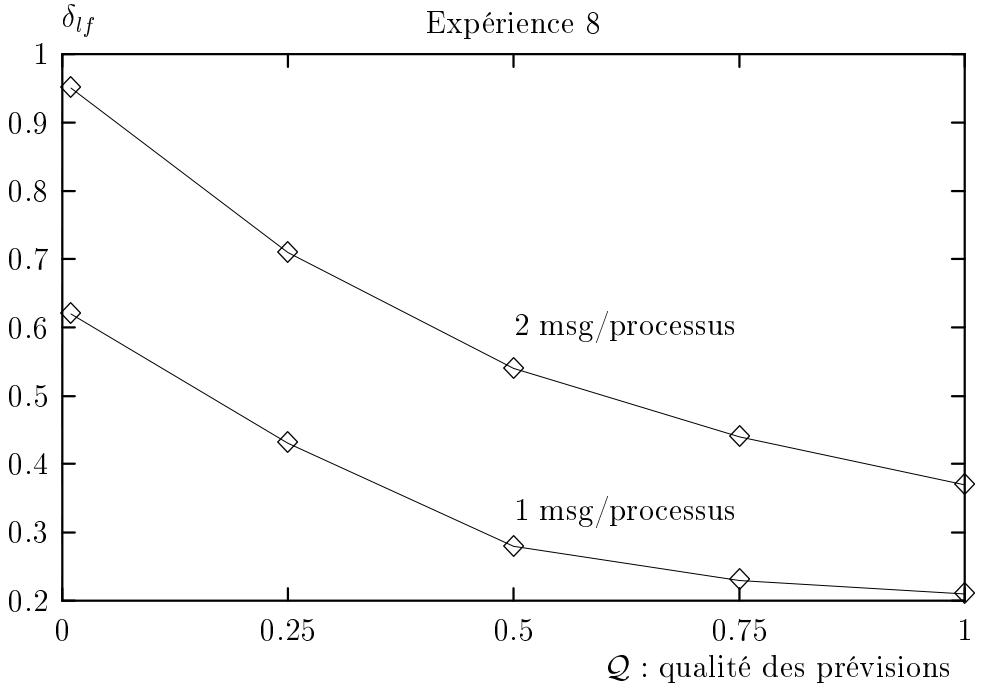

FIG. 4.13 – $\delta_{lf} = \mathcal{F}$ (qualité de la prévision) pour le tore

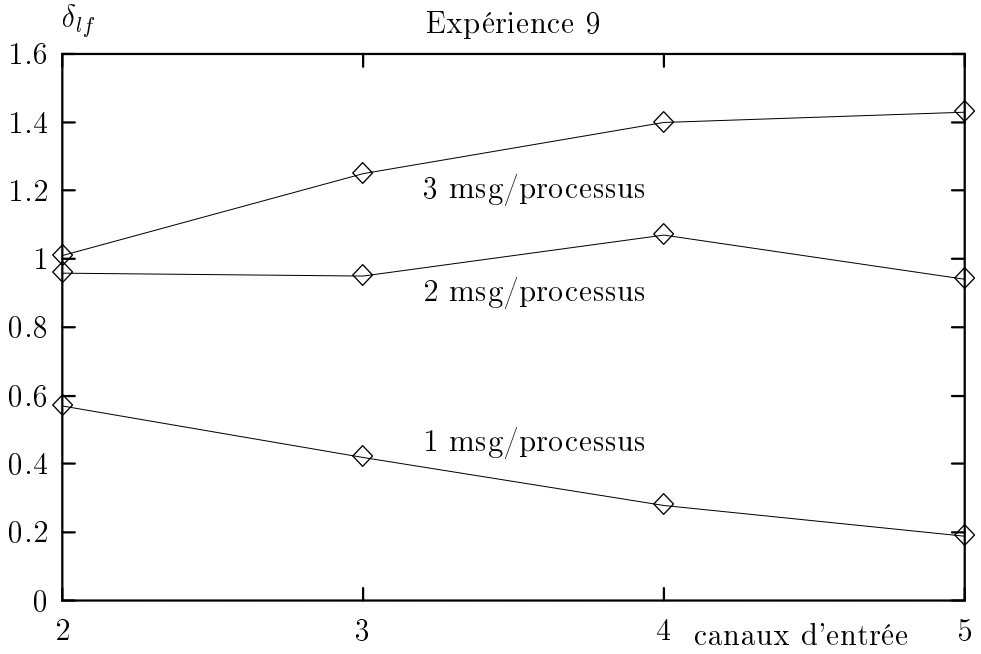

FIG. 4.14 – $\delta_{lf} = \mathcal{F}$ (nombre de canaux d'entrée) pour le *rev*

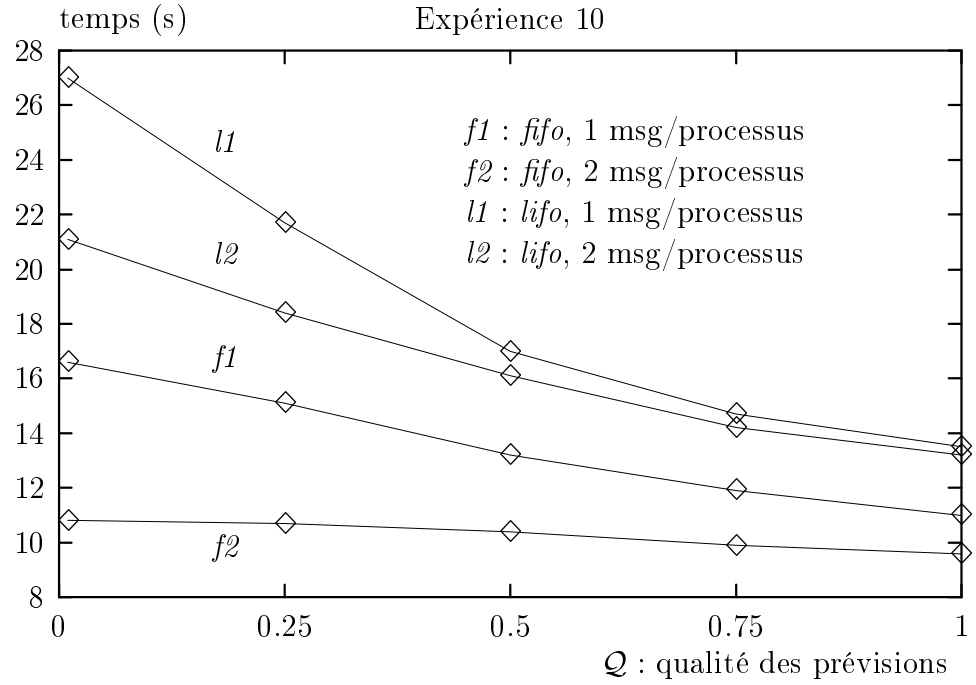

FIG. 4.15 – Temps = $\mathcal{F}$ (qualité de la prévision) pour le tore

Pour éviter des effets indésirables, nous avons préféré conserver toujours la même loi pour le temps de service. Nous avons estimé la prévision maximale comme étant le prochain temps de service ($\mathcal{Q} = 1$), et nous fournissons au noyau *Floria* une prévision égale à un certain pourcentage x de cette prévision maximale ($\mathcal{Q} = x$).

La figure 4.15 donne le temps de simulation en fonction de la qualité de la prévision pour un tore 16×16 (expérience 10). Elle permet de constater que la qualité de la prévision a plus d'importance pour des serveurs *lifo* que *fifo*. C'est une conséquence de l'optimisation du blocage des processus faite dans *Floria*, en particulier s'il y a suffisamment de messages (deux par processus dans ce cas-ci) les processus *fifo* ne se bloquent pratiquement plus et la prévision n'a plus d'influence. Les résultats obtenus avec des processus *lifo* coïncident avec ceux de Fujimoto.

La figure 4.16 donne, pour la même expérience, le pourcentage de messages *null* émis par rapport au nombre total de messages. Si l'augmentation de la qualité de la prévision permet de diminuer le nombre de messages *null* pour des serveurs *lifo*, par contre ce paramètre à peu d'influence sur le nombre de messages *null* pour des serveurs *fifo*. Pour des serveurs *lifo*, les messages *null* sont effectivement nécessaires pour pouvoir faire progresser l'horloge d'entrée he_i, et il peut être nécessaire de consommer plusieurs messages *null* avant de pouvoir délivrer au processus un message de la simulation ($\%null > 50\%$). Par contre, pour des processus *fifo*, compte tenu de la mise en œuvre des attentes, il y a souvent un message disponible pour le processus sans qu'il doive se bloquer ; il émettra, en moyenne, au plus un message *null* par message de la simulation.

Les figures 4.17 et 4.18 donnent le temps de simulation en fonction de la qualité de la prévision sur le réseau à nombre variable d'entrées (expériences 11 et 12). La qualité de la prévision est un facteur important, même pour des processus *fifo*, quand le nombre de canaux d'entrée par processus est élevé. Toutefois cet effet tend à diminuer quand le nombre de messages augmente (voir sections 4.9 et 4.10).

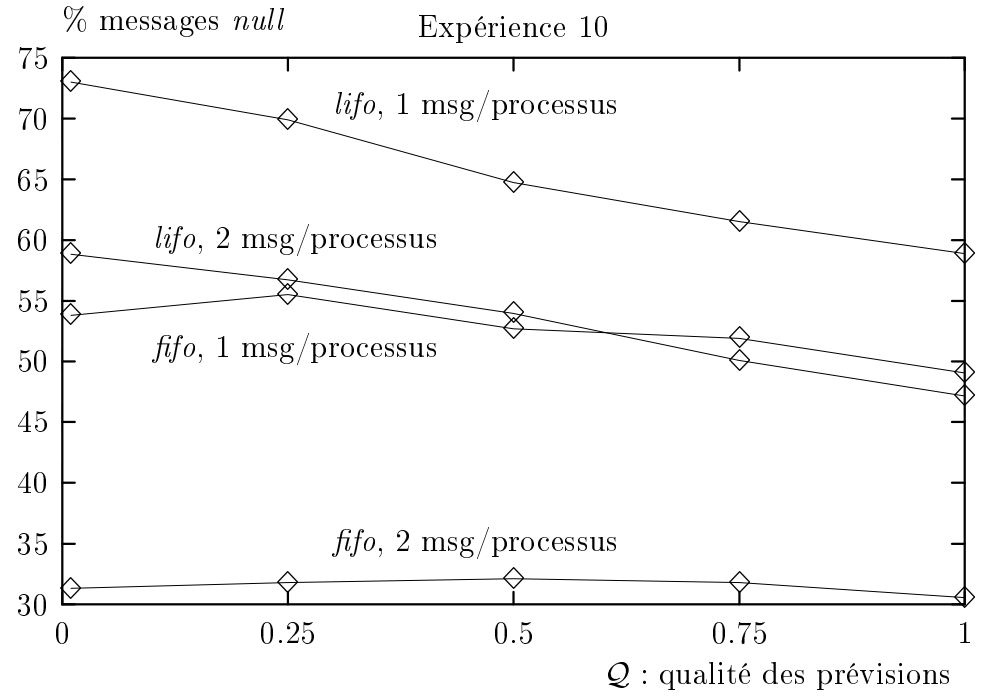

FIG. 4.16 – $\%null = \mathcal{F}$ (qualité de la prévision) pour le tore

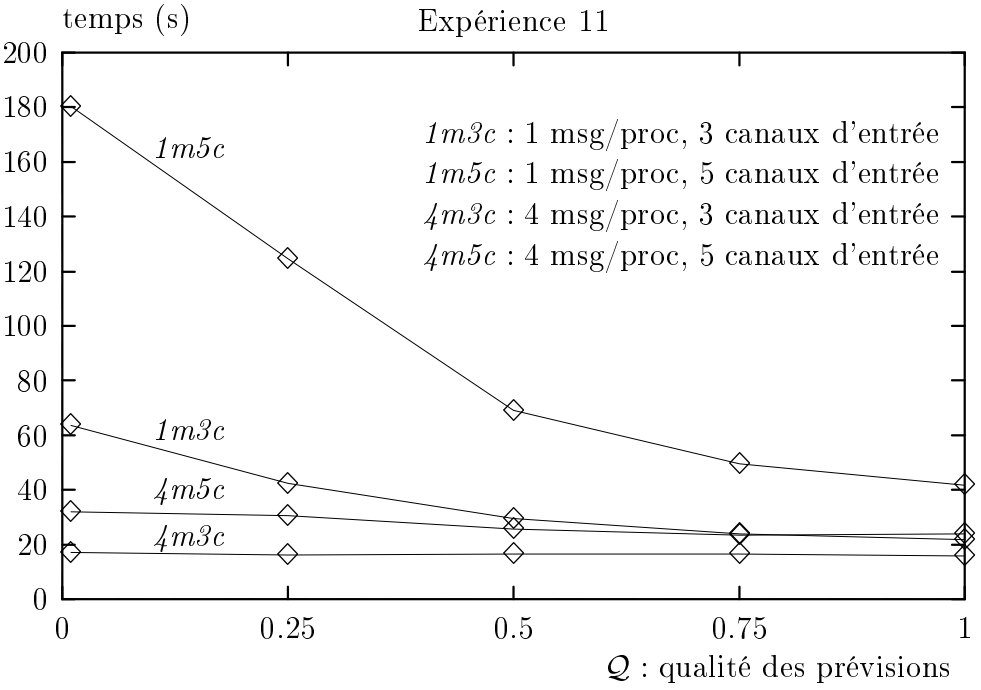

FIG. 4.17 – Temps = $\mathcal{F}$ (qualité de la prévision) pour le *rev*, processus *fifo*

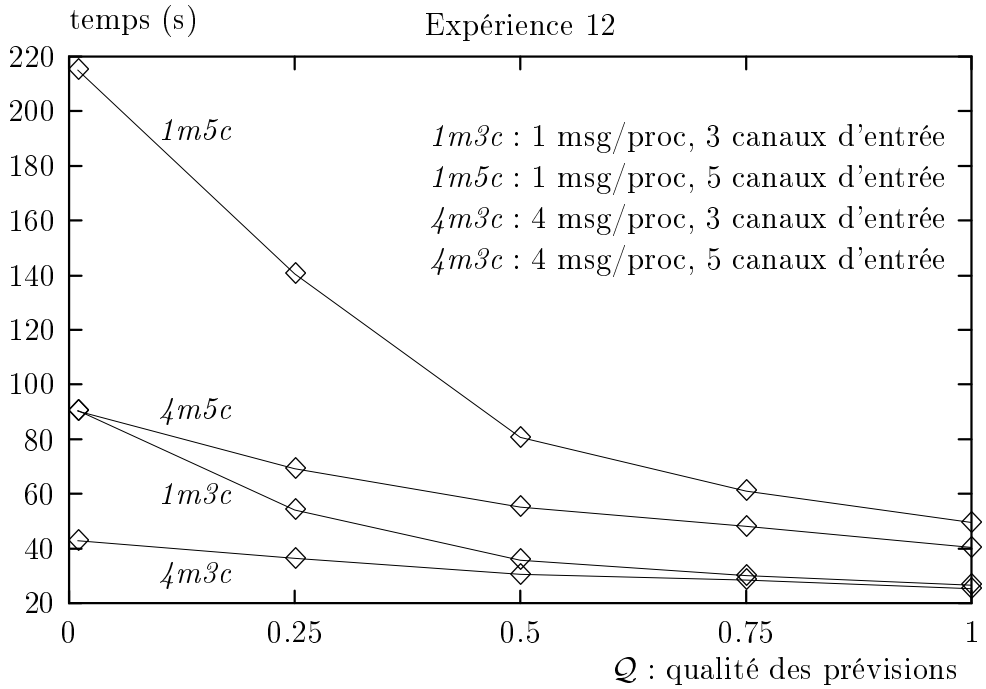

FIG. 4.18 – Temps = $\mathcal{F}$ (qualité de la prévision) pour le *rev*, processus *lifo*

4.9 Influence du nombre de messages

Nous étudions ici l'effet du nombre de messages en circulation sur le comportement du noyau. La population des messages liés au modèle reste constante pendant une simulation ; elle est fixée par le nombre de messages émis initialement par chaque processus, car le traitement d'un message produit toujours l'émission d'un message de la simulation. Les expériences ont été faites sur une durée de simulation fixée (5000 unités de temps virtuel), avec la même loi pour les temps de services ; le nombre de traitements de messages effectués reste donc constant.

L'augmentation du nombre de messages en circulation est pénalisante pour un simulateur séquentiel à cause de l'augmentation de la longueur de la file des événements. Dans une simulation répartie la taille des files de messages augmente également, mais ce phénomène est moins important parce qu'il existe a une file par processus, et non pas une file unique. De plus cet effet se fait surtout sentir pour des processus *lifo* (il faut parcourir toute la file pour accéder au dernier message). Par contre, intuitivement, comme l'algorithme de contrôle de la communication attend d'avoir un message sur chaque canal d'entrée d'un processus pour lui délivrer un message, l'augmentation du nombre de messages devrait conduire à une amélioration des performances de la simulation, grâce à une diminution des attentes.

La figure 4.19, donnant l'accélération pour une simulation du tore sur 8 processeurs, confirme partiellement cette intuition (expérience 13). L'amélioration se produit effectivement pour des processus *fifo*, par contre il n'y a pratiquement plus d'amélioration pour des processus *lifo* au delà de deux messages. La figure 4.20 donne, pour les mêmes expériences, le temps de simulation en fonction du nombre de messages par processus. Le nombre de messages a peu d'influence sur la durée totale de simulation quand une prévision maximale est utilisée, l'augmentation de l'accélération dans ces cas est donc

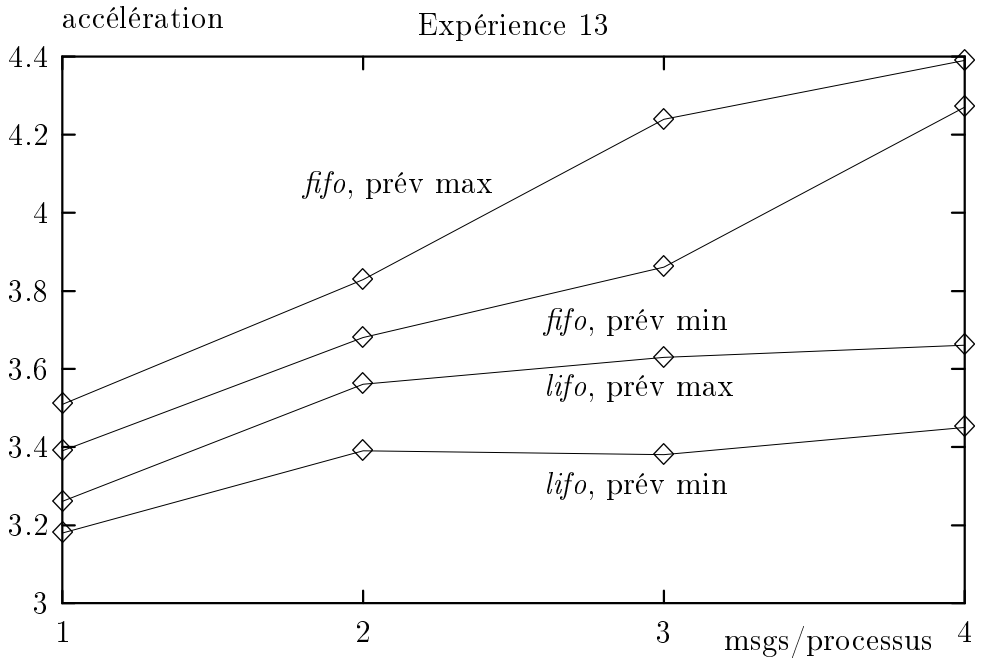

FIG. 4.19 – Accélération = $\mathcal{F}$ (nombre de messages) pour le tore

dûe essentiellement à l'augmentation du temps de simulation sur un seul processeur quand la charge en messages augmente. Par contre avec une prévision minimale le fait de passer de un à deux messages par processus fait diminuer le temps de simulation sur 8 processeurs : l'augmentation du nombre de messages de la simulation fait diminuer la nécessité d'attendre des messages *null*. Au delà de deux messages par processus, il n'y a plus d'effet. Cela est dû à la topologie du tore : un processus a deux canaux d'entrée, quand aucun canal d'entrée n'est vide (en moyenne), les messages supplémentaires n'améliorent plus les performances.

La figure 4.21 donne, pour les mêmes expériences, le temps passé dans l'exécution du modèle par opposition au temps passé dans le noyau, ce temps inclut le temps passé dans les manipulations de la file *Fconnus* par le processus. L'augmentation du nombre de messages a peu d'influence sur ce temps pour des processus *fifo* parce que la longueur de la file importe peu dans ce cas, le temps passé dans le modèle augmente de façon importante pour un processus *lifo* avec une prévision maximale parce qu'il faut parcourir toute la file à chaque fois.

Cet effet ne se fait pas sentir avec des processus *lifo* et une prévision minimale à cause de la mise en œuvre des blocages / réactivations de processus évoquée dans la section 3.5 : un processus bloqué en attente de message peut être réactivé inutilement par l'arrivée d'un message *null*. La prévision minimum avec un processus lifo *implique un grand nombre de messages null*, et donc de nombreux déblocages inutiles. Le temps modèle est alors important avec un seul message, et ensuite l'augmentation du temps de parcours des files dûe à l'augmentation du nombre de messages est compensé par la diminution du nombre de messages *null*; c'est ce qui explique la relative stabilité du temps modèle dans ce cas.

Les figures 4.22 et 4.23 donnent les temps de simulation sur le réseau à nombre variable d'entrées en fonction du nombre de messages par processus (expériences 14 et

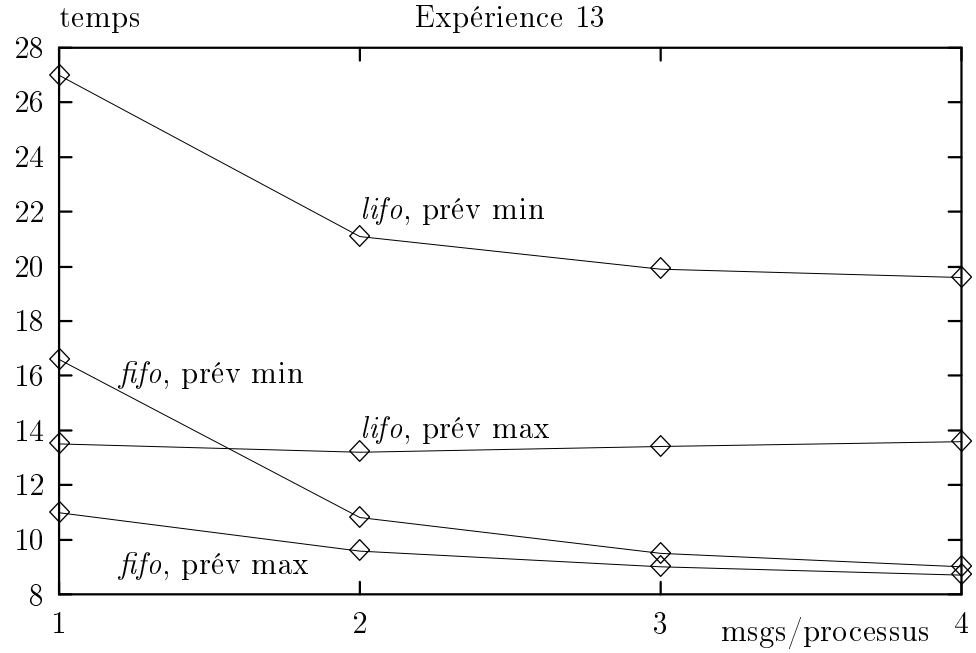

FIG. 4.20 – Temps = $\mathcal{F}$ (nombre de messages) pour le tore

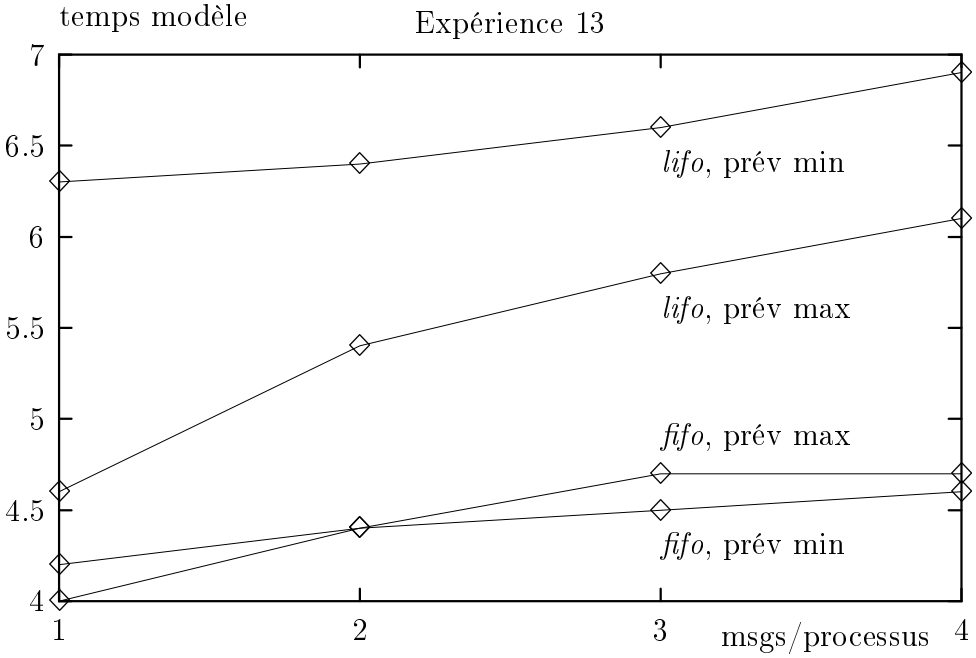

FIG. 4.21 – Temps modèle = $\mathcal{F}$ (nombre de messages) pour le tore

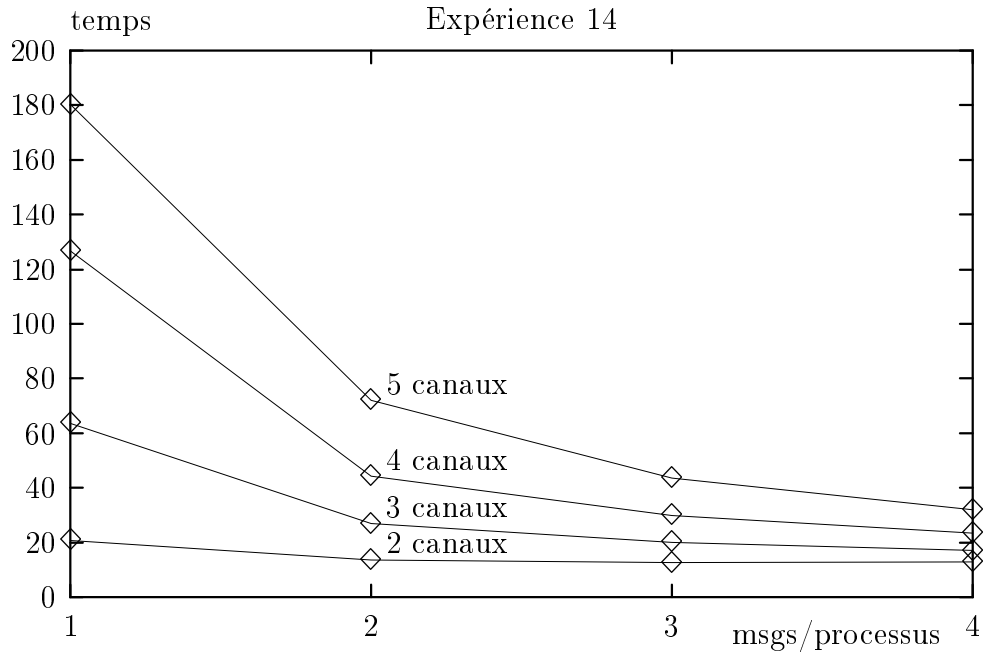

FIG. 4.22 – Temps = $\mathcal{F}$ (nombre de messages) pour le *rev*, processus *fifo*

15). Nous y constatons également une diminution des temps de simulation quand le nombre de messages augmente, diminution d'autant plus forte que le nombre de canaux d'entrée par processus est grand.

4.10 Influence de la topologie

Il est difficile d'évaluer l'impact de la forme du réseau de processus simulés sur les performances du simulateur, car il est difficile de quantifier ce paramètre. Pour ne pas avoir d'interférences avec des problèmes de placement nous n'avons considéré que des réseaux réguliers. Notre étude a porté essentiellement sur le nombre de canaux d'entrée pour chaque processus, car, compte tenu de l'algorithme de contrôle utilisé, cela semble a priori un paramètre important. Un autre facteur, a priori intéressant, le nombre et la taille des circuits dans le graphe, n'a pas été étudié.

La figure 4.24 donne le temps de simulation pour des réseaux dont le nombre d'entrées varie de 1 à 15, pour des processus *fifo* et *lifo* (expérience 16). Une augmentation importante du temps de simulation avec le nombre de canaux d'entrée par processus est constatée, surtout quand la prévision est de mauvaise qualité. Ce phénomène est assez indépendant du comportement des processus (*fifo* ou *lifo*). Les figures 4.25 et 4.26 donnent respectivement les pourcentages de messages *null* et le temps-modèle pour les mêmes expériences. Nous avons pû observer, dans cette expérience, d'une part une forte augmentation du nombre de messages *null*, à peu près indépendante de la prévision, et d'autre part une augmentation du temps-modèle liée fortement à celle-ci. Le nombre élevé de messages *null* a deux effets négatifs différents : le noyau passe du temps à les émettre et à les recevoir, mais comme nous l'avons indiqué précédemment ils peuvent également impliquer dans certains cas des réactivations inutiles des processus. C'est ce

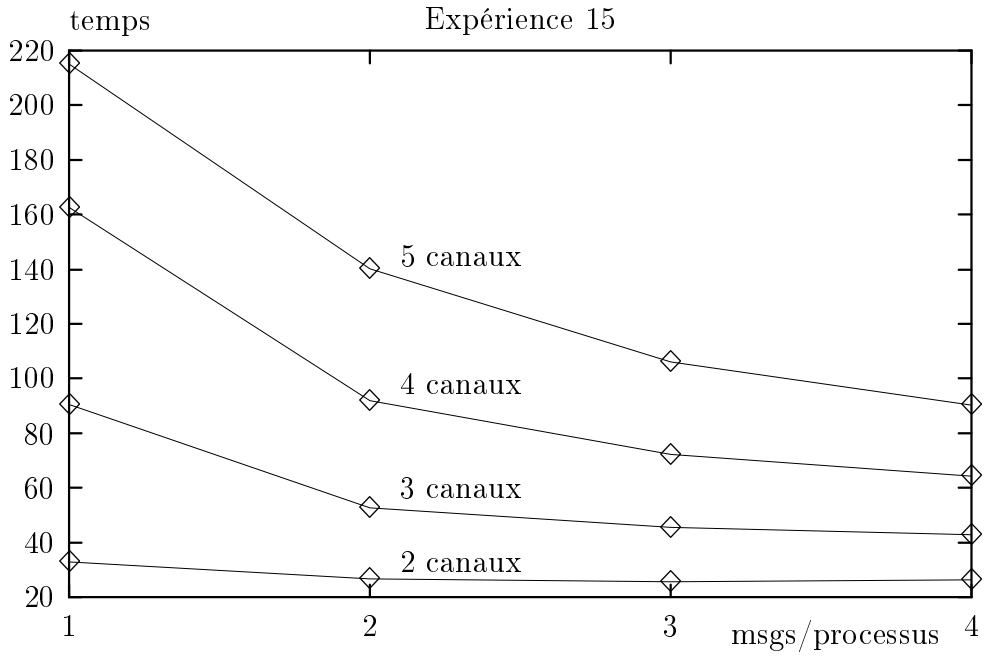

FIG. 4.23 – Temps = $\mathcal{F}$ (nombre de messages) pour le *rev*, processus *lifo*

phénomène qui explique le temps modèle plus élevé dans le cas de la prévision minimale : dans ce cas un processus doit recevoir plusieurs messages *null* avant de pouvoir traiter un message de la simulation.

4.11 Conclusion

Globalement les expériences réalisées ont montré qu'il est effectivement possible d'obtenir une diminution du temps de simulation, en répartissant une simulation avec un algorithme de contrôle de type conservatif. Même si l'efficacité (rapport entre le gain obtenu par la distribution et le nombre de processeurs utilisés) reste limitée, comme nous l'avons déjà signalé, l'important est de pouvoir diminuer les temps de simulation, et cet objectif est atteint.

Les résultats que nous avons obtenus indiquent que les performances des simulations réparties réalisées avec le support de *Floria* sont d'autant meilleures que la charge en calcul est importante et que le nombre de messages en circulation est élevé. Ces deux caractéristiques sont pénalisantes pour les simulations séquentielles, toutefois elles sont sources de parallèlisme potentiel. Les simulations qui se prêtent le mieux à la parallélisation sont donc celles qui posent le plus de problème en séquentiel, c'est à dire, justement celles qui justifient la parallélisation. Cela constitue une indication très encourageante pour l'utilisation de ce type de technique.

Les expériences effectuées ont permis de vérifier l'intérêt des désynchronisations introduites dans la gestion des horloges locales et d'entrée des processus. L'écart important observé entre les temps d'exécution des réseaux comportant des processus *fifo* ou *lifo* en est la démonstration. Les résultats obtenus avec *Floria* sur des réseaux de processus *lifo* sont compatibles avec ceux présentés par d'autres auteurs (notamment

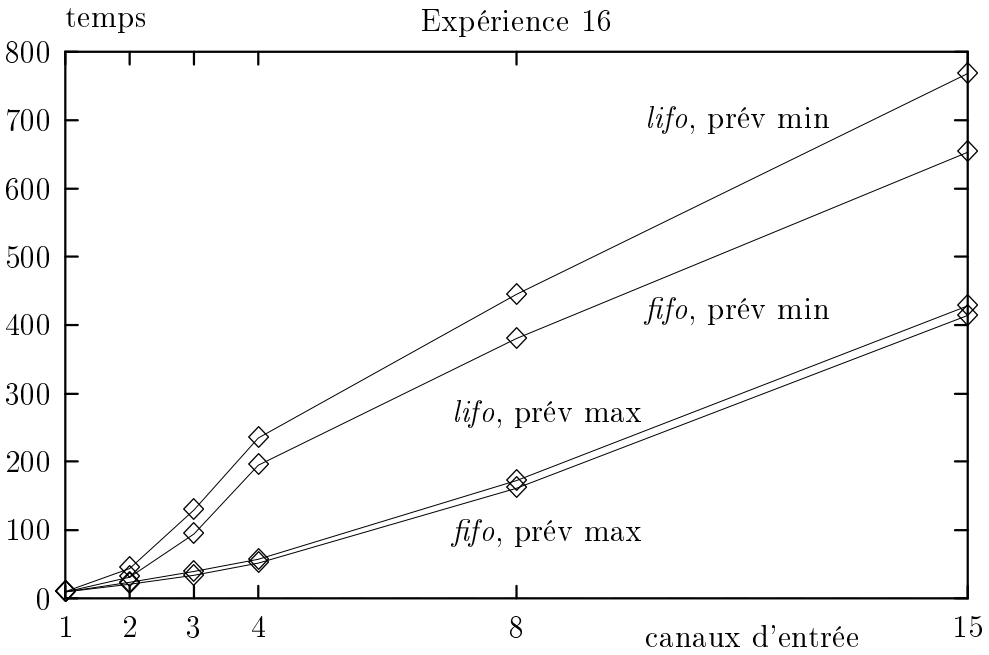

FIG. 4.24 – Temps = $\mathcal{F}$ (nombre de canaux d'entrée) pour le *rev*

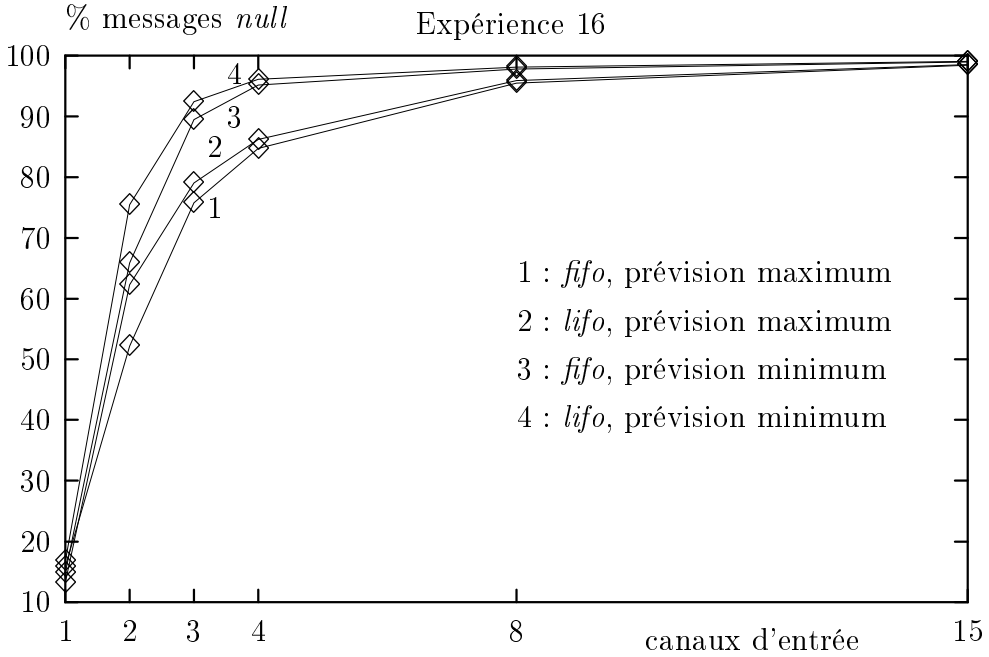

FIG. 4.25 – $\%null = \mathcal{F}$ (nombre de canaux d'entrée) pour le *rev*

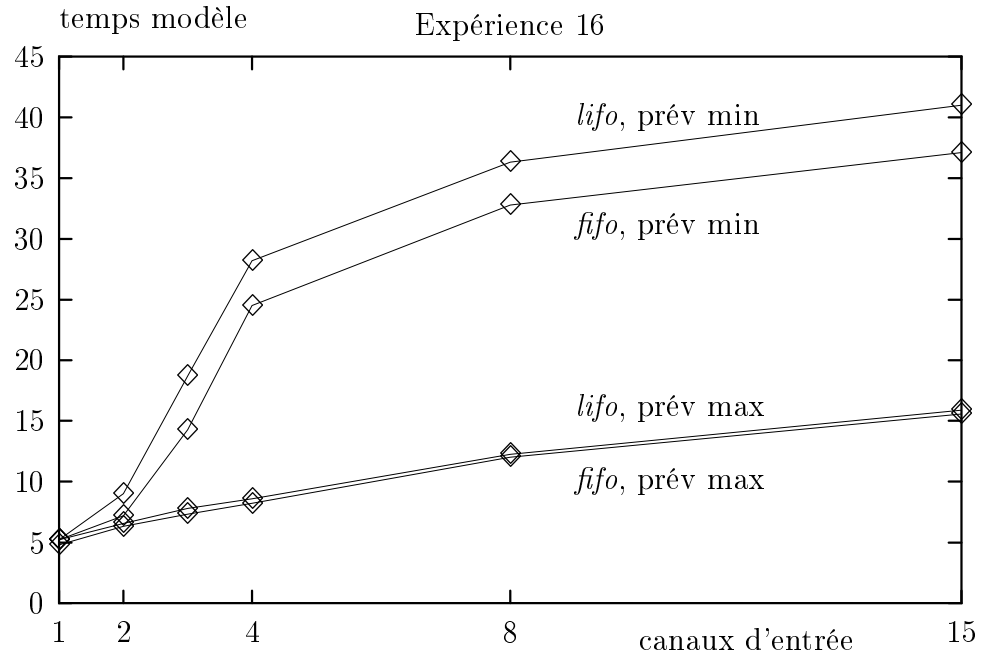

FIG. 4.26 – Temps modèle = $\mathcal{F}$ (nombre de canaux d'entrée) pour le *rev*

Fujimoto [Fuj88a, Fuj90]) pour des processus ayant un comportement quelconque. Ces résultats représentent le comportement de *Floria*, dans les conditions les moins favorables. Pour des processus *fifo*, les performances verifiées sont nettement supérieures, ce qui démontre le potentiel de ce noyau.

D'autres résultats intéressants concernent la qualité des prévisions fournies par les processus : elle est d'autant plus importante que les autres caractéristiques du modèle sont défavorables (nombre de canaux d'entrée, nombre de messages, etc.). De façon générale, la qualité des prévisions exerce une influence moins importante sur des réseaux de processus *lifo* que sur *fifo*. Nous retrouvons même des situations où la qualité des prévisions n'exerce quasiment pas d'influence sur la durée de la simulation, comme l'indique la figure 4.15. Encore une fois, ces résultats sont dûs à l'optimisation de la gestion des accès aux messages arrivés aux processus.

Chapitre 5

Des synchronisations et des ordres

L'ordre établi ne l'est souvent que par rapport aux ordres donnés par ceux qui ont ordonné de l'établir.

Pierre DAC

Dans les chapitres précédents nous avons présenté le problème de synchronisation dans une simulation répartie et des solutions pour la mise en œuvre répartie des simulations dirigées par le temps ou par les événements. Nous avons aussi présenté la mise en œuvre d'un noyau de système réparti destiné à permettre la réalisation de simulations réparties dirigées par les événements sur une machine multiprocesseur à mémoire distribuée. Le noyau construit introduit des mécanismes originaux en simulation répartie, notamment les désynchronisations permises entre les processus et leurs canaux d'entrée. Nous avons pu montrer l'efficacité de ces désynchronisations par les résultats des expériences effectuées (chapitre 4).

Dans le présent chapitre nous nous intéressons à mieux situer les concepts de synchronisation utilisés par le noyau *Floria* dans le contexte de l'algorithmique répartie. Nous nous intéressons aussi aux liens existants entre les techniques utilisées pour la simulation répartie et d'autres techniques de synchronisation pour les algorithmes répartis, notamment les synchroniseurs de réseaux. Dans le même esprit, nous analysons le rapport entre l'ordre établi par ces techniques et les autres relations d'ordre connues (causalité, *fifo*, etc.).

5.1 Une classification des méthodes de synchronisation

Dans la section 2.4, nous avons vu que les messages arrivant à un processus doivent lui être délivrés dans l'ordre de leurs estampilles, pour que le principe de causalité soit respecté. En fonction de l'approche choisie pour assurer le respect de la causalité par la simulation, il est possible de classifier les méthodes de synchronisation pour la simulation répartie en deux grands groupes : les méthodes « pessimistes » et les méthodes « optimistes ». Les méthodes dites pessimistes garantissent la livraison des messages aux processus dans l'ordre de leurs estampilles, le respect de la causalité

étant ainsi assuré a priori. Dans les méthodes optimistes, les messages sont délivrés aux processus dans l'ordre de leur arrivée, et la causalité est garantie a posteriori : lorsqu'une violation du principe de causalité est détectée, l'exécution de la simulation revient en arrière pour la corriger.

Les méthodes classiques optimistes (*Time-Warp*) et pessimistes (prévention ou détection d'interblocage) peuvent être considérées comme des solutions extrêmes au problème de synchronisation pour la simulation répartie. Il est en effet possible de concevoir des méthodes intermédiaires, ayant un degré moins fort de pessimisme, ou d'optimisme. Par exemple, dans [Meh91], Mehl propose une méthode optimiste faisant usage d'un optimisme restreint : chaque processus peut consommer les messages de façon optimiste, mais les émissions ne sont effectuées qu'une fois vérifiée la correction de l'exécution. Cette limitation de l'optimisme a pour but d'éviter la propagation de calculs éventuellement incorrects à d'autres processus. Le mécanisme de retour en arrière généralisé de *Time-Warp* cède ici sa place à un retour en arrière limité à chaque processus : la détection d'une violation de la causalité conduira à un retour en arrière sur l'exécution du processus impliqué, sans conséquence pour ses successeurs (*i.e.* sans envoi d'anti-messages). D'autres méthodes optimistes intermédiaires ont été présentées dans la section 2.8.

Les méthodes pessimistes peuvent également présenter des degrés différents de pessimisme. Par exemple, la méthode pessimiste mise en œuvre par *Floria* peut être considérée comme moins pessimiste que les méthodes classiques de prévention ou détection d'interblocages. En effet, dans une méthode pessimiste classique, un processus connaît toujours l'ordre des messages qu'il peut consommer : un processus arrivé à l'instant t dans le temps virtuel connaît forcément tous les messages qui lui ont été envoyés jusqu'à t. Le schéma de synchronisation mis en œuvre par *Floria* permet à un processus d'avancer son horloge locale sans qu'il soit obligé d'attendre que ses canaux d'entrée avancent d'autant (section 3.5.3) ; un processus peut ainsi ne connaître qu'une partie des messages qu'il peut consommer.

Nous pouvons détailler la classification des méthodes de synchronisation pour la simulation répartie au delà du simple aspect optimiste ou pessimiste de ces méthodes. Nous suggérons la classification suivante en quatre groupes, en fonction du degré d'optimisme/pessimisme présenté par les méthodes :

- *Pessimiste strict* : chaque processus connaît complètement l'ordre des messages qu'il peut consommer ; l'évolution du processus dans le temps virtuel est toujours conditionnée à l'avancement de son horloge d'entrée. Comme l'ordre des messages est garanti a priori, les actions des processus sont définitives ; des retours en arrière ne sont pas nécessaires. Dans cette catégorie se trouvent les méthodes pessimistes classiques : prévention ou détection et résolution d'interblocages.
- *Pessimiste souple* : le processus ne connaît qu'en partie l'ordre des messages qu'il peut consommer. Pour éviter des violations de la causalité, seuls les accès aux messages dont l'ordre est reconnu stable sont permis. Si le processus tente d'accéder à des messages dont il ne connaît pas l'ordre, il reste bloqué en attendant que l'ordre soit rétabli. L'ordre des messages est ainsi assuré a priori, mais l'évolution du processus dans le temps virtuel n'est pas forcément liée à l'arrivée de nouveaux messages ; cela rend cette méthode plus performante que les méthodes pessimistes classiques. Nous plaçons dans cette catégorie la méthode de synchro-

nisation utilisée par *Floria*.

- *Optimiste local* : le processus consomme les messages dans l'ordre où ils arrivent, sans contrôle préalable. L'évolution du processus n'est donc pas liée à l'arrivée de nouveaux messages. Toutefois, les messages produits par le processus ne sont envoyés à ses successeurs que lorsqu'ils sont garantis corrects. De cette façon, la détection d'une violation de la causalité provoque un retour en arrière de l'exécution du processus, sans affecter les autres processus. Un exemple de cette classe est la proposition de Mehl [Meh91].
- *Optimiste global* : les processus consomment les messages dans l'ordre où ils arrivent ; les messages produits par les processus sont aussitôt transférés vers leurs destinataires. La détection d'une violation de la causalité provoque un retour en arrière de l'exécution du processus et peut aussi provoquer l'envoi d'anti-messages pour annuler les messages envoyés pendant le calcul erroné. Ici nous retrouvons la méthode *Time-Warp* [Jef85].

Le mécanisme de gestion de la file de messages connus de chaque processus (*Fconnus*) mis en œuvre par *Floria* permet en quelque sorte d'introduire une certaine dose « d'optimisme sûr » dans la synchronisation des processus. Évidemment, nous ne pouvons par parler ici d'optimisme au même sens que dans les méthodes de synchronisation comme *Time-Warp*, car aucune action pouvant impliquer des retours en arrière n'est autorisée par le noyau ; le déroulement de la simulation n'est jamais remis en cause. Toutefois, le schéma de synchronisation imposé par le noyau *Floria* représente une évolution par rapport aux techniques pessimistes classiques. En effet, la méthode de synchronisation mise en œuvre dans *Floria* présente le plus grand degré d'optimisme que l'on puisse ajouter à une méthode pessimiste classique sans prendre le risque de remettre en cause l'évolution des processus et de les contraindre à effectuer des retours en arrière.

5.2 Analyse de la synchronisation offerte par *Floria*

Nous avons vu dans le chapitre 4 que *Floria* permet d'obtenir des accélérations supérieures à celles obtenues avec des méthodes pessimistes classiques, notamment lorsque le comportement des processus est *fifo*. Dans cette section nous essayerons de mieux comprendre les causes du gain en vitesse d'exécution fournit par le noyau dans ce cas.

Dans une simulation, un événement $[e, t_e]$ daté dans le temps simulé peut être causalement dépendant de certains des événements qui le précédent, c'est à dire, de certains des événements $[e', t'_e] \mid t'_e < t_e$. Pour être sûr de la dépendance (ou indépendance) causale entre e et e' il serait nécessaire de regarder la structure du modèle de simulation, à la recherche de liens entre ces événements. Dans une simulation séquentielle, les événements sont traités dans l'ordre strict de leur occurrence, pour éliminer tout risque de violation de la causalité. Toutefois, pour l'exécution répartie d'une simulation, il est nécessaire de faire en sorte que les événements causalement indépendants puissent être traités simultanément, dans le but d'accélérer le calcul.

La structuration du modèle de simulation sous la forme d'un réseau de processus permet de résoudre en partie ce problème. Les interactions entre processus sont re-

présentées par des messages estampillés ; ces échanges de messages représentent donc les seuls liens causaux existants entre les exécutions internes aux processus. Il devient ainsi possible de déceler des événements causalement indépendants et de les traiter simultanément.

Les méthodes classiques de synchronisation pessimiste pour la simulation répartie ne permettent de traiter simultanément que les événements causalement indépendants se trouvant sur des processus distincts ; à l'intérieur de chaque processus tous les événements sont traités dans un ordre chronologique strict, notamment les interactions avec d'autres processus (échanges de messages). Les désynchronisations mises en œuvre dans *Floria* rendent possible le traitement dans un ordre non chronologique de certains des événements sans lien causal à l'intérieur d'un processus, pour obtenir ainsi des gains significatifs en vitesse de calcul.

La figure 5.1 illustre la différence de comportement, dans le temps réel, entre *Floria* et une méthode pessimiste classique. Un processus p_i à l'instant $hv = t$ doit attendre t_{att} unités de temps virtuel pour ensuite envoyer un message m. Cette action est indépendante de tout ce qui pourra se passer sur p_i pendant la période d'attente $[t \cdots t + t_{att}]$, y compris l'arrivée de nouveaux messages. Dans cette situation, une méthode pessimiste classique ne prendrait pas en compte cette indépendance ; l'envoi du message m ne serait effectué que lorsque tous les messages pouvant arriver pendant la période d'attente seraient reçus. Le noyau *Floria* évite ce « pessimisme excessif », qui peut dégrader les performances de la simulation, en permettant au processus d'effectuer ses actions sans prendre en compte l'arrivée de nouveaux messages.

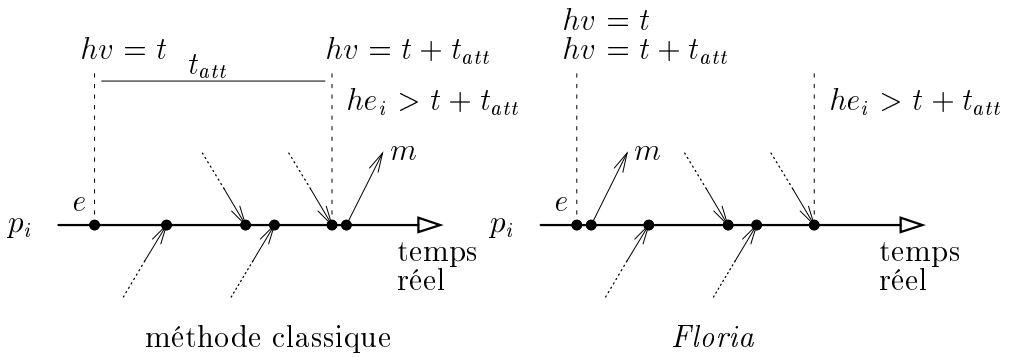

FIG. 5.1 – Traitement d'actions causalement indépendantes

Nous pouvons mieux illustrer le gain apporté par notre noyau avec l'exemple suivant, où nous comparons des traces d'exécution obtenues avec *Floria* à d'autres obtenues avec une méthode pessimiste classique. Considérons un processus ayant un seul canal d'entrée, sur lequel arrive un message à chaque t_e unités de temps virtuel. L'arrivée des messages est aussi distribuée de façon uniforme dans le temps réel. Le processus prend les messages dans l'ordre où ils arrivent (politique *fifo*), les traite pendant t_s unités de temps virtuel et les envoie à la sortie. Nous considérons que le traitement de chaque message ne consomme pas de temps réel.

La figure 5.2 montre l'évolution de l'horloge locale de ce processus en utilisant une méthode de synchronisation classique et aussi avec *Floria*, dans le cas où les messages

arrivent plus rapidement (dans le temps virtuel) que le processus ne peut les traiter : $t_s > t_e$. Nous pouvons constater qu'avec *Floria* l'horloge locale du processus avance plus rapidement qu'avec une méthode classique, et que la différence Δ_t entre les durées d'exécution des deux méthodes (dans le temps réel) grandit au fur et à mesure que la simulation progresse.

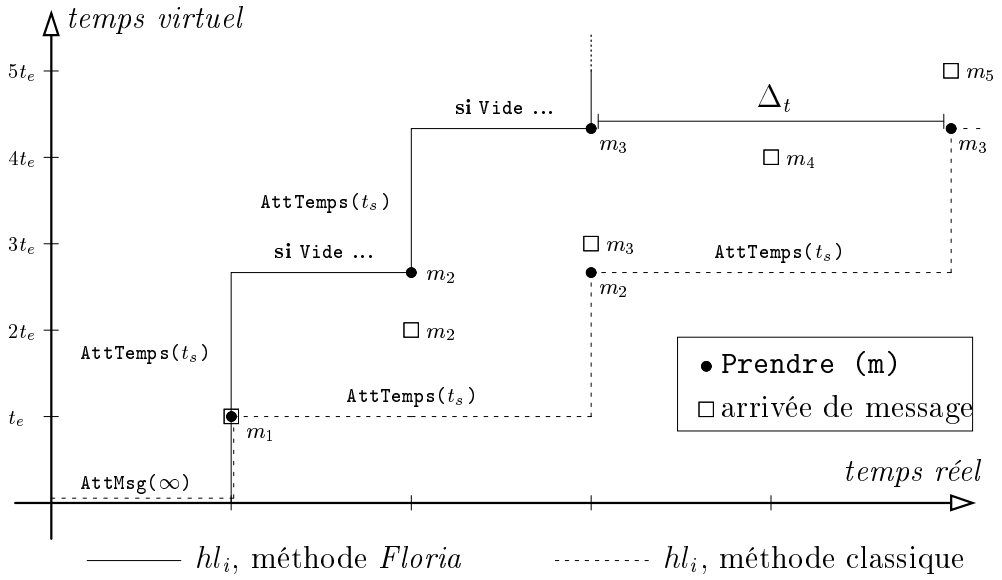

FIG. 5.2 – Évolution avec $t_s > t_e$

La figure 5.3 montre le cas contraire, c'est à dire, une situation dans laquelle le processus traite les messages plus rapidement qu'ils ne lui arrivent : $t_s < t_e$. Dans ce cas les performances de *Floria* sont similaires à celles obtenues avec une méthode pessimiste classique.

Dans les schémas pessimistes de synchronisation connus auparavant, les aspects liés à la communication se confondent avec ceux liés à l'évolution des processus. La désynchronisation entre l'évolution d'un processus et l'évolution de ses canaux d'entrée n'a été rendue possible qu'à cause de la structure choisie pour notre noyau, qui a permis de séparer le contrôle de l'évolution des processus de la synchronisation des communications. Il est important aussi de remarquer que les désynchronisations proposées par *Floria* dans le contrôle de l'évolution des processus seraient également possibles avec l'emploi d'autres méthodes pessimistes pour la mise en œuvre de la couche de contrôle des communications.

La stratégie de synchronisation proposée pour le noyau *Floria* lui permet de tirer profit de certaines des caractéristiques des processus du modèle de simulation, quand cela est possible, sans pour autant faire des hypothèses préalables sur leur comportement, qui pourraient mener à une spécialisation indésirable du noyau. Il est intéressant de pousser la recherche dans cette direction, en déterminant d'autres caractéristiques pouvant être exploitées par le noyau de façon dynamique et transparente, notamment dans le domaine des prévisions sur le comportement futur des processus.

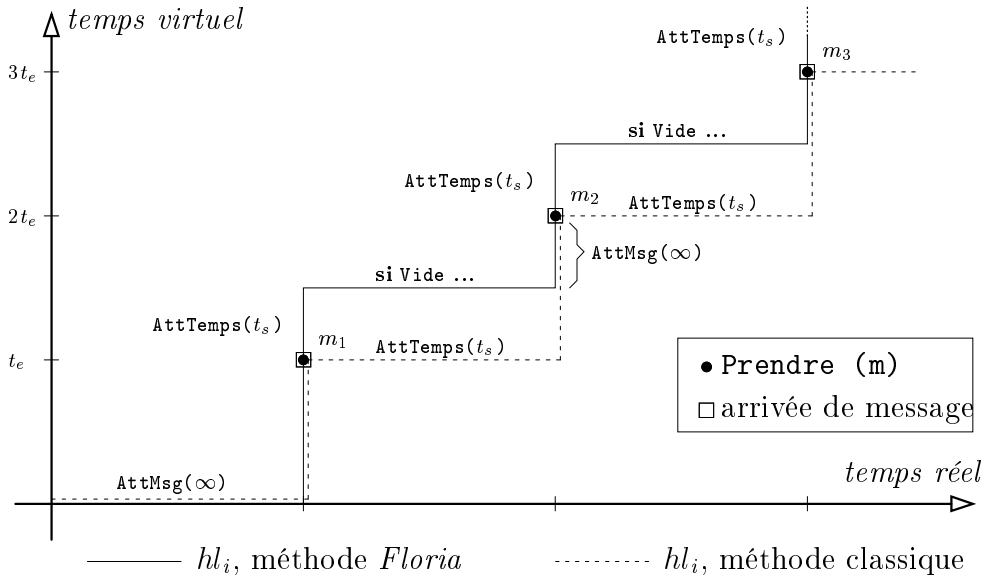

FIG. 5.3 – Évolution avec $t_s < t_e$

5.3 Simulation répartie et algorithmes synchrones

Dans la section 2.3 nous avons introduit la notion d'algorithme synchrone et de synchroniseurs, dans le but de présenter des mécanismes d'exécution pour la mise en œuvre répartie de simulations dirigées par le temps. Il y a, en effet, beaucoup de similarités entre les techniques utilisées pour la synchronisation des simulations réparties et celles utilisées pour l'exécution d'algorithmes synchrones. Nous allons maintenant analyser plus profondément les relations existantes entre les algorithmes synchrones et la simulation répartie, en essayant aussi d'étendre cette analyse à la simulation dirigée par les événements.

Un algorithme réparti synchrone [Awe85, HR88] est composé d'un ensemble de processus s'exécutant sur un support d'exécution synchrone caractérisé par une horloge globale unique, dont chaque cycle correspond à une pulsation, et un service d'échange de messages dont le délai de transfert est inférieur à une pulsation d'horloge (un message émis à la pulsation p sera reçu par son destinataire pendant cette même pulsation p). Un processus envoie au plus un message vers chaque successeur, au début de chaque pulsation ; ensuite les messages concernant la pulsation actuelle sont reçus et traités, et le processus attend la pulsation suivante :

Processus d'algorithme synchrone :
  lors d'une pulsation p **faire**
    envoyer les messages relatifs à la pulsation p ;
    traiter tous les messages arrivés pendant p ;
    attendre la pulsation suivante $p+1$;
  fin ;

Pour permettre l'exécution d'algorithmes synchrones sur des supports d'exécution asynchrones, Awerbuch [Awe85] introduit la notion de *synchroniseur* : un algorithme

réparti (asynchrone) construisant un support d'exécution synchrone, comme caractérisé ci-dessus, à partir d'une machine parallèle asynchrone. Ces schémas de synchronisation sont fondés sur la notion de *sûreté* des processus par rapport aux pulsations : un processus d'un algorithme synchrone est dit *sûr* par rapport à une pulsation p dès lors qu'il ne pourra plus provoquer d'actions de l'algorithme synchrone dans le réseau (*i.e.* des envois de messages), relativement à cette pulsation [Ada90].

Les synchroniseurs peuvent être *forts* ou *faibles*, selon le degré de synchronisation qu'ils imposent aux processus de l'algorithme synchrone. Avec un synchroniseur fort, tous les processus du programme synchrone se trouvent à la même pulsation, à un instant donné. Un synchroniseur faible preserve uniquement une simultanéité logique, tout en permettant des décalages entre les processus. Ainsi, la mise en œuvre d'un synchroniseur dit fort exige la sûreté de tous les processus du programme synchrone par rapport à la même pulsation avant qu'ils ne puissent passer à la pulsation suivante. Par contre, avec un synchroniseur faible, un processus sûr par rapport à une pulsation p peut passer à la pulsation suivante $p+1$ si, et seulement si, tous ses voisins sont aussi sûrs par rapport à p.

Comme nous l'avons montré dans la section 2.3, le concept d'algorithme synchrone et les mécanismes employés pour son exécution peuvent être utilisés pour la mise en œuvre répartie de simulations dirigées par le temps. Rappelons, dans une simulation dirigée par le temps le temps simulé avance par petits pas de durée δ. Il est possible de considérer une telle simulation comme un algorithme synchrone dans lequel chaque pulsation correspond à δ unités de temps simulé ; cette simulation peut alors être exécutée à l'aide d'un synchroniseur.

Il est possible aussi de considérer un algorithme synchrone comme un cas spécial de simulation dirigée par les événements, dans lequel nous associons à chaque pulsation une durée d'une unité de temps virtuel. Les messages arrivant pendant une même pulsation sont traités en même temps ; leur traitement ne consomme pas de temps virtuel. Dans ce contexte, la comparaison entre le support d'exécution des processus d'un algorithme synchrone et l'environnement de temps virtuel proposé par le noyau *Floria* est immédiate. Nous pouvons donc exprimer des algorithmes synchrones avec les primitives offertes par *Floria* :

```
Processus d'algorithme synchrone :
   répéter
      envoyer messages ;                    /* envoyer les msgs relatifs à la pulsation p */
      Prévision (1) ;                       /* prochaine émission seulement à p + 1 */
      tant que Vide est faux faire          /* traiter tous les messages arrivés pendant p */
         msg ← Prendre (Premier) ;
         traiter msg ;
      fin tant que ;
      AtttendreTemps (1) ;                  /* attendre la pulsation suivante p + 1 */
   jusqu'à Temps > t_max ;                  /* fin de l'exécution */
```

Il n'est cependant pas possible de faire le chemin contraire, c'est à dire, d'exprimer tout programme de simulation comme un algorithme synchrone. En effet, ce modèle de calcul synchrone ne permet pas de modéliser des processus pouvant répondre de façon instantanée aux message qui lui arrivent ; la réponse à un message reçu pendant une pulsation p ne peut être émise qu'au début de la pulsation suivante $p+1$. Toutefois, si le

modèle de simulation ne présente pas ce genre de comportement (réponses immédiates), il est toujours possible de déterminer une durée adéquate dans le temps virtuel pour les pulsations, et ainsi de considérer le programme de simulation comme un algorithme synchrone. Dans la section 2.3 nous avons présenté l'utilisation de cette approche pour la mise en œuvre répartie des simulations dirigées par le temps.

La comparaison entre les modèles de simulation et les algorithmes synchrones peut être étendue aux mécanismes utilisés pour permettre leur exécution sur une machine parallèle. Cette comparaison est notamment intéressante dans le cadre de la simulation dirigée par les événements. Dans une mise en œuvre synchrone de la simulation dirigée par les événements, l'évolution des processus est gérée de manière semblable à celle d'un synchroniseur fort : tous les processus de l'algorithme synchrone (ou du modèle de simulation) avancent de concert, se trouvant tous à la même pulsation (au même instant du temps simulé), au même moment (cf. section 2.4).

Analysons le cas de la technique pessimiste de prévention d'interblocages à l'aide des messages *null*, celle utilisée dans la version actuelle de *Floria*. Ce schéma est très proche du principe de fonctionnement du synchroniseur faible de type α [HR88]. Le principe de fonctionnement de ce synchroniseur est simple : il consiste à envoyer des messages de contrôle *sûr(p)* à tous les voisins d'un processus dès que celui-ci se trouve sûr par rapport à sa pulsation courante p. Le processus sera autorisé à passer à la pulsation suivante $p+1$ une fois qu'il aura reçu un message *sûr(p)* de chacun de ses voisins (comme les canaux sont *fifo*, le processus aura alors reçu tous les messages du calcul synchrone envoyés vers lui à p) [Ada90] [1].

Les messages de contrôle *sûr(p)* véhiculés par le synchroniseur α ont la même fonction que les messages $[null, t]$ dans la méthode de prévention d'interblocages : communiquer aux successeurs du processus émetteur que celui-ci s'est rendu à une certaine date virtuelle, pour permettre leur avancement. Les prévisions, explicites dans le cas de la simulation, deviennent implicites avec les algorithmes synchrones, car à chaque pulsation au plus un message du calcul circule par chaque canal : en recevant un message estampillé p sur un canal d'entrée, le processus sait que le prochain message sur ce canal aura comme estampille $p+1$. Cette analogie avec le schéma des messages *null* permet de proposer une amélioration au mécanisme de base du synchroniseur α : au lieu d'envoyer des messages *sûr(p)* à tous les voisins d'un processus, le synchroniseur ne les envoie qu'aux voisins auxquels aucun message du calcul synchrone n'a été envoyé pendant p. Ainsi, au cours de chaque pulsation, exactement un message circule sur chaque canal du réseau. Un processus peut alors passer à la pulsation suivante lorsqu'il aura reçu de chacun de ses voisins un message quelconque (du calcul ou de contrôle) relatif à la pulsation p.

Nous pouvons aussi établir cette analogie dans l'autre sens : le mécanisme mis en œuvre par les messages *null* peut être vu comme un synchroniseur dans lequel seulement les pulsations considérées « utiles » sont communiquées aux successeurs du processus. De cette façon, lorsqu'un processus à l'instant t sait qu'il passera les k prochaines pulsations sans envoyer des messages du calcul, il envoie un message $[null, t + k \times \delta]$ à chacun de ses successeurs. Encore une fois, cette comparaison peut aussi être utilisée

[1] Dans [Awe85], Awerbuch propose un synchroniseur appelé α, basé sur ce même principe. Sa condition de sûreté est toutefois plus forte : un processus n'est sûr que si tous les messages qu'il a émis à p ont été reçus, ce qui implique des mécanismes d'acquittement.

pour l'amélioration du mécanisme du synchroniseur α, en permettant des messages de contrôle *sûr(p + k)*.

Ainsi, les synchroniseurs peuvent être considérés comme un cas particulier des mécanismes de synchronisation pour la simulation répartie. En effet, ces deux schémas cherchent à construire une référence temporelle indépendante du temps physique, avec des caractéristiques appropriées pour l'exécution des applications auxquelles ils se destinent. Cette similarité est d'autant plus flagrante lors de la comparaison des solutions algorithmiques proposées pour les mettre en œuvre ; celles-ci reposent sur les mêmes principes, et sont très proches. Cela permet alors de concevoir des synchroniseurs forts ou faibles, analogues aux mises en œuvre synchrones et asynchrones de la simulation répartie (cf. section 2.4). Nous pourrions même envisager un synchroniseur optimiste, inspiré par exemple de *Time-Warp*. Il est en autre intéressant d'étudier les techniques proposées pour la mise en œuvre d'autres types de synchroniseurs [Awe85, HR88] et d'évaluer leur intérêt pour la simulation répartie.

5.4 L'ordre construit par le noyau

Dans un calcul réparti, il est utile d'établir un ordre entre les événements qui se produisent dans le système, afin de simplifier le contrôle de l'exécution répartie et assurer la correction de ses résultats. Dans cette section nous nous intéressons particulièrement aux relations d'ordre possibles entre les événements d'interaction entre processus : les émissions et les réceptions de messages.

Une des plus simples relations d'ordre sur ces interactions est celle connue sous le nom *fifo*, de l'anglais *first in, first out*. Elle établit que, entre deux processus, les événements correspondant à leurs interactions sont perçus dans la même séquence. Par exemple, lorsqu'un processus p_i envoie les messages m_a et m_b vers un processus p_j, si *envoi*(m_a) précède *envoi*(m_b), alors *réception*(m_a) doit précéder *réception*(m_b), pour que l'ordre *fifo* soit respecté. Dans la figure 5.4, les interactions entre p_i et p_j respectent l'ordre *fifo*, au contraire de celles entre p_j et p_k.

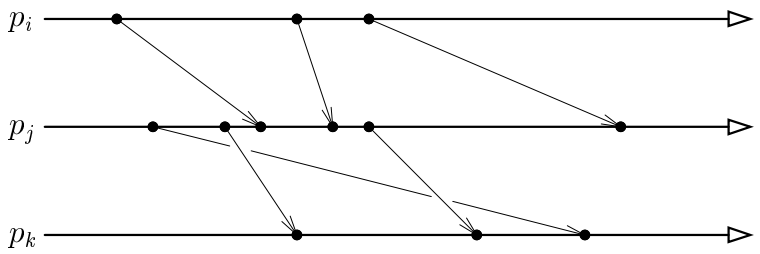

FIG. 5.4 – Non-respect de l'ordre *fifo*

Le principe de causalité présenté dans le chapitre 1 montre que tout événement ne peut être dépendant que des événements ayant lieu avant lui. Toutefois, un événement n'est pas forcement dépendant de *tous* les événements se produisant avant lui, mais seulement de ceux appartenant, pour ainsi dire, à sa branche du calcul. Dans [Lam78], Lamport définit une relation de *dépendance potentielle* (« happened before ») sur les

événements se produisant dans un calcul réparti, notée « $\rightarrow$ » (« $e_a \rightarrow e_b$ » signifie alors « e_a précède e_b »). Cette relation, vérifiée pour tout calcul réparti, est définie par les règles suivantes :

- Un événement peut dépendre causalement de tous les événements se produisant avant lui sur le même processus ($\tau(e_x)$ indique la date d'exécution de l'événement e_x, dans le temps physique) : $e_x \rightarrow e_a \ \forall \ x \mid \tau(e_x) < \tau(e_a),\ e_x, e_a \in p_i$.
- Si e_a est l'émission d'un message m_a et r_a sa réception, alors $e_a \rightarrow r_a$.
- Si $e_a \rightarrow e_b$ et $e_b \rightarrow e_c$ alors $e_a \rightarrow e_c$.

L'ordre causal consiste à reproduire, sur les réceptions de messages par un processus, les dépendances causales existantes entre leurs émissions [RST91]. En d'autres mots, si m_a et m_b sont deux messages dont les émissions sont causalement liées, alors l'ordre de leurs réceptions, par un même processus, devra respecter cette relation : $e_a \rightarrow e_b \Rightarrow r_a \rightarrow r_b$. Par exemple, pour que l'ordre causal soit respecté dans l'exécution répartie de la figure 5.5, le processus p_k devrait recevoir le message m_a avant le message m_c, car l'émission de m_a précède causalement celle de m_c : $e_a \rightarrow e_b \rightarrow r_b \rightarrow e_c$.

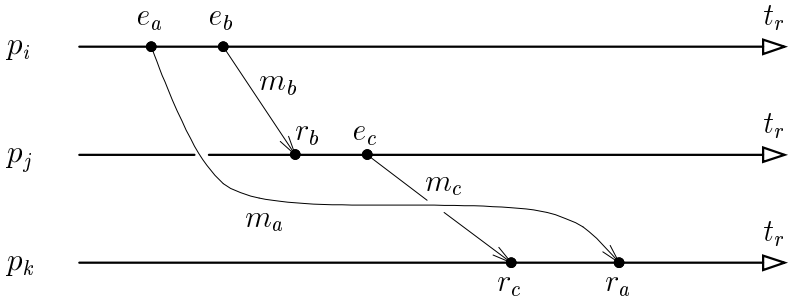

FIG. 5.5 – Non respect de l'ordre causal

Une autre relation d'ordre importante est celle caractérisée par des communications synchrones entre les processus. Dans une communication synchrone, l'émission et la réception d'un message ont lieu en même temps et peuvent être considérées comme un seul événement, constituant alors un point de synchronisation entre les processus. Comme les communications entre processus ne sont jamais synchrones (au sens « instantanées ») dans le temps physique, Charron-Bost *et al.* introduisent dans [CBMT91] la notion de calcul « réalisable avec des communications synchrones », pour définir une classe de calculs qui respectent l'ordre synchrone. Un calcul est dit réalisable avec des communications synchrones si (et seulement si) l'ordre des interactions entre les processus ne serait pas altéré en faisant l'hypothèse de communications instantanées. La figure 5.6 montre des exemples de calculs réalisable (cas $\mathcal{A}$) et non-réalisable (cas $\mathcal{B}$) avec des communications synchrones.

Un calcul réalisable avec des communications synchrones est caractérisé par l'absence de dépendances cycliques entre ses émissions et réceptions de messages. Ces dépendances cycliques constituent des structures appelées « couronnes » [CBMT91] : une couronne de dimension k est caractérisée par la séquence de dépendances $e_1 \rightarrow r_2$, $e_2 \rightarrow r_3$, $\cdots$, $e_{k-1} \rightarrow r_k$, $e_k \rightarrow r_1$, où e_x et r_x représentent respectivement l'émission et la réception d'un message m_x. Il est montré dans [CBMT91] qu'un calcul réparti

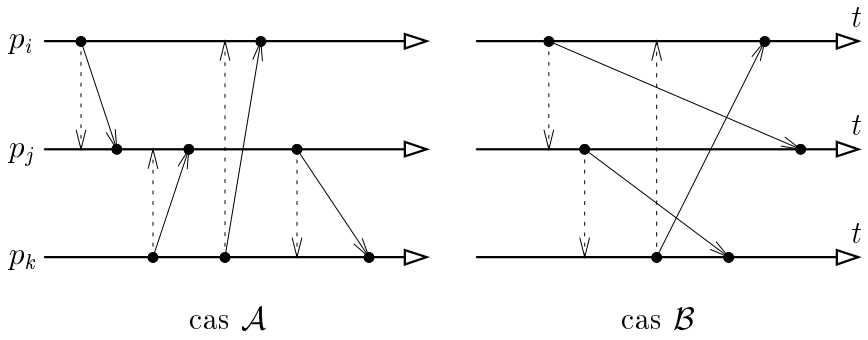

FIG. 5.6 – Respect et non-respect de l'ordre synchrone

asynchrone est réalisable avec des communications synchrones si et seulement si son diagramme d'exécution ne présente pas de couronnes. La figure 5.7 montre une couronne de dimension 3.

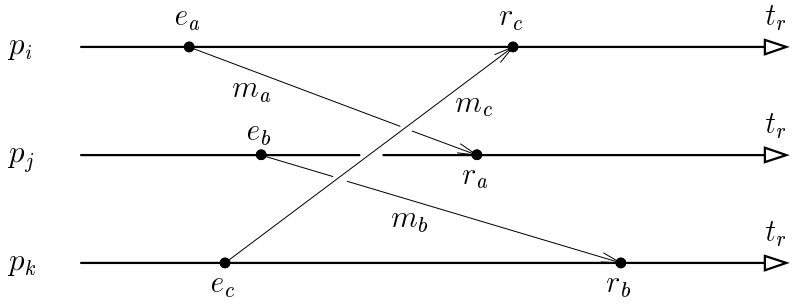

FIG. 5.7 – Une « couronne » de dimension 3

Dans [CBMT91], Charron-Bost *et al.* montrent qu'il est possible d'établir une hiérarchie entre ces relations d'ordre : *synchrone* $\subset$ *causal* $\subset$ *fifo*. De cette façon, tout calcul synchrone respecte aussi les ordres causal et *fifo*, et ainsi de suite. Par exemple, les deux calculs de la figure 5.6 respectent les ordres causal et *fifo*.

L'environnement de temps virtuel construit par le noyau *Floria*, décrit dans la section 3.1, est caractérisé par les propriétés $\mathcal{P}_1$ et $\mathcal{P}_2$ qui définissent respectivement une horloge virtuelle commune aux processus de l'application et des communications entre processus instantanées par rapport à cette horloge virtuelle. Les communications instantanées définies par la propriété $\mathcal{P}_2$ construisent alors un support d'exécution synchrone dans le référentiel de temps établi par $\mathcal{P}_1$.

Le modèle de calcul établi par les propriétés $\mathcal{P}_1$ et $\mathcal{P}_2$ est toutefois trop rigide et pourrait restreindre les possibilités d'applications du noyau *Floria*, car il oblige un processus à consommer les messages dans l'ordre où il les reçoit (dans le temps virtuel). Nous avons donc rajouté à ce modèle la propriété $\mathcal{P}_3$, qui permet de séparer la réception d'un message de sa consommation effective (section 3.1.1). La file de messages connus par un processus, établie par la propriété $\mathcal{P}_3$, permet au processus de traiter les messages qu'il reçoit dans l'ordre qu'il souhaite. De plus, cette propriété permet de considérer

un processus comme toujours prêt à recevoir les messages qui lui sont adressés.

Dans un calcul de type synchrone l'émission et la réception d'un message peuvent être vues en fait comme un seul événement, qui constitue ainsi un point de synchronisation entre l'émetteur et le récepteur du message. Dans le modèle de calcul défini par les propriétés $\mathcal{P}_1$, $\mathcal{P}_2$ et $\mathcal{P}_3$ les processus sont toujours disponibles pour recevoir des messages ; un processus ne reste donc jamais bloqué en attendant de pouvoir émettre un message, car son récepteur sera toujours prêt pour l'interaction. Toutefois, un processus peut rester bloqué en attendant une réception. La propriété $\mathcal{P}_3$ a introduit un comportement asymétrique entre émetteur et récepteur, au contraire de celui défini pour le langage CSP [Hoa84], équivalent au calcul défini uniquement par $\mathcal{P}_1$ et $\mathcal{P}_2$.

Si l'on considère les communications entre processus comme des couples *[émission, réception]* le modèle de calcul reste synchrone ; ces deux événements se produisent en même temps, comme défini par les propriétés $\mathcal{P}_1$ et $\mathcal{P}_2$. Par contre, si les communications sont représentées par des couples *[émission, consommation]*, le calcul devient non seulement asynchrone, mais aussi les ordres causal et *fifo* ne sont pas préservés. Ceci est dû au fait de que le processus peut consommer dans le désordre les messages qui lui sont envoyés, même si ceux-ci lui sont délivrés dans l'ordre de leurs estampilles.

Nous connaissons maintenant les caractéristiques de l'ordre établi par le noyau dans le temps virtuel. Mais, quelles sont les conséquences des synchronisations effectuées dans le temps virtuel, pour construire cet ordre, sur le déroulement du calcul asynchrone sous-jacent, dans le temps physique ? La mise en œuvre des algorithmes de synchronisation dans le temps simulé suppose l'offre d'un service de communications asynchrone *fifo* par la machine cible. Cet ordre est donc implicitement respecté par le calcul.

Nous pouvons montrer que le calcul réparti sous-jacent dans le temps physique respecte l'ordre causal. Pour cela considérons la réception, par un processus d'un programme de simulation, de deux messages $[m_a, t_a]$ et $[m_b, t_b]$, $t_a \neq t_b$, tels que $e_a \rightarrow e_b$ (e_x correspond à l'émission d'un message m_x et r_x à sa réception). Si le calcul respecte l'ordre causal, la réception de m_a doit précéder celle de m_b ($r_a \rightarrow r_b$). Comme le mécanisme de synchronisation pour la simulation répartie délivre les messages aux processus dans l'ordre croissant de leurs estampilles, alors $r_a \rightarrow r_b \Leftrightarrow t_a < t_b$. Il nous reste montrer que $e_a \rightarrow e_b \Rightarrow t_a < t_b$.

Si $e_a \rightarrow e_b$ alors il existe une suite d'événements $a_1, \ldots, a_n$ sans événements intermédiaires, tels que $e_a \rightarrow a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_n \rightarrow e_b$. Chacun de ces événements a_i est estampillé $t(a_i)$ dans le temps simulé. Deux situations sont possibles pour les estampilles de deux événements consécutifs $a_i \rightarrow a_{i+1}$: s'ils se trouvent sur le même processus, alors $t(a_i) \leq t(a_{i+1})$; sinon, a_i correspond à une émission, a_{i+1} à une réception et nous avons $t(a_i) = t(a_{i+1})$. Donc $t(a_1) \leq t(a_2) \leq \cdots \leq t(a_n)$ et par conséquent $t(e_a) \leq t(e_b)$. Comme par hypothèse $t_a \neq t_b$ alors $t(e_a) < t(e_b)$, ce qui est équivalent à $t_a < t_b$.

Donc $e_a \rightarrow e_b \Rightarrow t_a < t_b$, et par conséquent $r_a \rightarrow r_b$, ce qui garantit le respect de l'ordre causal dans le temps réel. Toutefois, nous avons jusqu'ici considéré seulement des événements se produisant à des dates distinctes dans le temps virtuel. Considérons deux messages $[m_a, t_a]$ et $[m_b, t_b]$, $t_a = t_b$, émis par deux processus différents vers un troisième processus : l'ordre de leurs livraisons n'est absolument pas assuré par le mécanisme de synchronisation, même si leurs émissions sont liées causalement ($e_a \rightarrow e_b$). Donc l'ordre causal n'est pas respecté par le calcul sous-jacent dans le temps réel pour des

événements ayant la même date d'occurrence dans le temps virtuel. Dans [Meh92], Mehl discute ce problème, et propose l'utilisation d'un schéma de contrôle de l'ordre causal, superposé au mécanisme de synchronisation dans le temps virtuel, pour garantir l'ordre de livraison des messages estampillés à une même date virtuelle.

Nous venons de constater que les synchronisations du noyau dans le temps virtuel ont pour conséquence d'assurer *au minimum* l'ordre causal dans le temps réel. Analysons maintenant s'il n'assure plus que cet ordre. Nous pouvons appliquer le test de la « couronne » à un calcul de simulation répartie, pour vérifier s'il est réalisable avec des communications synchrones. Attribuons aux émissions et réceptions des messages m_a, m_b et m_c de la couronne de la figure 5.7 des estampilles dans le temps virtuel : $t(e_a)$, $t(r_a)$, $t(e_b)$, $t(r_b)$, $t(e_c)$ et $t(r_c)$. La structure de la couronne implique $e_a \rightarrow r_c \Rightarrow t(e_a) < t(r_c)$, $e_b \rightarrow r_a \Rightarrow t(e_b) < t(r_a)$ et $e_c \rightarrow r_b \Rightarrow t(e_c) < t(r_b)$ (nous considérons que tous les événements ayant lieu sur un même processus se produisent à des dates distinctes dans le temps simulé). La synchronisation dans le temps virtuel impose des communications instantanées dans cette référence de temps : $\forall x\ t(e_x) = t(r_x)$. Comme il n'existe pas de solution satisfaisant toutes ces contraintes, la présence d'une couronne est impossible. La synchronisation dans le temps virtuel impose donc un calcul sous-jacent, dans le temps réel, réalisable avec des communications synchrones.

If est important de remarquer que l'existence d'une couronne ne serait possible que si $t(e_a) = t(r_a) = t(e_b) = t(r_b) = t(e_c) = t(r_c)$, c'est à dire, que si nous considérions tous les événements ayant lieu à la même date virtuelle. Dans ce cas reste valide l'analyse effectuée sur les relations d'ordre causal entre événements simultanés dans le temps virtuel.

Pour construire un environnement d'exécution sur lequel les processus puissent s'exécuter de façon asynchrone et *fifo* dans le temps virtuel, nous sommes obligés de passer par une étape synchrone dans le temps virtuel. L'environnement constitué par la machine physique offre toutefois ces deux propriétés dans le temps réel (asynchronisme et *fifo*). Le choix d'une autre référence temporelle rend nécessaire une étape très synchrone, aussi bien dans le temps simulé que dans le temps réel. Est-ce une conséquence des communications instantanées dans la nouvelle référence temporelle, ou cette étape synchone serait-elle nécessaire même avec des communications non instantanées ? Il nous semble que le choix d'une référence de temps dont l'évolution n'est pas liée à celle du temps physique implique la résynchronisation complète du programme réparti, pour qu'il s'exécute sur une base cohérente par rapport à la nouvelle référence temporelle. Les désynchronisations permises au programme doivent alors être établies par rapport à cette base synchrone. Si les communications entre les processus n'étaient pas instantanées, elles auraient quand même une durée bien définie dans le temps simulé, ce qui impliquerait encore l'existence d'une base d'exécution cohérente dans cette référence temporelle.

Conclusion et perspectives

Dans cette thèse nous avons présenté la réalisation et l'évaluation d'un noyau de système réparti destiné à permettre l'exécution de programmes de simulation dirigée par les événements sur des machines multiprocesseur à mémoire répartie. Nous avons également situé les problèmes liés à la synchronisation de simulations dans le contexte de l'algorithmique répartie, et examiné les liens existants entre les techniques utilisées pour les résoudre et des techniques connues pour la synchronisation d'autres types de calcul réparti (synchroniseurs, etc.).

La version actuelle du noyau est réalisée directement sur le système d'exploitation (sous-ensemble d'UNIX) de l'iPSC d'INTEL. Comme nous n'avons utilisé que le service d'échange asynchrone de messages offert par cette machine, la migration de *Floria* vers d'autres machines multiprocesseur de type MIMD ne pose pas de difficultés majeures.

La structure conçue pour le noyau *Floria* est divisée en trois couches, dont chacune se charge de résoudre une partie précise du problème de gestion d'une simulation répartie : la distribution du modèle, les communications instantanées et l'évolution des processus dans le temps simulé. Contrairement aux techniques conventionnelles utilisées pour la mise en œuvre des simulations parallèles, dans *Floria* nous avons séparé la gestion des communications dans le temps simulé de la gestion de l'évolution des processus. Malgré cette séparation en couches, des bonnes performances sont observés lors des expériences avec le noyau. En effet, l'introduction de désynchronisations supplémentaires entre l'exécution d'un processus et l'état de ses canaux d'entrée a permis la diminution des temps d'exécution des simulations et aussi l'atténuation de l'influence de la qualité des prévisions sur les performances.

Dans le cadre d'une simulation de réseaux de files d'attente, des processus réalisant la mise en oeuvre de divers types de files ont été écrits (*fifo*, *lifo*, *quantum*, avec ou sans préemption, etc.). L'expérience montre que les primitives offertes par le noyau permettent d'exprimer facilement l'évolution des processus dans le temps virtuel.

Il est possible d'énumérer un bon nombre de suites possibles pour ce travail, allant de l'amélioration de la mise en œuvre actuelle du noyau *Floria* à une meilleure compréhension des relations d'ordre imposées par l'utilisation d'une référence abstraite de temps. La version actuelle du noyau *Floria* utilise un gestionnaire de processus légers dont les possibilités sont en effet assez limitées, et qui est intimement lié à la structure du noyau. Il serait alors intéressant de mener une réflexion sur l'adaptation de la structure de *Floria* à un système d'exploitation offrant les services de gestion de processus légers, tels que les micro-noyaux de système réparti CHORUS ou MACH [Bla90, ZGMB84]. Cela rendrait d'ailleurs plus facile la migration de *Floria* sur d'autres machines supportant le micro-noyau utilisé.

La version actuelle du noyau *Floria* utilise comme méthode de base pour la synchronisation dans le temps simulé la prévention d'interblocages à l'aide de messages *null*. Nous avons réalisé des essais avec une méthode de détection et résolution d'interblocages (section 2.5.2), et les premiers résultats sont tout à fait encourageants. L'utilisation conjointe d'une méthode de prévention et d'une méthode de résolution d'interblocages nous semble aussi une voie prometteuse. Avec des bonnes prévisions, la méthode de prévention d'interblocages affiche des performances supérieures à celles de la méthode de détection. Cette dernière présente toutefois l'avantage de rester valide même en absence de prévisions, et les performances des deux méthodes restent assez proches avec des mauvaises prévisions. Ces deux méthodes sont donc complémentaires en ce qui concerne la sensibilité à la qualité des prévisions, et leur utilisation conjointe peut servir à combler les handicaps dus à ce paramètre.

Cette coopération entre les deux méthodes pessimistes va dans le même sens que les mécanismes d'accélération pour la méthode de prévention d'interblocages (section 2.5.1). Ces mécanismes servent à mettre à jour les *temps_ canal* plus rapidement que les messages *null* ne le feraient, surtout lorsque les prévisions fournies par les processus sont de mauvaise qualité, et que le réseau d'interconnexion comporte beaucoup de circuits. En effet, ces mécanismes d'accélération sont en général des versions simplifiées d'algorithmes de détection et résolution d'interblocages.

Par souci de simplicité, la version actuelle du noyau *Floria* ne permet pas le partage de variables entre les processus du modèle. Toutefois, la mise en œuvre de variables partagées dans un contexte de temps virtuel implique des problèmes de synchronisation intéressants, en plus du contrôle des accès concurrents aux variables. Le problème ici provient du fait que les accès aux variables partagées dans une simulation parallèle doivent être effectués dans l'ordre de leur date d'occurrence dans le temps simulé. Par exemple, toute opération d'écriture sur une variable partagée $\mathcal{X}$ à l'instant simulé t_e doit être exécutée après les opérations de lecture de $\mathcal{X}$ ayant lieu à des instants simulés $t_l < t_e$, et vice-versa. Comme les processus du programme de simulation peuvent, à un moment donné, se trouver à des dates distinctes dans le temps simulé, il devient nécessaire de contrôler l'ordre d'accès des processus aux variables partagées. Quelques solutions pour la mise en œuvre de variables partagées en simulations parallèles sur des machines à mémoire répartie sont présentées en détail dans [Meh93].

Les modèles acceptés par la version actuelle de notre noyau sont statiques. La création d'un processus pendant le cours de l'exécution d'une simulation répartie implique la synchronisation de ce processus et des nouveaux canaux par rapport aux processus qui l'entourent. Il est important d'assurer que le nouveau processus soit inséré dans un contexte cohérent de temps virtuel, c'est à dire, que les messages que ce processus pourra désormais envoyer ne puissent pas provoquer des violations du principe de causalité chez ses destinataires. Ceux-ci doivent être mis au courant de la création du nouveau processus et ajuster leurs horloges d'entrée en conséquence. Ce problème est analogue à celui posé par l'introduction de nouveaux processus dans le système d'exploitation Isis [BT85, BTKS88]. Dans Isis, les processus sont organisés en groupes, et le transfert de messages à l'intérieur d'un groupe obéit à l'ordre causal ; l'inclusion dynamique de nouveaux processus dans le système pose le problème de maintien de cet ordre dans le groupe auquel le nouveau processus appartient.

Il serait intéressant d'utiliser le noyau *Floria* pour le support d'un langage réel de si-

mulation dirigée par les événements. Cela nous permettrait d'utiliser notre noyau dans un cadre plus réaliste, et ainsi mieux évaluer ses possibilités. D'autre part, cette expérience nous permettrait d'étudier les mécanismes de calcul automatique de la prévision sur les canaux d'entrée. En effet, la définition de bonnes valeurs pour les prévisions sur le comportement futur des processus peut être une tâche assez compliquée, comme nous avons pu constater d'après les modélisations effectuées à titre d'exemple dans la section 3.6. Des études ont été réalisées pour l'automatisation de ce calcul sur des classes de modèles spécifiques, comme les réseaux de files d'attente et les réseaux de Petri [WL89, KH90].

En principe, il n'est pas possible d'ajouter des possibilités de calcul automatique des prévisions directement sur le noyau *Floria*, car ces prévisions sont fonction du comportement interne des processus, auquel le noyau n'a pas accès. De plus, le noyau doit rester indépendant d'un langage ou modèle de simulation précis. Les prévisions doivent donc être calculés par les processus eux mêmes. Toutefois, l'utilisation de *Floria* pour le support d'un langage de simulation implique une phase de traduction de ce langage vers un programme contenant des appels aux primitives offertes par le noyau. Pendant cette phase de traduction, il est possible d'ajouter des mécanismes permettant à un processus de déterminer automatiquement ses prévisions. Par exemple, nous avons étudié la mise en œuvre sur notre noyau d'un sous-ensemble du langage ESTELLE [907], utilisant un temps virtuel via la clause `delay` d'ESTELLE. Il est possible de générer automatiquement des prévisions pour les processus écrits dans ce langage, en analysant l'automate d'états de chaque processus et les transitions contenant des émissions de messages.

Sur un plan plus général, la conception d'un noyau réparti pour les applications pilotées par un temps virtuel est très intéressante par les contraintes de synchronisation qu'elle définit sur les communications : tout message émis à la date virtuelle t est reçu à cette même date t. On trouve une approche analogue dans les mécanismes de contrôle des accès aux données utilisant un ordre créé par l'estampillage [BG81, MOO87]. Ceci suggère une étude plus approfondie des propriétés liées à la communication qu'offrent les noyaux des systèmes répartis [CBMT91]. Depuis les schémas totalement asynchrones jusqu'au schéma très contraint défini par la propriété $\mathcal{P}_2$, il semble important de dégager des types de communication de base et de caractériser proprement les applications qui les nécessitent. Le noyau *Floria* est un pas concret dans cette direction.

Bibliographie

[907] ISO 9074. *Proposed draft addendum to ISO 9074 :1989 — Estelle tutorial.* ISO 9074 :1989 ISO/IECJTC1/SC21/F60.

[Ada90] M. Adam. Synchronisme des systèmes distribués. Thèse, Université de Rennes I, Mai 1990.

[AP88] F. André and J.-L. Pazat. Le placement de tâches sur des architectures parallèles. *Technique et science informatiques*, 7(4) :385–401, 1988.

[Awe85] B. Awerbuch. A new distributed depth-first search algorithm. *Information Processing Letters*, 20(3) :147–150, 1985.

[Aya89a] R. Ayani. Parallel simulation on shared memory multiprocessors. Research report, Dept. of Telecommunication Systems and Computer Systems, The Royal Institute of Technology – Sweden, March 1989.

[Aya89b] R. Ayani. A parallel simulation scheme based on distances between objects. *Distributed Simulation, by The Society for Computer Simulation*, pages 113–118, 1989.

[Aya91] R. Ayani. Parallel discrete-event simulation on shared memory multiprocessors. *International Journal in Computer Simulation*, pages 81–97, 1991.

[BCM87] R.L. Bagrodia, K. M. Chandy, and J Misra. A message-based approach to discrete event simulation. *IEEE Transactions on Software Engineering*, 13(6) :654–665, June 1987.

[BDMN73] G.M. Birtwistle, O.J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula begin.* Chartwell-Bralt Ltd., 1973. 391 pp.

[BG81] P.A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *Computing Surveys*, 13,2 :185–221, June 1981.

[BJ85] O. Berry and D. Jefferson. Critical path analysis of distributed simulation. In *Proceedings of the SCS Distributed Simulation Conference, San Diego*, pages 57–60, January 1985.

[Bla90] D. L. Black. Scheduling support for concurrency and parallelism in the mach operating system. Research report 125, Carnegie Mellon University – Computer Science, 1990.

[Bry77] R.E. Bryant. Simulation of packet communication architecture computer system. Tech. report TR-188, Massachussets Institut of Technologie – LCS, 1977.

[BT85] K.P. Birman and Joseph T.A. Replication and fault-tolerance in the isis system. *ACM SIGOPS*, 10 :79–86, 1985.

[BT87] G. Bracha and S. Toueg. Distributed deadlock detection. *Distributed Computing*, 1(2) :127–138, 1987.

[BTKS88] K.P. Birman, Joseph T.A., K. Kane, and F. Schmuck. The isis system manuel, June 1988.

[CBMT91] B. Charron-Bost, F. Mattern, and G. Tel. Synchronous and asynchronous communication in distributed computations. Research report 91-55, LITP - Paris 7, September 1991. 33 p.

[CG88] J. C. Comfort and R. R. Gopal. Environment partitioned distributed simulation with transputers. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 103–108, 1988.

[Che87] S. Cheung. Distributed computer simulation of a data communication network. Technical Report CSD-870039, University of California – Los Angeles, July 1987.

[CJS87] I. Cidon, J. Jaffe, and M. Sidi. Local distributed deadlock detection by cycle detection and clustering. *IEEE Transactions on Software Engineering*, SE-13(1) :3–14, 1987.

[CM79] K.M. Chandy and J. Misra. Distributed simulation : a case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, 5(5) :440–452, September 1979.

[CM81] K. M. Chandy and J Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11), April 1981.

[CS89] K. M. Chandy and R. Sherman. Space-time and simulation. In *Proc. ECM'89 (Dist. Simulation Conf.), Tampa, Florida*, May 1989.

[DV90] R.C. De Vries. Reducing null messages in Misra's distributed discrete event simulation. *IEEE Transactions on Software Engineering*, 16(1) :82–91, January 1990.

[Fil92] J.-M. Filloque. Synchronisation répartie sur une machine parallèle à couche logique reconfigurable. Thèse de doctorat, Université de Rennes I, Novembre 1992.

[Fuj87] R.M. Fujimoto. Performance measurement of distributed simulations strategies. Tech. Report UUCS-87-026a, University of Utah, November 1987.

[Fuj88a] R. M. Fujimoto. Lookahead in parallel discrete event simulation. *Proceedings of the 1988 International Conference on Parallel Processing, Pennsylvania*, pages 34–41, August 1988.

[Fuj88b] R.M. Fujimoto. Performance measurements of distributed simulation strategies. In *Proc. SCS conference on Distributed Simulation, San Diego*, pages 14–20, February 1988.

[Fuj90] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33 :31–53, October 1990.

[Gla92] D. W. Glazer. Load balancing parallel discrete-event simulations. PhD. Thesis, McGill University – Canada, May 1992.

[Gre72] S. Greenberg. *A GPSS primer*. John Wiley & sons, 1972.

[GT91] B. Grošelj and C. Tropper. The distributed simulation of clustered processes. *Distributed Computing*, 1(4) :111–121, 1991.

[Had92] K. Hadjar. étude expérimentale d'algorithmes de simulation parallèle utilisant une mémoire commune. Rapport de stage de dea, IRISA–IFSIC, juillet 1992.

[Hei86] P. Heidelberger. Statistical analysis of parallel simulations. In *Proceedings of the Winter Simulation Conference*, 1986.

[Hoa84] C.A.R. Hoare. *Communicating Sequential Processes*. Prenctice-Hall, 1984.

[HR88] J.M. Hélary and M. Raynal. *Synchronisation et contrôle des programmes et des systèmes répartis*. Eyrolles, 1988. 200 p.

[IM92] Ph. Ingels and C. Maziero. Évaluation des performances d'un noyau de simulation répartie. Rapport de Recherche 1751, INRIA, Septembre 1992. 35 p.

[IMR92] Ph. Ingels, C. Maziero, and M. Raynal. A distributed kernel for virtual time driven applications. In *IEEE International Conference on Computers and Information (ICCI 92), Toronto*, pages 430–433, May 1992.

[IMR93] Ph. Ingels, C. Maziero, and M. Raynal. Un noyau réparti pour les applications fondées sur la progression d'un temps virtuel. *Réseau et informatique répartie*, 3(2) :145–168, 1993.

[IR90] Ph. Ingels and M. Raynal. Simulation répartie : schémas d'exécution pour un modèle à processus. *Technique et Science Informatiques*, 9(5) :383–397, 1990.

[J+87] D. Jefferson et al. Distributed simulation and the time-warp operating system. In *11^{th} ACM Symposium on Operating Systems Principles, Austin - Texas*, volume 21, pages 77–93. Operating System Review, ACM Press, November 1987.

[Jef85] D. Jefferson. Virtual time. *ACM Toplas, Vol. 7, No 3*, pages 404–425, July 1985.

[Jon86] D. W. Jones. Concurrent simulation : an alternative to distributed simulation. In *Proceedings of the Winter Simulation Conference*, pages 417–423, 1986.

[JS85] D. Jefferson and H. Sowizral. Fast concurrent simulation using the time warp mechanism. In *Proc. SCS conf. on Distributed Simulation, san Diego*, pages 63–69, January 1985.

[KH90] D. Kumar and S. Harous. An approach towards distributed simulation of timed petri nets. In *Proocedings of the Winter Simulation Conference, New Orleans, LA*, pages 428–435, December 1990.

[KPH76] W.H. Kaubisch, R.H Perrot, and C.A.R. Hoare. Quasiparallel programming. *Software Practice and Experience, vol. 6, no. 3*, pages 341–356, July-September 1976.

[Lam78] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7) :558–565, July 1978.

[Ler80] J. Leroudier. *La simulation à événements discrets.* Monographies de l'AFCET, Editions Hommes et Techniques, 1980. 101 p.

[LG93] J. M. Lin and Abraham S. G. Utilizing global simulation information in conservative parallel simulation on shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, (18) :516–523, 1993.

[LP91] Z. Lahjomri and T. Priol. Koan : a shared virtual memory for the ipsc/2 hypercube. Research report, IRISA, july 1991.

[Lub89] B. Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Communications of the ACM*, 32(1) :111–123, January 1989.

[MÖ88] M. Mühlhäuser. Using distributed simulation for distributed application development. In *The 21st Annual Simulation Symposium, Tampa - Florida*, pages 189–206, March 1988.

[Mat93] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4) :423–434, Aug 1993.

[Meh91] H. Mehl. Speed-up of conservative distributed discret event simulation methods by speculative computing. In *IEEE/ACM/SCS Workshop on parallel and distributed simulation, Anaheim – California*, January 1991.

[Meh92] H. Mehl. A deterministic tie–breaking scheme for sequential and distributed simulation. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS'92), Newport Beach – California*, January 1992.

[Meh93] H. Mehl. Shared variables in distributed simulation. In *IEEE Workshop on Advances in Parallel and Distributed Systems, Princeton, New – Jersey*, October 1993.

[MHC62] H. M. Markowitz, B. Hausner, and H. W. Carr. *Simscript : a simulation programming language.* Prentice Hall, 1962. RAND Corporation, RM3310.

[Mis86] J. Misra. Distributed discrete-event simulation. *Computing Surveys*, 18(1) :39–65, March 1986.

[MOO87] M. Maekawa, A.E. Oldehoeft, and R.R. Oldehoeft. *Operating systems : advanced concepts.* The Benjamin/Cummings Pub. Co, 1987. 414 p.

[MR92] C. Maziero and M. Raynal. Discrete event simulations on distributed memory parallel machines. In *International Workshop on Parallel Algorithms*, Sofia – Bulgaria, August 1992.

[MT91] T. Muntean and E.G. Talbi. Méthodes de placement statique de processus sur architectures parallèles. *Technique et Science Informatiques*, 10(5) :355–374, 1991.

[Pet81] J. L. Peterson. *Petri net theory and the modelling of systems.* Prentice Hall Int., 1981. 290 pp.

[Pid88] M. Pidd. *Computer simulation in management science.* John Wiley & sons, 1988. 307 p.

[Pid89] M. Pidd. *Computer modelling for discrete simulation*. John Wiley & sons, 1989. 274 p.

[Pre90] B. R. Preiss. Performance of discrete event simulation on a multiprocessor using optimistic and conservative synchronization. In *International Conference on Parallel Processing*, pages 218–222, 1990.

[PV84] D. Potier and M. Veran. Qnap2 : a portable environment for queueing systems modelling. In *Int. Conf. on Modelling Technics and Tools for Performance Evaluation*, 1984.

[Ray92] M. Raynal. *Synchronisation et état global dans les systèmes répartis*. Eyrolles, Collection EDF, Janvier 1992. 228 p.

[RJ90] P.L. Reiher and D. Jefferson. Dynamic load management in the time warp operating system. *Transactions of the Society for the Computer Simulation*, 2(7) :91–120, June 1990.

[RM91] H. Rakotoarisoa and Ph. Mussi. Parseval : parallélisation sur réseaux de transputers de simulations pour l'évaluation de performances. Research report RT-131, INRIA, September 1991.

[RST91] M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39 :343–350, 1991.

[RW89] R. Righter and J.C. Walrand. Distributed simulation of discrete event systems. In *Proceedings of the IEEE*, volume 77,1, pages 99–113, January 1989.

[SDC88] S. V. Sheppard, C. K. Davis, and U. Chandra. Parallel simulation environments for multiprocessor architectures. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 109–114, 1988.

[SMP88] B. Samadi, R.R. Muntz, and D.S. Parker. A distributed algorithm to detect a global state of a distributed simulation system. In *IFIG WG 10.3, on Distributed Processing*, pages 19–34. Elsevier Science Publishers B.V, North-Holand, 1988.

[Ste91] S. Steinman. Speedes : Synchronous parallel environment for emulation and discrete event simulation. In *IEEE/ACM/SCS Workshop on parallel and distributed simulation, Anaheim – California*, January 1991.

[Wag91] D. Wagner. Algorithmic optimizations of conservative parallel simulations. In *IEEE/ACM/SCS Workshop on parallel and distributed simulation, Anaheim – California*, January 1991.

[WL89] B. D. Wagner and E. D. Lazowska. Parallel simulation of queueing networks : limitations and potentials. In *Proocedings of the ACM Sigmetrics and Performance*, pages 146–155, May 1989.

[ZGMB84] H. Zimmermann, M. Guillemont, G. Morisset, and J. S. Banino. Chorus, a communication and processing architecture for distributed systems. Research report 328, Inria – Grenoble, septembre 1984.

Annexe : Un programme sur *Floria*

Le code $\mathcal{C}$ ci-dessous présente la modélisation du tore de serveurs *lifo* (figure 4.1) utilisé dans les mesures de performance du noyau (chapitre 4). Ce code est divisé en deux parties : la première définit le corps type de chaque processus et le contenu de chaque type de message ; la deuxième (définie par la fonction `ModelDescription`) réalise la création et la connexion des processus.

```
/*************************************************************/

#include "floria.h"   /* definition de l'interface de Floria */
#include "random.c"   /* generateurs de nombres aleatoires */
#include <stdio.h>

/***** parametres de la simulation ****************************/

#define DIMTORE     16     /* dimension du tore */
#define NUMNODE     8      /* nombre de processeurs */
#define CHARGE      0      /* charge physique de calcul */
#define NBMESSAGE   1      /* nombre de messages par processus */
#define SIMULTIME   500.0  /* duree de la simulation */

/***** definition des types de messages du modele **************/

typedef struct
{       HeaderType Header ; /* tous les messages ont un en-tete */
                        /* ce type de message n'a que l'en-tete */
} CustomerType, *CustomerPtr ;

/***** definition des corps des processus du modele ************/

void Server ()
{
  CustomerPtr Msg ;
  real ServTime, x ;
  int  i, j, Seed;


  /* definition du germe aleatoire du processus */
  /* MyID () rend le numero du processus */
  Seed = 17 * MyID () + 13 ;

  /* envoyer les NBMESSAGE messages initiaux */
  for (i = 1 ; i <= NBMESSAGE; i++)
    {
```

```
      /* creer un nouveau message */
      Msg = (CustomerPtr) MakeMsg (sizeof (CustomerType)) ;
      /* definir le type du message cree */
      SetType ((MsgPtr) Msg, 1) ;
      /* envoyer le message sur une sortie aleatoire */
      SendMsg ((MsgPtr) Msg, INTG (0, 1, &Seed)) ;
    }

  while (MyTime () <= SIMULTIME)
    {
      /* determination du prochain temps de service */
      ServTime = 1.0 + EXP (9.0, &Seed) ;
      /* determination de la prochaine prevision */
      SetLook (0.001 * ServTime) ;

      /* attendre l'arrivee de nouveaux message */
      if (Empty ()) (void) WaitMsg (Infinity) ;
      /* prendre dernier message arrive */
      Msg = (CustomerPtr) Take (Last ()) ;

      /* avancement du temps virtuel */
      Hold (ServTime) ;

      /* charge physique de calcul */
      for (i = 1; i <= CHARGE ; i++)
        for (j = 1; j <= 50; j++)
          x = 3.1416 + j ;

      /* envoyer le message sur une sortie aleatoire */
      SendMsg ((MsgPtr) Msg, INTG (0,1,&Seed)) ;
    }
}
```

```
/***** creation et connexion des processus du modele ************/

void ModelDescription ()
{
  int i, j, p ;

  /* creation des processus du tore */
  for (i = 1; i <= DIMTORE; i++)
    for (j = 1; j <= DIMTORE; j++)
      Create (DIMTORE*(i-1)+j, ((i-1)*NUMNODE)/DIMTORE,
              Server, 10000, 2, 2) ;
```

```
  /* connection des canaux pour former le tore */
  for (i = 1; i <= DIMTORE; i++)
    for (j = 1; j <= DIMTORE; j++)
      {
        p = (i - 1) * DIMTORE + j ;
        Connect (p, 0, (i-1) * DIMTORE + (j % DIMTORE) + 1, 0) ;
        Connect (p, 1, (i % DIMTORE) * DIMTORE + j, 1) ;
      }

  /* duree de la simulation */
  MaxTime  = SIMULTIME ;
}

/*************************************************************/
```