Observation d'exécutions parallèles

Jacques Chassin de Kergommeaux

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Présentation des travaux de recherche pour obtenir le diplôme
d'HABILITATION À DIRIGER DES RECHERCHES
de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

par

Jacques Chassin de Kergommeaux
(Arrêté ministériel du 23 Novembre 1988)

Discipline : **Informatique**

Observation d'exécutions parallèles

Date de soutenance : 19 décembre 2000

Composition du Jury

Rapporteurs :	Jan	Van Campenhout
	Bernard	Lecussan
	Jacques	Mossière
Examinateurs :	Vincent	Olive
	Brigitte	Plateau

Diplôme préparé au Laboratoire Informatique et Distribution
(Institut d'Informatique et de Mathématiques Appliquées de Grenoble)

REMERCIEMENTS

Je remercie tous d'abord les rapporteurs Jan Van Campenhout, Bernard Lecussan et Jacques Mossière, qui ont accepté d'évaluer ce document en dépit de leurs lourdes charges, et plus particulièrement le dernier d'entre eux pour qui il s'agissait d'une récidive.

Je remercie Brigitte Plateau dont la direction éclairée a permis la croissance de notre équipe dans une ambiance sympathique. Ses avis m'ont toujours été précieux.

Je remercie Vincent Olive partenaire ouvert à une collaboration naissante, qui a accepté de participer à ce jury.

Plus que tout autre, un travail d'habilitation manifeste un travail collectif, mené en grande partie par des doctorants et des stagiaires. Le premier d'entre eux fut Alain Fagot qui a accepté de passer de nombreuses nuits devant son terminal pour montrer expérimentalement que la ré-exécution déterministe ne coûtait pas cher. Benhur Stein a accompli avec bonne humeur et discrétion un travail considérable en quantité et qualité et dont les prolongements nous permettent de poursuivre notre coopération. Parmi les stagiaires de DEA je voudrais mentionner particulièrement Florin Teodorescu qui, bien que cela n'ait pas constitué son sujet de stage, a été le premier à visualiser l'exécution de *threads* ; Youri Vassilief, Denise Stringhini et Anne Benoit ont également apporté une contribution importante à ces travaux.

Certaines collaborations internationales ont permis des avancées importantes et tout particulièrement la coopération avec l'Université de Gand. Koen De Bosschere s'est toujours montré très disponible pour monter des projets et faire partager ses compétences. Michiel Ronsse a résolu rapidement et efficacement un problème pour lequel nous n'avions pas de bonne solution. *Dank je wel ! Dat deed me genoegen om met je te werken, Michiel.*

Plus informelle mais toute aussi réelle a été l'aide des autres membres de l'équipe devenue maintenant laboratoire : travail collectif dans le cadre du projet APACHE, explication d'un problème mal compris, aide à la résolution d'une difficulté d'ordre technique, LaTex, système, administratif, etc. Merci à vous tous, ce travail vous doit beaucoup.

TABLE DES MATIÈRES

1

INTRODUCTION

Apache :

– n.m. 1902 ; de *Apaches*, Indiens du Texas, réputés féroces.
– *Vieilli*. Malfaiteur, voyou de grande ville prêt à tous les mauvais coups.
Petit Robert, 1981.

– *Any of the Athapascan languages of the Apache.*
– *A member of the Parisian underworld.*
American Heritage Dictionary, 1986.

Mon activité de recherche a pour cadre le développement d'environnements de programmation et d'outils logiciels pour faciliter l'utilisation des systèmes parallèles. Dans ce contexte, l'essentiel de mon activité a été consacrée aux outils d'**aide** à la mise au point de programmes parallèles, tant d'un point de vue optimisation de performances que pour l'aide à la détection d'erreurs. Il ne s'agit pas de trouver des erreurs de logique ou de performances automatiquement mais d'aider les programmeurs à détecter ces erreurs. La perspective qui est adoptée est de donner aux programmeurs les moyens d'observer aussi facilement, précisément, exactement que possible les exécutions parallèles. Ces travaux se sont développés dans le contexte du projet Apache[1].

[1]APACHE est un acronyme pour « Algorithmique Parallèle et PArtage de CHargE ». C'est

1.1 CONTEXTE DE LA RECHERCHE

Les architectures parallèles se sont imposées dans le domaine du calcul hautes performances. De nombreuses architectures ont été proposées, pour tirer parti de progrès technologiques ou optimiser l'exécution de langages parallèles. En liaison avec ces nouvelles architectures, des constructeurs de machines parallèles sont apparus puis ont disparu, souvent victimes d'évolutions technologiques rendant obsolètes leurs choix fondateurs. L'évolution des architectures parallèles semble obéir à une tendance de fond qui est la progression continue de l'utilisation de composants standard. Cette évolution a tout d'abord touché les processeurs et, depuis plusieurs générations déjà, la plupart des machines parallèles utilisent des micro-processeurs produits en grande série. La raison réside bien évidemment dans la progression exponentielle des performances de ces processeurs, cette dernière découlant des investissements considérables qui leur sont consentis en raison de l'importance des marchés qu'ils représentent : les performances des micro-processeurs de série sont maintenant très similaires à celles des processeurs spécialisés, lesquels sont beaucoup plus coûteux. L'évolution vers des composants standard touche maintenant les réseaux d'interconnexion. En raison du développement de ce marché, la progression des performances des réseaux produits en série s'accélère de façon spectaculaire ce qui permet leur utilisation dans les architectures parallèles.

Depuis quelques années, on voit donc apparaître des architectures hautes performances à bas prix appelées « grappes », construites uniquement à partir de composants matériels de série (*off the shelf*). Les nœuds sont des ordinateurs personnels (PC) de série, comportant un nombre modéré (de un à quatre habituellement) de processeurs de grande série, connectés par un bus et une mémoire commune. Le réseau d'interconnexion est également un réseau de série. L'utilisation de composants de série n'implique cependant pas l'uniformité : on peut trouver des grappes variées en performances et en prix, différant par la puissance des processeurs utilisés (32 ou 64 bits par exemple), ou les performances du réseau de communication (Fast Ethernet, Myrinet, SCI, etc.).

Le projet Apache a suivi, s'il ne l'a pas anticipé, cette évolution des architectures parallèles. L'environnement ATHAPASCAN[2], développé dans le projet Apache, vise à faciliter la programmation efficace et portable de telles architectures.

ATHAPASCAN se décompose en deux niveaux :

un projet CNRS-INPG-INRIA-UJF.

[2]Nom de la langue parlée par les indiens Apaches

– Un noyau exécutif, ATHAPASCAN-0, à base de fils d'exécution communicants [92]. ATHAPASCAN-0 exploite deux niveaux de parallélisme : parallélisme entre nœuds, mis en œuvre par un nombre fixe de processus «systèmes» et parallélisme interne à chacun des nœuds, exploité par un réseau de fils d'exécution (*threads*) évoluant dynamiquement [73].
– Une interface de programmation parallèle, ATHAPASCAN-1, basée sur un modèle de parallélisme de tâches avec cohérence de données et permettant une répartition automatique de la charge de calcul [100].

ATHAPASCAN comporte également des outils d'aide à la mise au point de programmes parallèles dont la présentation fait l'objet de ce document.

En plus du développement de ATHAPASCAN, une part importante de l'activité du projet Apache est consacrée au développement d'applications parallèles. Celles-ci ont le double rôle de générer des problèmes et de tester les solutions proposées par le projet. Les applications étudiées concernent des domaines variés qui vont de la dynamique moléculaire [60] à l'algèbre linéaire [63].

1.2 (DIFFICULTÉS DE LA) MISE AU POINT DE PROGRAMMES PARALLÈLES

Régulièrement des études mettent l'accent sur la difficulté d'utilisation des systèmes et machines parallèles et sur l'absence d'outils adéquats pour faciliter l'utilisation de ces systèmes [90], et en particulier le besoin d'outils de mise au point [91]. Par mise au point on entend d'une part élimination des erreurs de logique ou d'implantation (*correctness debugging*) — qui font qu'un programme parallèle ne produit pas le résultat escompté — et d'autre part élimination des erreurs de performances (*performance debugging*) — qui empêchent un programme de tirer parti de la puissance de l'architecture parallèle utilisée. L'élimination des « erreurs de performance » est considérée comme aussi importante que celle des erreurs de logique, pour plusieurs raisons :

– Les systèmes parallèles sont utilisés comme architectures hautes performances dont le prix dépasse notablement celui des postes de travail individuels.
– Il est souvent difficile d'approcher les performances théoriques des architectures parallèles.

La perception d'un besoin d'outils de mise au point ne signifie pas l'absence de recherche dans ce domaine : une bibliographie sur les débogueurs parallèles, publiée en 1993, comportait 293 références [114]. Une version plus récente de cette bibliographie comportait 659 références en 1994 [130] et si cette biblio-

graphie ne semble pas avoir été mise à jour depuis, ce n'est pas faute de nouvelles publications sur ce thème[3] ! Dans une synthèse bibliographique datant de 1989 [83], C.E. McDowell et D.P. Helmbold remarquent tout d'abord que les programmes parallèles ne permettent pas toujours d'utiliser la technique du débogage cyclique,utilisée couramment pour le débogage des programmes séquentiels. En effet de nombreux programmes parallèles n'ont pas de comportement reproductible en raison de conditions de concurrence (*races*, voir chapitre 2). McDowell et Helmbold classent les techniques de débogage de systèmes concurrents en quatre groupes :

1. Extension aux programmes parallèles d'outils de débogage traditionnels. Chaque débogueur de ce type se compose fonctionnellement d'une collection de débogueurs « séquentiels », à raison de un par nœud du système parallèle. C'est le cas de la plupart des débogueurs commerciaux dont le produit le plus abouti à l'heure actuelle est probablement TotalView [121]. La principale différence relativement aux débogueurs séquentiels consiste en la notion de *contexte* : les fonctionnalités héritées des débogueurs séquentiels n'affectent en effet que le ou les processus ou fils d'exécution qui font partie du contexte courant.

   Ces débogueurs donnent aux programmeurs la possibilité de contrôler le déroulement d'un programme parallèle : pose de points d'arrêt sur un ou un ensemble de processus, exécution pas à pas d'un processus, etc. Ils offrent également la possibilité d'observer l'état de la pile de chacun des processus ou des fils d'exécution ainsi que les valeurs des variables, la visualisation se faisant à l'aide de systèmes multifenêtrages.

   Bien qu'indispensables à la mise au point des applications parallèles, les débogueurs parallèles « classiques » sont cependant insuffisants pour lutter contre les erreurs fugitives : pour ce faire ils doivent être utilisés conjointement à des outils de ré-exécution déterministe (voir ci-dessous et chapitre 2). Si nos travaux ne portent pas directement sur les outils de débogage parallèle traditionnels, nous nous sommes néanmoins intéressés à eux à travers deux projets menés en coopération avec des universités étrangères (voir §4.3). Le premier, mené en coopération avec l'Universidade Nova de Lisboa, a pour objectif d'adapter un débogueur symbolique distribué PDBG à nos besoins. Dans le cadre du second, mené en coopération avec l'Universidade Federal do Rio Grande do Sul à Porto Alegre, est développée une interface graphique au débogueur parallèle précité ainsi que des mécanismes sophistiqués pour focaliser l'observation sur les fils d'exécution les plus intéressants.

[3] La cause probable de cette contradiction est abordée ultérieurement (voir §4.4).

2. Débogueurs basés sur les événements et pour lesquels l'exécution d'un programme parallèle peut être considérée comme une ou plusieurs séquences parallèles d'événements. Les outils de débogage de cette sorte diffèrent selon les types d'événements considérés et les traitements qui leur sont appliqués. Une partie importante des travaux présentés dans ce manuscrit concernent l'enregistrement d'événements permettant de ré-exécuter les programmes de façon déterministe (voir chapitre 2). Nous nous intéresserons également à l'enregistrement d'événements pour la mesure de performances de programmes non déterministes (voir §2.5). En revanche, nous ne nous intéresserons pas aux techniques permettant de tester `post mortem` [103] ou en ligne [99], à partir de traces d'exécution, qu'une exécution donnée d'un programme parallèle satisfait ou non une propriété (prédicat).
3. Techniques graphiques présentant aux utilisateurs le contrôle et les données des programmes parallèles afin d'en faciliter la compréhension. Les techniques de visualisation d'exécutions parallèles seront abordées dans ce document dans le cadre des outils de mise au point pour les performances (voir chapitre 3). Leur utilisation pour le débogage est l'un des objectifs des études en cours présentées au chapitre 4.
4. Techniques d'analyse statique des programmes parallèles pour détecter certaines classes d'anomalies dans les programmes parallèles. McDowell et Helmbold distinguent particulièrement les erreurs de synchronisation, susceptibles de provoquer des erreurs telles que l'interblocage (*deadlock*) et les conditions de concurrence (*races*) sur les données. Plus récemment, Helmbold et McDowell font le point sur les nombreux travaux consacrés aux méthodes de détection des conditions de concurrence [89] dans les programmes parallèles. Malgré l'existence de solutions exactes ou approchées pour certains modèles de programmation ou de communication, ils concluent que le problème général de la détection de conditions de concurrence est insoluble.

1.3 RECHERCHES DÉVELOPPÉES

Comme nous l'avons indiqué ci-dessus, les travaux présentés dans ce document poursuivent principalement deux objectifs : permettre le débogage cyclique des applications parallèles et en faciliter la compréhension en utilisant des techniques de visualisation. Ces deux axes peuvent être fédérés en un objectif unique plus général qui est celui de faciliter l'observation des exécutions parallèles, rendu

difficile par le comportement non reproductible des exécutions d'une part et par leur complexité d'autre part. La perspective adoptée est d'offrir des outils d'aide à la mise au point plutôt que de détection automatique de telle ou telle catégorie d'erreur. La raison est que les outils automatiques ne peuvent détecter que les erreurs répertoriées et prévisibles et que nous pensons qu'il existera toujours un grand nombre d'erreurs échappant aux schémas pré-établis. Les outils que nous avons développés, complémentaires des outils de débogage traditionnels mentionnés plus haut, ont pour objectif d'aider à la mise en évidence des erreurs quelles qu'elles soient.

Plus précisément, l'objet principal de l'activité de recherche présentée dans ce document concerne l'observation des exécutions de programmes parallèles à base de fils d'exécution communicants, dans le but de faciliter leur mise au point. Relativement aux travaux existants, nous nous sommes principalement intéressés aux problèmes introduits par la dynamicité du modèle de programmation. À la différence des outils existants, qui supposent que les exécutions de programmes mettent en œuvre un nombre fixe de processus — généralement un par nœud du système utilisé —, nous nous intéressons à des exécutions comportant un nombre éventuellement important de fils d'exécution évoluant dynamiquement (création, terminaison) et communiquant soit par mémoire partagée (verrous, variables condition ou sémaphores) soit par passage de messages.

Le choix qui sous-tend ce travail est de laisser au programmeur la maîtrise de l'activité de mise au point : on suppose que s'il est capable d'observer précisément l'exécution de son programme, il sera à même d'identifier l'origine des erreurs de logique et de performances. Les travaux qui sont présentés dans les chapitres qui suivent visent à résoudre deux des problèmes que pose l'observation de programmes parallèles : la sensibilité à l'indéterminisme d'une part et la complexité provenant du grand nombre d'objets mis en œuvre d'autre part. Le dernier chapitre avant la conclusion brosse une perspective à ces travaux, consistant à intégrer les solutions proposées dans un environnement de mise au point.

1.3.1 MAÎTRISE DE L'INDÉTERMINISME

Bien que considérée comme peu souhaitable en raison des difficultés qu'elle induit pour la mise au point des applications [95], l'écriture de programmes indéterministes est rendue possible par de nombreux modèles de programmation parallèle comme par exemples ceux qui sont induits par les bibliothèques MPI [72] et PVM [88]. Ces possibilités peuvent être utilisées pour implanter certains modèles d'algorithmes — fermier et travailleurs (*process farm*) par exemple — ou pour adapter des programmes aux conditions d'exécution, en y incorporant des mécanismes de régulation dynamique de la charge. Il est tout à fait possible

d'écrire des programmes non déterministes dont les résultats sont déterministes. Ce non déterminisme « interne » des applications en interdit la mise au point « cyclique », qui consiste à ré-exécuter les applications autant de fois qu'il est nécessaire pour en identifier les erreurs. En effet des erreurs « fugitives » peuvent apparaître épisodiquement dans des exécutions non déterministes, ou encore une erreur peut disparaître dès qu'un outil de mise au point — débogueur ou traceur par exemple — est mis en œuvre.

La ré-exécution déterministe est la technique classiquement utilisée pour permettre la mise au point cyclique d'applications parallèles (ou distribuées) non déterministes. Le travail qui a été fait dans ce domaine consiste essentiellement en une adaptation efficace de solutions existantes au modèle de programmation parallèle à base de fils d'exécution communicants. D'autres études complémentaires ont établi la possibilité d'utiliser ces techniques comme base d'un environnement de mise au point, tant dans le domaine de la correction des erreurs de logique (*correctness debugging*) que des performances (*performance debugging*).

Ces travaux ont fait l'objet de la thèse de Alain Fagot [62] et des stages de DEA de Emmanuelle Gay [122], Florin Teodorescu [127] et Youri Vassilieff [128] que j'ai encadrés. Ils ont donné lieu à deux publications dans des revues internationales [10, 12], huit publications dans des conférences internationales [29, 32, 33, 34, 39, 40, 44, 45] et deux publications dans des conférences nationales [28, 98]. Une partie importante de ce travail a été effectuée en collaboration avec Michiel Ronsse et Koen De Bosschere de l'Université de Gand en Belgique, dans le cadre d'un projet du programme d'actions intégrées Tournesol.

1.3.2 Visualisation d'exécutions parallèles

Les outils développés pour aider les programmeurs à identifier leurs erreurs de performance sont basés sur le traçage logiciel d'applications parallèles et la visualisation `post mortem` des exécutions à partir de ces traces. Cette technique semble la plus appropriée pour identifier les surcoûts dus aux communications ainsi que les périodes d'oisiveté des processeurs, provenant par exemple d'un mauvais recouvrement entre calculs et communications [17, 125].

La recherche menée dans ce domaine a conduit à la réalisation de l'outil Pajé. Pajé offre non seulement des solutions permettant la visualisation de programmes parallèles à base de fils d'exécution communicants, mais de plus permet à l'utilisateur de visualiser d'autres modèles de programmation, en lui laissant la possibilité de spécifier simplement comment cette visualisation doit être faite. Cette possibilité permet d'envisager son utilisation comme outil de visualisation scientifique pour une grande variété de domaines : elle a pour l'instant été utilisée pour visualiser l'exécution de programmes ATHAPASCAN-1, sans aucune modification à

Pajé ; son utilisation pour visualiser des exécutions de programmes Java [54] ou aider à l'administration de grappes de grande taille est actuellement en cours de test. Pajé présente également un degré d'interactivité peu courant sur ce type d'outil puisqu'il est possible de naviguer dans le temps ou encore de faire la liaison entre les visualisations et le code source des applications.

Ces travaux ont fait l'objet de la thèse de Benhur Stein [67] que j'ai encadrée. Ils ont fait l'objet d'une publication dans un journal international [9], de deux publications dans des conférences internationales [27, 42] et d'une publication dans une conférence nationale [41]. Le transfert de Pajé est actuellement en cours de discussion avec divers partenaires.

1.3.3 VERS UN ENVIRONNEMENT INTÉGRÉ DE MISE AU POINT

L'objectif est d'intégrer tous les outils de mise au point dans un environnement homogène. Les bénéfices escomptés sont d'une part de pouvoir effectuer une mise au point cyclique des applications parallèles non déterministes à base de fils d'exécution communicants et d'autre part de bénéficier de l'outil de visualisation Pajé dans toutes les phases de la mise au point. Cette intégration ne semble pas poser de problème majeur mais elle implique de transformer les outils de traçage et visualisation afin de pouvoir faire de la visualisation en ligne. Il sera en outre nécessaire de synchroniser la visualisation avec les outils de débogage.

Ce travail fait l'objet de la « thèse sandwich » de Denise Stringhini de l'Universidade Federal do Rio Grande do Sul (UFRGS) à Porto Alegre (Brésil), dans le cadre du projet de coopération INRIA-CNPq PAGE. Le stage de DESS de Frédéric Ribette [126] lui était également consacré. Il a donné lieu à une publication dans une conférence internationale [43] et à un article dans une École du CNRS [16]. L'intégration d'outils de mise au point est également au coeur du projet de coopération INRIA-ICCTI mené avec l'Universidade Nova de Lisboa depuis 1998.

2

MAÎTRISE DE L'INDÉTERMINISME

« Il n'y a pas de lois dans l'indéterminisme. »
Claude Bernard

2.1 INTRODUCTION

2.1.1 INFLUENCE DE L'INDÉTERMINISME SUR LE DÉBOGAGE PARALLÈLE

Un grand nombre de programmes parallèles présentent un comportement indéterministe. On peut distinguer l'indéterminisme « externe », qui apparaît lorsqu'un programme retourne des résultats différents à des exécutions successives avec les mêmes paramètres en entrée. Ce type d'indéterminisme résulte souvent, mais pas nécessairement d'une erreur. L'autre type d'indéterminisme « interne » apparaît lorsque des exécutions répétées d'un même programme avec les mêmes paramètres donnent les mêmes résultats mais que les chemins d'exécution internes sont différents. L'indéterminisme interne est programmé volontairement dans les

applications qui s'adaptent dynamiquement aux données en entrées ou à l'environnement d'exécution. C'est donc le cas de programmes utilisant certains schémas algorithmiques comme le « maître-esclave » (*process farm*) ou implémentant une certaine forme de régulation dynamique de charge.

L'indéterminisme est rendu possible lorsque le modèle de programmation permet l'existence de conditions de concurrence (*races*) pour l'accès à la mémoire partagée ou entre messages reçus (voir figure 2.1). Certaines de ces conditions de concurrence sont contrôlées par le programmeur : accès à un objet de synchronisation pour des fils d'exécution partageant le même espace d'adressage ; réception de messages « conflictuels » pour un fil d'exécution jouant un rôle de serveur (voir §2.4.4). En revanche, des conditions de concurrence sur les données (*data races*), sans contrôle par un objet de synchronisation, résultent le plus souvent d'une erreur de programmation.

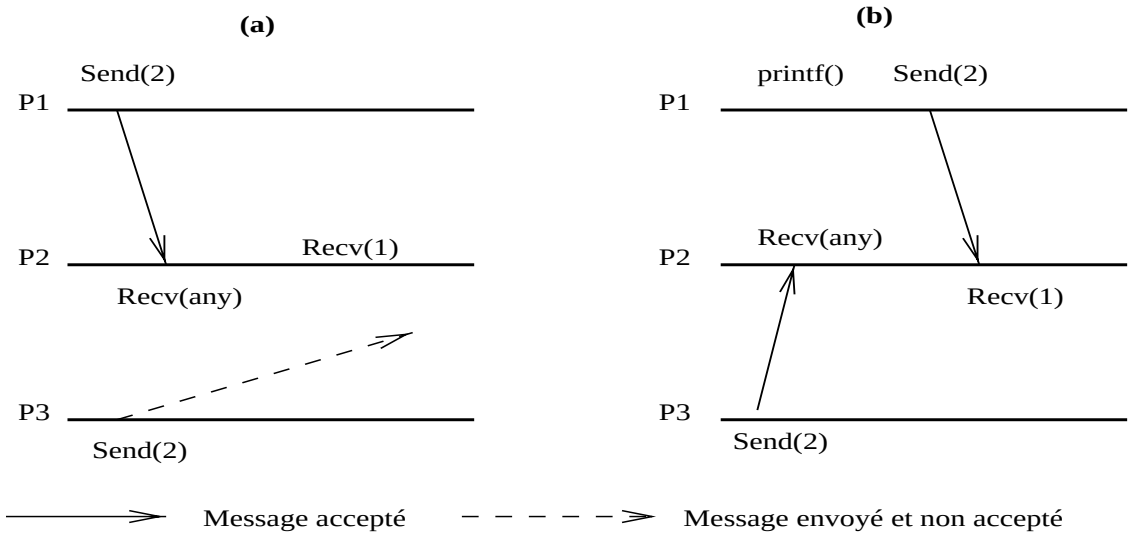

FIG. 2.1 – Conditions de concurrence entre messages reçus

L'erreur de programmation qui apparaît dans l'exécution (a) est masquée dans l'exécution (b) par la perturbation causée par l'ajout d'un ordre de traçage (*printf*).

À partir de paramètres initiaux semblables, un comportement indéterministe peut alors être provoqué par un grand nombre de facteurs, incontrôlables par le programmeur d'une application indéterministe : les contenus initiaux des caches mémoires, la charge du réseau de communication, l'état initial du système d'exploitation, l'exécution simultanée d'autres processus que ceux de l'application, etc. Le comportement non déterministe d'un programme peut être à l'origine du caractère fugitif des erreurs, qui n'apparaissent pas dans toutes les exécutions ou disparaissent dès qu'une instruction de traçage est insérée ou qu'un metteur au

point parallèle est utilisé (voir figure 2.1). Ce type de comportement rend impossible la mise au point cyclique utilisée habituellement pour les programmes séquentiels, où l'on ré-exécute le programme fautif autant de fois qu'il est nécessaire pour localiser une erreur. Chaque cycle de mise au point comporte habituellement quatre phases : extraction, analyse d'informations, présentation des résultats d'analyse et modification de l'application. Les débogueurs symboliques tels que `gdb` [59] facilitent les trois premières phases, permettant de focaliser la recherche de plus en plus près de l'origine de l'erreur, au fur et à mesure du déroulement des cycles.

2.1.2 Influence de l'indéterminisme sur la mesure de performances

La mesure de performances des programmes parallèles en perturbe également le comportement. Cette mesure est faite fréquemment par des traceurs logiciels, dont l'intrusion peut éventuellement être compensée par un traitement `post mortem` des traces [17, 65, 81]. Cependant, les algorithmes de compensation de l'intrusion ne sont pas applicables aux programmes dont l'observation a changé la causalité. Cette situation est analogue à celle de la figure 2.1, dans laquelle les deux messages seraient des `Recv(any)` mais où la prise de trace, en ayant le même effet que le `printf` de la figure, inverserait l'ordre d'arrivée des messages dans l'exécution tracée, relativement à celui de l'exécution non tracée.

2.1.3 Travaux développés

Puisqu'il n'est ni possible ni souhaitable de supprimer l'indéterminisme des programmes parallèles, nos travaux ont porté sur la mise en œuvre de techniques permettant d'en limiter les conséquences négatives sur l'observation des exécutions de programmes parallèles. Il s'agit d'une part de permettre le débogage cyclique de programmes, en adaptant les techniques de ré-exécution déterministe aux programmes à base de fils d'exécution communicants. La ré-exécution déterministe a tout d'abord été adaptée efficacement au modèle « procédural parallèle », qui correspond au modèle de programmation initial du projet Apache, ATHAPASCAN-0a [61]. Le mécanisme de ré-exécution déterministe a servi de base à plusieurs études concernant entre autres le calcul d'état global (*snapshot*) ou la correction de l'intrusion due à la mesure de performances par enregistrement de traces. Enfin un mécanisme de ré-exécution déterministe pour ATHAPASCAN-0 [92] a été réalisé, combinant des techniques optimisées existantes pour reproduire

l'ordre d'accès aux objets de synchronisation avec des techniques originales efficaces pour ré-exécuter des primitives de communication non bloquantes.

2.2 RÉ-EXÉCUTION DÉTERMINISTE DE PROGRAMMES PARALLÈLES

La technique la plus classiquement utilisée pour détecter les erreurs fugitives apparaissant durant les exécutions des programmes parallèles consiste à **enregistrer** les choix non déterministes d'une exécution initiale puis à utiliser les informations enregistrées pour **forcer** les exécutions suivantes à se comporter de façon déterministe, relativement à cette exécution initiale. Le débogage d'un programme erroné revient alors à enregistrer une exécution erronée puis à appliquer les techniques de mise au point cycliques durant les ré-exécutions successives du programme. Pour que cette technique soit utilisable en pratique, il est d'une importance capitale que la perturbation causée par l'enregistrement des traces demeure suffisamment faible pour que les erreurs apparaissant durant des exécutions non enregistrées ne disparaissent pas durant les exécutions « enregistrées » et vice-versa ; il est également nécessaire que le surcoût en temps des mécanismes de ré-exécution demeure tolérable ; enfin il faut que le volume des informations enregistrées soit raisonnable. On peut remarquer que surcoût en temps et en espace sont souvent liés puisqu'un algorithme sophistiqué de compression de données est susceptible d'augmenter le surcoût en temps. Si le surcoût dû à l'enregistrement est suffisamment faible, il est possible de laisser l'enregistrement actif durant chaque exécution de programme parallèle, en sorte que toute erreur, même peu fréquente, puisse être enregistrée et reproduite à volonté.

Le principe de la ré-exécution déterministe est fondé sur le comportement déterministe de chaque processus pris individuellement : s'il reçoit les *mêmes* entrées dans le *même* ordre, un processus produira les *mêmes* sorties dans le *même* ordre, … qui seront utilisées comme entrées d'autres processus. Deux approches ont été développées pour permettre la ré-exécution déterministe des programmes parallèles : la première, basée sur les **données**, enregistre le contenu de toutes les entrées des processus mis en œuvre par les applications parallèles ; la seconde, basée sur le **contrôle**, enregistre l'ordre de événements en condition de concurrence et impose le même ordre durant les ré-exécutions. La ré-exécution dirigée par les données n'est pas considérée comme utilisable en pratique en raison du volume important de données qu'elle nécessite d'enregistrer [113]. Dans ce qui suit, nous nous intéresserons essentiellement aux techniques de ré-exécution déterministe

basées sur le contrôle[1].

2.2.1 CADRE FORMEL

Il est possible de définir formellement la notion d'équivalence entre deux exécutions d'un programme parallèle et d'établir des conditions suffisantes pour l'équivalence d'exécutions, permettant d'en déduire des mécanismes pratiques de ré-exécution déterministe. Intuitivement, deux exécutions X et Y d'un même programme P seront considérées comme équivalentes si chaque processus de P passe par la même suite d'états lors de deux exécutions. Deux exécutions équivalentes produiront donc le même résultat ou exhiberont la même erreur [64]. Équivalentes ne signifie pas nécessairement identiques : dans les modèles de ré-exécution déterministe existants, l'ordre relatif des exécutions des instructions élémentaires dans les processeurs n'a aucune importance.

La première étape est la modélisation des exécutions de programmes parallèles. À partir d'un modèle il est ensuite possible de définir la notion d'équivalence entre exécutions. L'étape la plus importante est alors de montrer que le mécanisme utilisé — enregistrement d'informations de contrôle et utilisation de ces informations pour forcer le même ordre d'accès aux objets de synchronisation et le même ordre de réception de messages — *suffit* à assurer l'équivalence entre l'exécution « enregistrée » et les exécutions ré-exécutées. Ce travail de modélisation a été fait pour les modèles de programmation à mémoire partagée [66, 108] et par passage de messages [102, 106, 111].

Le modèle le plus complet est celui présenté par É. Leu dans sa thèse [64]. Une exécution X est modélisée par un état initial S_0^X, l'ensemble E^X des événements élémentaires de X et une relation R^X d'ordre partiel entre les événements de X appelée relation de précédence directe.

[1] Tout en remarquant qu'il est de toutes façons nécessaire d'enregistrer certaines données telles que les résultats des primitives non déterministes comme *random* ou encore la lecture de l'horloge physique.

Définitions 2.2.1

*Un **événement élémentaire de calcul** est défini comme un ensemble d'instructions ne comprenant aucun accès à la mémoire partagée, aucun envoi et aucune réception de message.*
*Un **événement élémentaire de communication** est défini comme l'une (et une seule) des trois actions suivantes : envoi de message, réception de message ou accès à une unité de mémoire partagée.*
Un niveau de granularité plus grossier des événements est envisageable dans le contexte de la mémoire partagée, si des protocoles assurant une linéarisation des accès (CREW[a] par exemple) sont utilisés.
*Une **relation de précédence directe** R existe entre deux événements $e_{i,j}$ et $e_{k,l}$ si et seulement si l'une au moins des trois conditions suivantes est satisfaite :*

1. *$e_{i,j}$ et $e_{k,l}$ sont des événements consécutifs d'un même processus : $e_{k,l} = e_{i,j+1}$;*
2. *$e_{i,j}$ est l'émission d'un message et $e_{k,l}$ sa réception ;*
3. *$e_{i,j}$ et $e_{k,l}$ sont deux accès consécutifs à une même unité de mémoire partagée m.*

[a] Concurrent Read Exclusive Write

Définition 2.2.2

*Deux exécutions $X = \langle S_0^X, E^X, R^X \rangle$ et $Y = \langle S_0^Y, E^Y, R^Y \rangle$ sont **équivalentes** si et seulement si :*

$$S_0^X = S_0^Y \quad E^X = E^Y \quad R^X = R^Y$$

Leu démontre ensuite que si deux exécutions sont équivalentes, alors la suite des états par lesquels passent les deux exécutions est identique. C'est ce qui importe aux programmeurs qui veulent faire du débogage cyclique. Ce cadre théorique permet de démontrer un théorème donnant des conditions suffisantes pour que deux exécutions de programmes soient équivalentes. Ce théorème suppose que les deux exécutions considérées ont le même état initial et que les événements observés dans chacune des deux exécutions sont tous élémentaires.

Théorème 2.2.3

Sous ces hypothèses, deux exécutions d'un programme parallèle sont équivalentes si l'ordre de réception des messages par chacun des processus et l'ordre partiel d'accès aux cellules de mémoire partagée sont identiques dans les deux exécutions.

2.2.2 SYSTÈMES EXISTANTS

Ce résultat est utilisé pour implémenter la plupart des mécanismes de ré-exécution déterministe existants. La plupart des modèles de programmation parallèles offrent des communications par passage de messages *ou* par mémoire partagée mais rarement les deux simultanément. Pour les modèles de programmation basés sur le partage de mémoire, il est théoriquement nécessaire d'enregistrer tous les accès aux variables partagées, ce qui a un coût potentiellement considérable. Sur monoprocesseur ont été développés des mécanismes moins coûteux pour la ré-exécution déterministe de programmes concurrents, en enregistrant l'ordonnancement des fils d'exécution puis en forçant le même ordonnancement durant les ré-exécutions [117].

Pour les multiprocesseurs, le surcoût dû à l'enregistrement peut être limité si on utilise un protocole assurant la linéarisation des accès à la mémoire partagée [79]. Le plus souvent, on fait l'hypothèse que les programmes ne font pas d'accès non synchronisés à la mémoire partagée. Cette hypothèse permet de réduire considérablement le nombre d'enregistrements nécessaires à la ré-exécution déterministe et il n'est plus nécessaire d'enregistrer que l'ordre d'accès aux objets de synchronisation : verrous, variable condition, sémaphore, etc. Durant les ré-exécutions, on force les processus à accéder à ces objets de synchronisation dans le même ordre que durant l'exécution enregistrée. L'utilisation d'horloges logiques permet de réduire encore de façon très importante le nombre d'enregistrements nécessaires à la ré-exécution, en n'enregistrant que les synchronisations donnant effectivement lieu à une condition de concurrence [79, 87] (voir §2.4.3).

Les mécanismes de ré-exécution déterministe pour les modèles de programmation par passage de message enregistrent, durant une exécution initiale, l'ordre d'arrivée des messages concurrents, puis durant les ré-exécutions déterministes forcent le même ordre de réception de ces messages par les processus [102, 106, 111]. Là encore, l'utilisation d'horloges logiques (vectorielles) permet de réduire considérablement le nombre d'enregistrements [111, 112].

Les ré-exécutions des programmes parallèles sont équivalentes aux exécutions enregistrées, pourvu que ces ré-exécutions remplissent les hypothèses du théo-

rème d'équivalence — même état initial, pas d'utilisation de fonctions telles que *random* — et, dans la plupart des cas, qu'elles ne fassent pas d'accès non synchronisé à la mémoire partagée. Cependant, étant donné que la ré-exécution d'un programme comportant des condition de concurrence sur les données est déterministe jusqu'à la première d'entre elles, on peut utiliser la ré-exécution déterministe pour la détecter, et ce autant de fois qu'il est nécessaire pour la corriger et passer à la détection des suivantes. Les conditions de concurrence sur les données peuvent également être détectées automatiquement durant les ré-exécutions, en utilisant des outils appropriés [115].

2.3 RÉ-EXÉCUTION DÉTERMINISTE POUR LE MODÈLE DE PROGRAMMATION PROCÉDURAL PARALLÈLE

L'apport principal de ce travail est l'utilisation du niveau d'abstraction du modèle de programmation pour réduire le volume des traces nécessaires à la ré-exécution déterministe ; en outre, le mécanisme ainsi défini est indépendant de la couche de communication sous-jacente et peut donc être facilement porté.

2.3.1 MODÈLE DE PROGRAMMATION PROCÉDURAL PARALLÈLE

Le paradigme de l'appel procédural parallèle dérive du *Remote Procedure Call* (RPC) de Birrel et Nelson [68], où chaque fonction appelée est exécutée par un fil d'exécution spécifique, spécialement créé pour traiter cet appel (voir figure 2.2). Ce fil d'exécution peut être lancé localement aussi bien qu'à distance. La durée de vie du fil d'exécution est limitée à la durée d'exécution de la fonction appelée. Les appels peuvent être synchrones ou asynchrones, c'est-à-dire qu'ils peuvent bloquer ou non l'exécution du fil d'exécution appelant (voir figure 2.3).

Dans le modèle procédural parallèle de base, il n'est pas permis de faire des effets de bord en modifiant des variables globales à plusieurs fils d'exécution : les seules communications entre fils d'exécution sont le transfert de paramètres et de résultats. Seules les procédures encapsulées dans des *points d'entrées* sont susceptibles de faire l'objet d'un appel à distance. Chaque point d'entrée est local à un processeur virtuel. Le *degré de concurrence* de chaque point d'entrée peut être borné pour limiter le nombre de fils d'exécution actifs simultanément exécutant la procédure encapsulée, ce mécanisme étant utilisé pour la synchronisation de fils d'exécution. Le modèle procédural parallèle a été mis en œuvre dans ATHAPASCAN-0a, maquette préliminaire du noyau exécutif ATHAPASCAN-0 [96, 61].

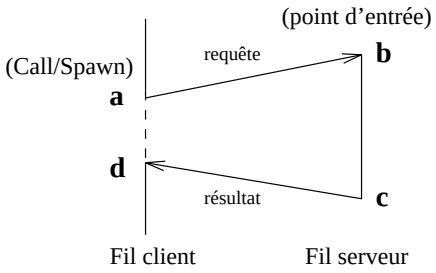

FIG. 2.2 – Séquence d'événements mis en oeuvre lors d'un appel à un point d'entrée

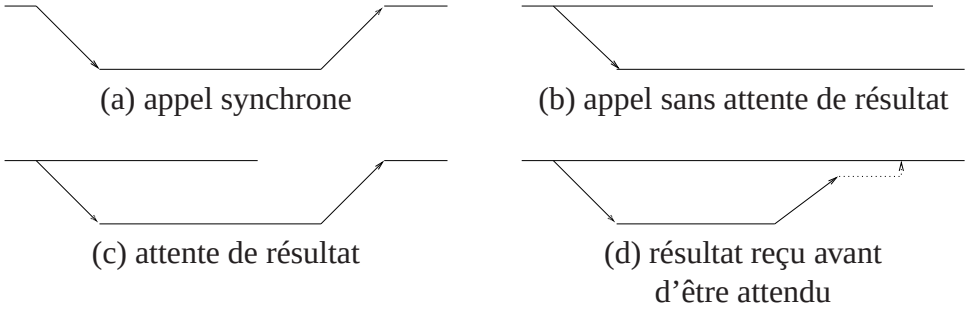

FIG. 2.3 – Appels de procédure à distance synchrones et asynchrones

2.3.2 PRINCIPE DE LA RÉ-EXÉCUTION DÉTERMINISTE

Le modèle procédural parallèle permet l'écriture de programmes comportant des conditions de concurrence pour l'appel d'une procédure à distance sur un point d'entrée. Deux appels, émis par des fils d'exécution distincts en direction d'un même point d'entrée, sont susceptibles de ne pas être traités selon le même ordre dans toutes les exécutions d'un programme donné.

Le mécanisme de ré-exécution déterministe utilise les caractéristiques du modèle de programme : il suffit d'enregistrer l'ordre de traitement des requêtes par les points d'entrée (événement **b** de la figure 2.2) pour pouvoir ré-exécuter les programmes de façon déterministe. Ce mécanisme est indépendant de l'implantation des communications sous-jacente et permet de réduire le nombre d'enregistrements d'un facteur deux à six relativement aux implantations « classiques » de la ré-exécution déterministe, qui enregistrent l'ordre d'arrivée de l'ensemble des messages dans le système [10].

2.3.3 MODÈLE FORMEL

Si on applique le modèle formel général défini par Leu, il est nécessaire d'enregistrer sur chaque nœud l'ordre d'arrivée de l'ensemble des messages reçus. On réduit le nombre d'enregistrements nécessaires à la ré-exécution en utilisant deux caractéristiques du modèle de programmation : la seule communication entre fils d'exécution est le passage de paramètres et de résultats, et chaque fil d'exécution retourne un résultat. Le modèle et la démonstration de condition suffisante d'équivalence sont adaptés de Mellor-Crummey [10, 66].

Dans ce qui suit, on appelle vp, ep le point d'entrée ep défini par le processeur virtuel vp.

Définition 2.3.1

Un historique de réception de requêtes $hr_{vp,ep}$ *est associé à chaque point d'entrée. Il définit l'ordre selon lequel les requêtes reçues sont traitées. L'historique du point d'entrée* ep *du processeur virtuel* vp *sera dénoté :*

$$hr_{vp,ep} = c_0^{vp,ep}, c_1^{vp,ep}, c_2^{vp,ep}, \ldots$$

Les historiques de réceptions de requêtes sont définis comme les séquences de fils d'exécution exécutés sur les points d'entrée. Chaque calcul $c_{th}^{vp,ep}$ est exécuté par le fil d'exécution identifié par le triplet unique $\langle vp, ep, th \rangle$ indiquant que le fil d'exécution th exécute le point d'entrée ep sur le processeur virtuel vp.

Définition 2.3.2

Un historique d'émission de requêtes he_p *est associé à chaque fil d'exécution* $p = \langle vp, ep, th \rangle$*. Il définit la séquence de requêtes émises par ce fil d'exécution durant l'exécution. Pour le fil d'exécution* p*, cet historique sera dénoté :*

$$he_p = e_0^p, e_1^p, e_2^p, \ldots$$

Chaque émission de requête $e_i^p = \langle vp, ep \rangle$ est transmise au point d'entrée ep du processeur virtuel vp.

Définition 2.3.3

La relation mapping M *est constituée d'un ensemble de triplets de la forme* $\langle p_1, i, p_2 \rangle$*, indiquant que la requête* $e_i^{p_1} = \langle vp, ep \rangle$ *est effectuée par le fil d'exécution* $p_2 = \langle vp, ep, th \rangle$*.*

Définition 2.3.4

Deux exécutions X et Y d'un programme procédural parallèle sont dites équivalentes si elles affectent à chaque fil d'exécution $p = \langle vp, ep, th \rangle$ le même historique d'émissions de requêtes.

Théorème 2.3.5

Soit $X = \langle H, E, M \rangle$ une exécution d'un programme procédural parallèle. Soit Y une autre exécution du même programme respectant les hypothèses habituelles de validité des mécanismes de ré-exécution déterministe (pas d'utilisation de primitives type random, etc., voir [10]). Pour que Y soit équivalente à X, il suffit de placer toutes les requêtes de Y en utilisant M.

La démonstration du théorème est faite par l'absurde en utilisant la relation d'ordre partiel entre les événements des exécutions parallèles induite par l'horloge logique de Fidge [71] (voir [10] pour plus de détails). Ce modèle et le résultat précédent sont applicables à la ré-exécution déterministe de tous les programmes parallèles utilisant un modèle de programmation basé sur l'appel de procédure à distance ainsi qu'aux applications structurées selon le modèle client-serveur.

2.3.4 VALIDATION

Expérimentalement, une maquette de ATHAPASCAN-0a comportant ce mécanisme a été implantée sur réseau de stations de travail et machine parallèle IBM SP-1. Cette maquette a été testée en comparant l'ordre des solutions produites par des programmes dont le comportement très indéterministe est reflété par l'ordre dans lequel sont produites leurs solutions [10].

Une évaluation systématique des surcoûts dus à l'enregistrement des traces a été faite en utilisant les outils de mesure de performances du projet Apache [34]. À partir de modèles d'algorithmes parallèles classiques, ces outils permettent de générer des programmes parallèles réels mais qui sont appelés « synthétiques » car ils ne font que consommer des ressources sans produire de résultat. Un même modèle permet de générer différents programmes synthétiques, paramétrés par le rapport entre coûts de communication et coûts de calcul.

Le principal résultat des mesures est que le surcoût dû à l'enregistrement de la

trace primaire ne dépasse pas 5%, même pour les programmes faisant beaucoup (trop) de communications. Aucun cas pathologique n'a pu être décelé en dépit d'un très grand nombre d'expériences (2400).

2.3.5 DÉVELOPPEMENTS ULTÉRIEURS

À partir du mécanisme de ré-exécution déterministe défini ci-dessus, plusieurs études ont été menées dans l'objectif de construire un outil de mise au point de programmes utilisant le modèle de programmation procédural parallèle.

2.3.5.1 SAISIE D'ÉTAT GLOBAL

Motivation. La possibilité de saisir un état global d'une exécution parallèle (*snapshot*) peut se révéler particulièrement utile pour le débogage. Elle permet en particulier de tester une propriété globale sur l'ensemble des processeurs participant à l'exécution parallèle considérée. Elle peut également servir de base à l'implantation d'un mécanisme de sauvegarde de point de reprise, ce qui peut faciliter la localisation d'une erreur dans un cycle de mise au point de programme de longue durée, en utilisant la ré-exécution déterministe.

Choix. Nous sommes partis d'une bibliographie des algorithmes proposés par la littérature [123] afin de sélectionner le plus approprié à notre objectif de mise au point et au modèle de programmation procédural parallèle. Le choix s'est porté sur un algorithme de saisie coordonnée, inspiré de Lai, Yang et Mattern [77, 82], plus adapté au débogage que les algorithmes de saisie non coordonnée. En effet, l'état global obtenu par un algorithme de saisie non coordonné comporte une plage de saisie potentiellement importante, tout en ayant une localisation imprévisible. L'algorithme de Lai, Yang et Mattern a été modifié afin de permettre de traiter plusieurs saisies d'états globaux successivement durant une exécution.

Validation. La maquette d'ATHAPASCAN-0a a été modifiée pour implanter l'algorithme durant la phase de ré-exécution déterministe [128]. Cette maquette a été validée expérimentalement en implantant une horloge vectorielle sur l'ensemble des processeurs virtuels d'une application et en testant la cohérence d'un état global ainsi saisi [128]. Ce travail a posé de nombreux problèmes techniques. On peut citer par exemple celui posé par la sauvegarde des messages en transit : la bibliothèque de communication PVM [88] utilisée par l'implémentation ATHAPASCAN-0a détruit le tampon contenant un message lors de sa lecture, ce qui a nécessité l'implantation d'une lecture non destructrice plus coûteuse en raison de

la duplication du message reçu qu'elle induit. Par ailleurs, les systèmes d'exploitation utilisés ne permettaient pas d'enregistrer simplement l'état d'un processus de telle sorte qu'il soit possible de le redémarrer ultérieurement à partir de l'état sauvegardé : la raison est principalement la difficulté à reproduire l'environnement du processus sauvegardé : fichiers, entrées-sorties, etc. La solution la plus simple, qui a été envisagée mais n'a pu être testée faute de temps, consiste à cloner (*fork*) chacun des processus (tâche selon la terminologie PVM) qui sauvegarde son état ; le retour à l'état sauvegardé est alors possible en repartant des processus clonés.

2.3.5.2 RÉ-EXÉCUTION PARTIELLE

Une autre étude a été entreprise pour la définition et l'implantation d'un mécanisme de ré-exécution partielle des programmes ATHAPASCAN-0a [122]. L'objectif était de pouvoir se concentrer sur les procédures et processus suspects et de réduire l'espace d'états à considérer ainsi que les ressources nécessaires pour la mise au point.

À la différence des mécanismes de ré-exécution déterministe considérés dans cette thèse qui sont tous basés sur le contrôle, la ré-exécution partielle ne peut qu'être basée sur les données et à ce titre, nécessite l'enregistrement de toutes les entrées des procédures et processus observés et dont le volume est potentiellement important. Dans une ré-exécution partielle, on ne ré-exécute que les processus observés et on ne peut donc pas faire l'hypothèse que tous les processus qui calculent les entrées des processus observés vont être ré-exécutés d'où la nécessité d'enregistrer ces entrées. Le volume des données à enregistrer étant potentiellement important, leur enregistrement est donc susceptible de perturber de façon importante l'exécution du programme parallèle mis au point. Dans un cycle de mise au point, il semble donc préférable d'enregistrer les données nécessaires à une ré-exécution partielle durant une ré-exécution déterministe.

2.3.5.3 MESURE DE PERFORMANCES DE PROGRAMMES NON DÉTERMINISTES

Le travail qui a été entrepris avait pour objectif d'utiliser la ré-exécution déterministe pour la mesure de performances de programmes non déterministes. Il est détaillé au §2.5.

2.4 RÉ-EXÉCUTION DÉTERMINISTE DE PROGRAMMES À BASE DE FILS D'EXÉCUTION COMMUNICANTS

2.4.1 INTRODUCTION

On s'intéresse à la ré-exécution déterministe de programmes comportant des fils d'exécution communicants. Comme nous l'avons indiqué, ce modèle a été conçu pour des systèmes parallèles composés de nœuds multiprocesseurs à mémoire partagée, interconnectés par un réseau de communication rapide. Ses principales fonctionnalités sont la création et la terminaison dynamiques de fils d'exécution— localement ou à distance —, le partage de mémoire entre fils d'exécution d'un même nœud — qui peuvent se synchroniser en utilisant des verrous, variables conditions ou sémaphores —, et enfin les communications bloquantes ou non bloquantes entre fils d'exécution non locaux — à travers des ports de communication. ATHAPASCAN-0 peut en fait être vu comme une version *thread aware*[2] de la bibliothèque de communication standard MPI [72].

Le non déterminisme des exécutions de programmes parallèles provient des conditions de concurrence pour l'accès aux variables partagées, aux objets de synchronisation et pour la réception de messages. Le mécanisme de ré-exécution déterministe assure la ré-exécution déterministe de programmes comportant des conditions de concurrence sur les objets de synchronisation et la réception de messages [29, 39, 49]. Les accès non synchronisés à la mémoire partagée ne sont pas enregistrés étant donné qu'un tel enregistrement accroîtrait énormément le temps d'exécution.

La technique utilisée combine une adaptation des techniques développées à l'Université de Gand avec un travail original pour prendre en compte le non déterminisme provenant de l'utilisation de primitives de communication non bloquantes. La plus grande partie de ce travail a été faite par Michiel Ronsse de l'Université de Gand, dans le cadre d'un projet de coopération du programme d'actions intégrées Tournesol.

2.4.2 NIVEAU D'ABSTRACTION DES ENREGISTREMENTS

L'analyse du modèle de programmation a indiqué que, bien que possible, un enregistrement au niveau d'abstraction du modèle de programmation serait très complexe à mettre en œuvre, en raison du nombre important d'interactions possibles entre fils d'exécution pouvant servir à écrire des programmes à comportement non déterministe. En revanche, si on s'intéresse aux niveaux d'abstraction

[2] Qui exploite intelligemment le parallélisme entre fils d'exécution d'un même nœud. Certaines versions de MPI, qui ne le sont pas, bloquent tous les fils d'exécution d'un nœud lorsqu'un seul d'entre eux est bloqué en attente de communication

inférieurs à celui du modèle de programmation, on constate que parmi les primitives utilisées pour l'implantation de ATHAPASCAN-0, celles qui sont susceptibles de conduire à des comportements non déterministes sont peu nombreuses. ATHAPASCAN-0 étant basé sur une combinaison entre la norme `POSIX`, définissant les fonctionnalités des fils d'exécution, et la bibliothèque de communication `MPI`, c'est le niveau d'abstraction qui a été choisi pour définir le mécanisme de ré-exécution déterministe (voir figure 2.4). Ce choix a permis de définir un mécanisme simple. En revanche, les primitives concernées étant utilisées fréquemment, le travail a essentiellement porté sur la réduction du nombre d'enregistrements nécessaires à la ré-exécution.

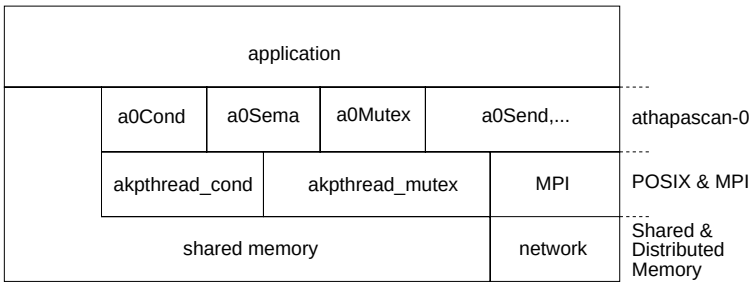

FIG. 2.4 – Niveau d'abstraction des enregistrements

Le résultat théorique présenté dans le §2.2.1 ainsi que les extensions apportées à l'université de Gand [107] sont utilisés pour définir le mécanisme de ré-exécution déterministe. Durant une exécution initiale sont enregistrés l'ordre de réception des messages ainsi que l'ordre d'accès aux objets de synchronisation. Les traces sont utilisées pour forcer les ré-exécutions des programmes à être équivalentes à l'exécution enregistrée, pourvu que ces ré-exécutions remplissent les hypothèses du théorème d'équivalence — même état initial, pas d'utilisation de fonctions telles que `random` et pas d'accès non synchronisé à la mémoire partagée.

2.4.3 RÉ-EXÉCUTION DES SYNCHRONISATIONS LOCALES

La ré-exécution déterministe des programmes en mémoire partagée est basée sur l'enregistrement de l'ordre des opérations de synchronisation. En forçant le même ordre durant les ré-exécutions, il est possible de garantir une ré-exécution déterministe correcte, à condition toujours que les hypothèses de validité de la ré-exécution déterministe soient assurées. C'est le cas si tous les accès à la mémoire partagée sont synchronisés. Si ce n'est pas le cas, il existe une condition de

concurrence sur la mémoire (*data race*) que nous considérons, sauf cas particulier [7, 47], comme une erreur. La ré-exécution doit permettre néanmoins de ré-exécuter de tels programmes de façon déterministe jusqu'à la première condition de concurrence au moins : il est ensuite possible au programmeur de rechercher les erreurs, soit manuellement, soit en utilisant des outils tels que JiTI [115], qui permet la détection automatique des conditions de concurrence.

La technique utilisée, développée initialement par Luk Levrouw [108] utilise l'horloge logique ROLT et la relation d'ordre partiel qui lui est associée pour réduire le nombre d'enregistrements nécessaires à la ré-exécution déterministe. L'horloge logique ROLT (*Reconstruction of Lamport Timestamps*) dérive de celle de Lamport [78] dont elle conserve les propriétés tout en ayant l'intérêt de croître moins rapidement [109]. La ré-exécution déterministe utilise la propriété de base de l'horloge de Lamport :

$$e_{p,i} \prec e_{q,j} \quad \Rightarrow \quad LC(e_{p,i}) < LC(e_{q,j})$$

avec :

- $\prec$: *happened before*, relation d'ordre partiel entre les événements d'une exécution parallèle, définie par Lamport.
- $e_{p,i}$, ($e_{q,j}$) : $i^{ième}$ ($j^{ième}$) événement du processus p (resp. q).
- $LC(e_{p,i})$, ($LC(e_{q,j})$) : horloge de Lamport de l'événement $e_{p,i}$ ($e_{q,j}$).

À chaque objet partagé (verrou, variable condition, etc.) on associe aussi un compteur de Lamport LC_O. Lors de l'accès à un objet partagé par un processus, ce compteur est utilisé pour calculer l'estampille de Lamport de l'événement :

$$LC(e_{p,i}) \leftarrow max(LC_p, LC_O) + 1$$

Cette estampille $LC(e_{p,i})$ est ensuite utilisée pour mettre à jour les horloges de Lamport du processus LC_p et de l'objet LC_O. Lors de la phase d'enregistrement (*Record*), on n'enregistre une information que si $LC_p < LC_O$, c'est à dire lorsque l'horloge de Lamport locale au processus est « en retard » et doit être incrémentée de plus d'une unité. Dans ce cas, on enregistre la valeur précédente de l'horloge de Lamport du processus $LC(e_{p,i-1})$ ainsi que sa nouvelle valeur $LC(e_{p,i})$ (voir figure 2.5).

Lors de chaque ré-exécution (*Replay*), on associe la valeur de l'horloge de Lamport de chaque événement de l'exécution enregistrée (*Record*) $e_{p,i}$ à chaque événement correspondant $\tilde{e}_{p,i}$ de la ré-exécution (*Replay*). On n'autorise la ré-exécution d'un événement que si tous les événements dont l'estampille est inférieure à celle de l'événement prêt à être exécuté ont déjà été exécutés, ce qui garantit l'accès à la même version de l'objet partagé. L'implantation de ce mécanisme utilise un tableau de compteurs SC partagé entre tous les fils d'exécution et dont la dimension est égale au nombre de processus utilisés par le programme

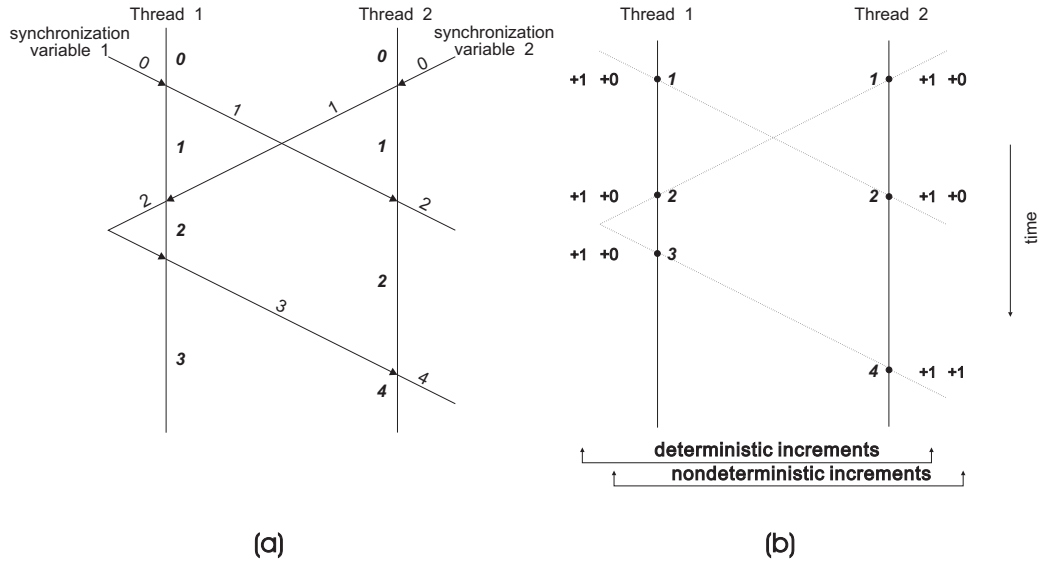

FIG. 2.5 – Mise à jour des horloges logiques ROLT

Les opérations concernant la même variable de synchronisation sont connectées par une flèche, partant de l'opération de synchronisation s'étant produite la première et dirigée en direction de la prochaine opération de synchronisation concernant la variable. Seuls les incréments non déterministes doivent être enregistrés.

parallèle. Après l'exécution d'un événement $\tilde{e}_{p,i}$, le $p^{ième}$ emplacement du tableau $SC[p]$ est mis à jour à $LC(\tilde{e}_{p,i+1}) = LC(e_{p,i+1})$, ce qui veut dire que $SC[p]$ est incrémenté de un si aucune valeur n'a été enregistrée ; dans le cas contraire, $SC[p]$ prend la valeur enregistrée ce qui revient à une incrémentation supérieure à un. Un accès à un objet partagé dénoté par l'événement $\tilde{e}_{q,j}$ ne peut être exécuté que si :

$$\forall p \;\; LC(\tilde{e}_{q,j}) \leq SC[p]$$

Dans [108], Levrouw *et al* démontrent formellement que leurs enregistrements sont suffisants pour assurer le même ordre d'accès aux objets de synchronisation dans l'exécution enregistrée et dans les ré-exécutions. Les mesures faites par Levrouw *et al* indiquent une réduction de la taille de la trace très supérieure à celle obtenue en utilisant la technique « optimale » (!) de Netzer et Miller [111] basée sur l'utilisation d'horloges vectorielles. La mise en œuvre de cette technique est réservée aux machines à mémoire partagée qui seules peuvent implanter efficacement le tableau SC. C'est pour cette raison qu'elle n'est utilisée que pour l'enregistrement de l'ordre des synchronisations locales à un nœud.

ATHAPASCAN-0 comporte trois types de synchronisations locales explicites : les verrous, les variables conditions et les sémaphores. En théorie, le verrou de synchronisation est à la base des mécanismes de variables condition et de séma-

phores et il devrait être suffisant de pouvoir ré-exécuter correctement les synchronisations par verrous pour traiter les autres synchronisations locales. En pratique, un traitement particulier a été nécessaire pour les variables conditions en raison de la possibilité, laissée par la bibliothèque *pthread*, d'exécuter un *signal* sur une variable condition sans détenir le verrou qui lui est associé. En outre, les deux synchronisations implicites lors de la création et de la terminaison des fils d'exécution doivent être prises en compte pour assurer la correction des ré-exécutions (voir [49] pour plus de détails).

2.4.4 INDÉTERMINISME DES COMMUNICATIONS

Dans la plupart des implémentations de MPI qui ne sont pas *thread-aware*, l'utilisation d'une primitive de communication bloquante par un fil d'exécution bloque l'ensemble fils d'exécution du même nœud. Pour éviter ce blocage, toutes les primitives de communication de ATHAPASCAN-0 sont implémentées par des communications non bloquantes dont la progression est gérée par un fil d'exécution particulier appelé **« démon de communication »**. Les structures de données gérées par ce démon sont protégées par un verrou de communication. Tout appel à une primitive de communication par un fil d'exécution applicatif provoque le verrouillage du verrou de communication, ce qui a pour effet de sérialiser les communications en provenance d'un même nœud.

2.4.4.1 MESSAGES POINT À POINT

La bibliothèque de communication MPI garantit la remise des messages suivant leur ordre d'émission, ce qui signifie que tous les messages échangés entre les fils d'exécution de deux nœuds distincts seront remis selon leur ordre d'émission. Dans la mesure où l'ordre des émissions de messages découlera de l'ordre de verrouillage du verrou de communication, ordre qui sera reproduit entre les ré-exécutions par le mécanisme décrit ci-dessus au §2.4.3, aucun traitement spécifique n'est requis pour reproduire l'ordre des communications entre les différents fils d'exécution de deux nœuds quelconques.

2.4.4.2 MESSAGES CONFLICTUELS

La seule situation de condition de concurrence à la réception de messages se produit en ATHAPASCAN lorsque deux fils d'exécution d'un même nœud reçoivent sur un même port de communication un message susceptible de provenir de n'importe quel nœud du système (paramètre `A0AnySource`, voir figure 2.6).

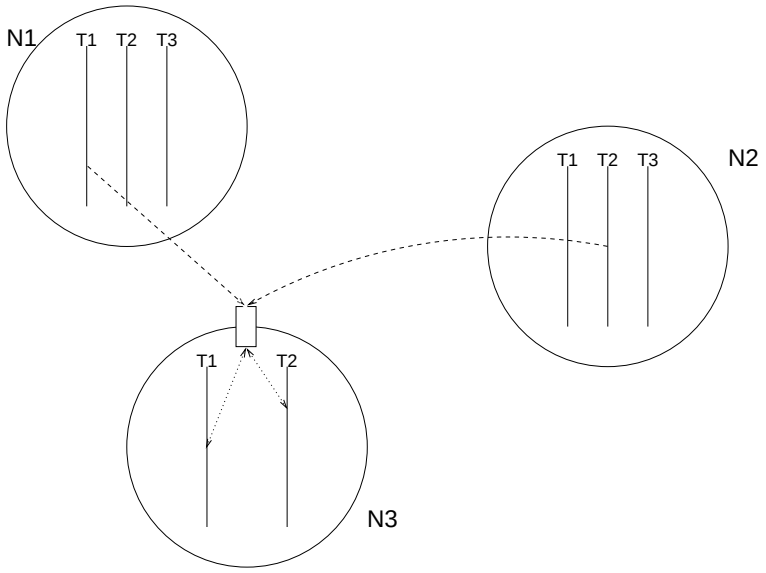

FIG. 2.6 – Réceptions de messages conflictuels sur un même port

La solution adoptée pour ré-exécuter de façon déterministe les programmes présentant cette situation est classique : enregistrer le nœud source du message et utiliser lors des ré-exécutions cet enregistrement pour forcer la prise en compte des messages par le fil d'exécution récepteur dans l'ordre de l'enregistrement. En fait, la situation est un peu plus complexe en raison de l'utilisation systématique de primitives de communication non bloquantes de `MPI` par l'implémentation d'ATHAPASCAN-0 (voir §2.4.5 et la référence [49]).

2.4.5 INDÉTERMINISME DES PRIMITIVES NON BLOQUANTES

En plus des conditions de concurrence pour les réceptions de message, une autre classe de primitives est à l'origine de comportements non déterministes de programmes à base de passage de messages : il s'agit des fonctions de test non bloquantes. Le non déterminisme introduit par l'utilisation de ces fonctions apparaît dans les bibliothèques de communication telles que `PVM` ou `MPI` et n'avait pas été traité par les mécanismes de ré-exécution déterministe existants [104]. La solution qui a été développée pour ATHAPASCAN-0 traite le problème au niveau `MPI` et est applicable aux bibliothèques `PVM` et `MPI` [29].

Les opérations de test non bloquantes sont pourtant utilisées intensivement dans les programmes à base de passage de message, par exemple pour recouvrir les communications par du calcul : ainsi une réception de message non bloquante

retourne un objet de type « requête » dès que la communication est initialisée, sans attendre son achèvement. Les objets de type requête peuvent être utilisés pour tester la terminaison d'une opération non bloquante par des primitives de test, qui elles mêmes peuvent être bloquantes (*Wait*), ou non bloquantes (*Test*).

De par leur nature non bloquante, les opérations de test sont susceptibles d'être utilisées dans des boucles de scrutation. Le nombre exact d'appels dépend de variations temporelles dans l'exécution des programmes parallèles et est donc non déterministe. Bien que de nombreux programmes n'utilisent pas le nombre d'opérations de tests, certains sont susceptibles de le faire (par exemple pour implémenter une sorte de chien de garde (*time-out*) et ne pourront donc être ré-exécutés correctement que si le même nombre de tests est exécuté durant la ré-exécution (voire figure 2.7). Une situation analogue se produit lorsque l'on considère une série d'appels aux fonctions testant l'arrivée d'un message (`MPI_IProbe`). La capacité de reproduire le même nombre d'opérations de test durant les ré-exécutions est absolument nécessaire pour le modèle ATHAPASCAN en raison de l'imbrication entre les appels à ces fonctions de test et l'utilisation des fonctions de synchronisation : si on ne sait pas reproduire le même nombre de tests, le nombre d'appels aux fonctions de synchronisation change et le mécanisme de ré-exécution des programmes en mémoire partagée ne peut fonctionner correctement.

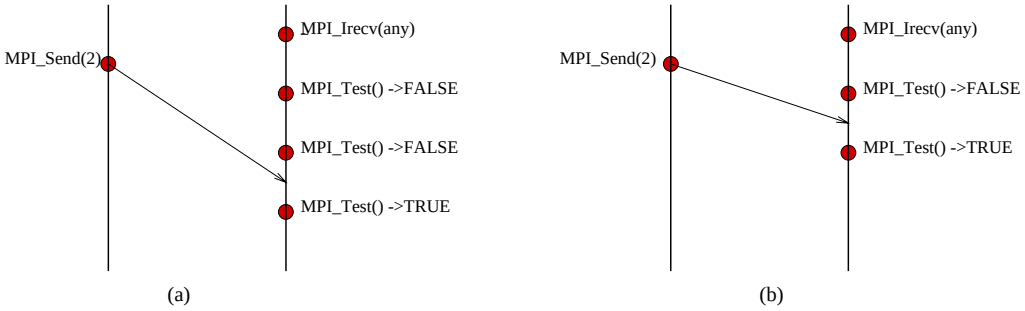

FIG. 2.7 – Le nombre d'opérations de test non bloquantes dépend de variations temporelles infimes.

La ré-exécution basée sur le contrôle implique que le positionnement de la condition testée (par exemple l'arrivée d'un message), soit retardé jusqu'à ce que le nombre d'opérations de test de l'exécution enregistrée ait été effectué. C'est alors seulement que l'opération qui positionne la condition testée peut être autorisée à reprendre. La ré-exécution de ces fonctions de test ne pose aucun problème pour un système de ré-exécution purement dirigé par le contrôle, pour lequel l'ordre de **tous** les tests est enregistré. Cette approche comporte un inconvénient important. Les opérations de test non bloquantes sont typiquement utilisées à l'intérieur de boucles de scrutation susceptibles de s'exécuter longtemps. Il n'est

donc pas acceptable de laisser grossir le fichier de trace lorsque le fil d'exécution concerné ne fait pas autre chose qu'attendre.

Il s'avère que la ré-exécution dirigée par les données est plus efficace pour traiter ce problème que la ré-exécution dirigée par le contrôle. Cela provient du fait que les fonctions de tests retournent habituellement une série d'échecs, suivie par un succès, à moins que le programme ne se termine avant le succès du test, quand la requête est annulée (`MPI_Cancel`) ou lorsque l'application cesse sa scrutation et utilise la primitive `MPI_Wait` pour attendre la fin de la requête en cours. Étant donné que tous les appels à une fonction de test donnée, pour une même requête, sont identiques, il est possible de les compter et enregistrer simplement le nombre de tests infructueux. Durant les phases de ré-exécution, le nombre d'essais, provenant des traces, est décrémenté à chaque test et dès que ce nombre devient nul, le nombre de tests de l'exécution enregistrée ayant alors été exécuté, l'appel à la fonction de test est transformé en attente de fin de requête `MPI_Wait`. Si la série d'appels à une fonction de test demeure sans succès, durant l'exécution enregistrée, une grande valeur est enregistrée ce qui force, durant les ré-exécutions, tous les tests à demeurer infructueux étant donné que le compteur ne devient jamais nul. Une technique analogue est utilisée pour enregistrer le nombre d'appels infructueux à `MPI_IProbe`.

La difficulté de cette technique est de lire dans les traces le nombre de tests infructueux dès que le premier appel d'une série de `MPI_Test` est effectué. De façon analogue, en cas de réception de message dans une situation de compétition, il est nécessaire de lire l'identité du nœud émetteur depuis les traces dès l'appel à la fonction de réception. Ces deux problèmes ont été résolus en assignant un numéro de requête au premier des appels d'une série de `MPI_Test` ainsi qu'à chaque `MPI_IRecv` donnant lieu à une condition de concurrence. Ces numéros de requêtes sont stockés dans les fichiers de trace avec le nombre de tests infructueux (ou l'identité du nœud émetteur) et les fichiers de trace sont triés avant la première ré-exécution. Une présentation technique détaillée de la solution adoptée est présentée dans un rapport de recherche [29]. Cette approche réduit de façon considérable la taille des traces enregistrées durant l'exécution de programmes qui utilisent intensivement les opérations non bloquantes. C'est le cas de tous les programmes ATHAPASCAN, puisque le noyau exécutif ATHAPASCAN transforme les requêtes bloquantes en requêtes non bloquantes, d'une part pour éviter les risques d'interblocages dus aux communications et d'autre part pour permettre un bon recouvrement des calculs par les communications.

L'utilisation d'une technique de ré-exécution basée sur les données pour la ré-exécution des opérations de test non bloquantes présente un autre avantage. Comme le nombre de tests infructueux est connu, il n'est plus nécessaire d'appeler les fonctions de test de la bibliothèque de communication. Les appels infructueux aux fonctions de test sont remplacés dynamiquement par un test et une décrémen-

tation de compteur qui retourne un échec, le test réussi étant quant à lui remplacé par une opération bloquante (*Wait*). Bien sûr ce mécanisme ne fonctionne que si l'on peut garantir que les appels infructueux aux primitives de test ne provoquent aucun effet de bord, ce qui est le cas pour MPI et PVM. Le remplacement des appels aux fonctions de la bibliothèque de communication par de simples tests accélère le traitement des fonctions de test non bloquantes. Les programmes qui pratiquent intensivement la scrutation s'exécutent parfois plus rapidement durant les ré-exécutions puisqu'il n'est plus nécessaire d'appeler la bibliothèque de communication.

2.4.6 VALIDATION

Une maquette d'ATHAPASCAN-0 a été instrumentée et testée sur un certain nombre de programmes jouets conçus pour présenter de nombreuses conditions de concurrence et fournir un résultat reflétant le chemin suivi par un exécution particulière. Par ailleurs cette maquette a été utilisée pour exécuter un certain nombre de programmes ATHAPASCAN-0 existants dont en particulier une implémentation de ATHAPASCAN-1 [100]. Quelques mesures ont été effectuées : le très faible ralentissement de l'exécution des programmes en mode *replay* s'explique vraisemblablement par le fait que durant la ré-exécution, la plupart des appels faits par le démon de communication à *MPI_test* ou *MPI_probe* sont remplacés par une simple comparaison entre deux entiers.

programme	normal	record	replay	taille traces(octets)
mandelbrot	1.858+-0.003s.	1.914+-0.037s.	1.864+-0.002s.	3912 (octets)
queens(12)	6.70 +-0.15 s.	6.58 +-0.05 s.	6.86 +-0.17 s.	1710 (octets)
scalprod	4.38 +-0.04 s.	4.44 +-0.07 s.	4.66 +-0.03 s.	874 (octets)

2.5 MESURE DE PERFORMANCES DE PROGRAMMES NON DÉTERMINISTES

2.5.1 MOTIVATION

Les performances des programmes parallèles sont mesurées le plus souvent par la collecte et l'analyse de traces d'exécutions parallèles[17]. Une trace est la

collecte des événements observables d'une exécution, envoi et réception de message par exemple, chaque événement comportant au moins un type, la date physique à laquelle il s'est produit ainsi que l'identité du (des) processus concerné(s). Le traçage est plus adapté au parallélisme que l'échantillonnage et plus précis que le comptage. Un traceur logiciel est plus facile à mettre en œuvre et plus portable qu'un traceur matériel ou hybride. Par contre, ce type de traceur perturbe l'exécution observée et suscite un doute sur la validité des observations. Pour corriger cet effet de sonde, des mécanismes de correction *post mortem* des traces ont été développés [17, 65, 81]. L'algorithme de É. Maillet, mis en œuvre dans le traceur Tape/PVM [65], utilise une évaluation statistique du surcoût du à l'enregistrement d'un événement. L'algorithme parcourt la trace d'exécution pour retrancher des valeurs d'horloge enregistrées le surcoût dû à l'enregistrement de la trace, pour restituer à ces estampilles temporelles la valeur qu'elles auraient eu en l'absence du traceur. Cette correction ne peut être faite pour certaines exécutions de programmes à comportement non déterministe, lorsque l'on constate que la prise de trace a modifié le comportement du programme. Dans ce cas, É. Maillet utilise une « approximation conservatrice » en conservant l'ordre des événements de l'exécution enregistrée.

2.5.2 APPROCHE

Étant donné que la prise de trace nécessaire à la ré-exécution déterministe ne perturbe que très faiblement l'exécution des programmes parallèles observés [34], on prend comme exécution de référence une exécution durant laquelle ces traces sont enregistrées (mode *record*). Des traces pour la mesure de performances sont ensuite enregistrées durant une ré-exécution du programme mesuré (mode *replay*). Un algorithme de correction *post mortem* analogue à celui de É. Maillet est ensuite mis en œuvre pour restituer aux estampilles temporelles les valeurs qu'elles auraient eu lors de l'exécution initiale (*record*). Ce mécanisme doit corriger les surcoûts dus au mécanisme de ré-exécution déterministe ainsi qu'à la prise de traces pour la mesure de performances. Il est basé sur l'hypothèse d'invariance suivante : « Les durées des phases de calcul et de communication sont invariantes entre les exécutions *record* et *replay* » [45].

Cette solution a été mise en œuvre dans un traceur ATHAPASCAN-0a [44, 45]. La collecte de trace initiale (*record*) a été modifiée pour enregistrer la date physique en plus des informations nécessaires à la ré-exécution.

Ce travail a mis en évidence un certain nombre de difficultés liées à la technique employée. La plus sérieuse concerne l'influence du système d'exploitation sous-jacent dans un environnement « perturbé ». Dans ce cas, l'exécution des

processeurs virtuels (processus UNIX) est susceptible d'être suspendue par le système d'exploitation (UNIX) durant l'une des exécutions, *record* ou *replay*. Ce phénomène n'est pas détectable au niveau du traceur et met en défaut l'hypothèse d'invariance sur laquelle est fondé la méthode de correction. En revanche, il peut être détecté durant la correction et signalé à l'utilisateur. Une autre difficulté provient de la précision « insuffisante » de la datation des événements relativement aux exécutions de processus légers observées : il peut ainsi sembler que deux processus légers soient actifs simultanément sur un processeur unique.

2.5.3 PERSPECTIVES

Un prolongement naturel du travail décrit ci-dessus consiste à l'étendre au modèle de programmation à base de fils d'exécution communicants. Afin de déterminer si l'importance de l'intrusion due au traçage justifie l'étude d'un mécanisme de correction, une évaluation expérimentale de cette intrusion a été faite, en utilisant le traceur ATHAPASCAN-0 [26]. Ses résultats indiquent que l'intrusion ne semble pas décelable [119]. Cette évaluation ayant été faite sur des applications « jouet », exécutées sur un faible nombre de processeurs, elle devra être confirmée pour des applications réelles exécutées sur des systèmes de plus grande taille.

Si des études complémentaires, infirmant les résultats préliminaires, justifient de corriger l'intrusion due à la prise de traces, il sera nécessaire de redéfinir complètement le modèle de correction de l'intrusion. Si l'on utilise l'outil de réexécution présenté au §2.4, les optimisations faites pour limiter l'intrusion et donc le nombre de points de trace en mode *record* poseront des difficultés pour mesurer les dates de référence utilisées par l'algorithme de correction.

Les techniques de correction de l'intrusion développées dans notre équipe ont également inspiré une méthode de correction de l'intrusion due à la prise de traces pour les programmes non déterministes utilisant la bibliothèque de communication MPI. Ce travail a été effectué par D. Kranzlmüller et C. Schaubschläger de l'Université de Linz (Autriche), dans le cadre d'un partenariat financé par le programme d'actions intégrées Amadeus [35].

2.6 CONCLUSION ET PERSPECTIVES

Nous nous sommes intéressés aux techniques permettant de limiter les conséquences du non déterminisme des programmes à base de fils d'exécution commu-

nicants sur les activités de débogage et de mesure de performances.

La première étude concernait le modèle de programmation procédural parallèle. Une technique originale indépendante du modèle de communication sous-jacent a été définie, implémentée et mesurée. Cette technique est applicable à d'autres modèles de programmation tels que le modèle client serveur. Une autre étude a été menée pour un modèle de programmation à base de fils d'exécution communicants, la communication se faisant par mémoire partagée à l'intérieur d'un même nœud et par passage de messages entre nœuds distincts. Le mécanisme de ré-exécution déterministe qui a été défini, implémenté et testé combine les techniques les plus efficaces en mémoire partagée avec des mécanismes originaux — applicables aux bibliothèques de communication PVM et MPI — pour le traitement des primitives non bloquantes. L'ensemble de ces travaux a permis d'acquérir la maîtrise des techniques nécessaires à la ré-exécution déterministe des programmes parallèles à base de fils d'exécution communicants. Ces travaux peuvent être utilisés directement pour concevoir un mécanisme de ré-exécution déterministe adapté à une bibliothèque de communication *thread-aware*. Par ailleurs, une expérimentation a été menée pour utiliser la ré-exécution déterministe pour mesurer les performances des programmes à comportement non déterministe et ainsi permettre de corriger l'intrusion due au traçage logiciel de ces programmes.

La principale perspective de ce travail concerne l'utilisation du mécanisme de ré-exécution comme base d'un environnement de mise au point. Cet environnement combinera un débogueur distribué avec l'outil de visualisation Pajé décrit au chapitre 3 : la présentation des objectifs de cet environnement et des difficultés que posent sa mise en œuvre font l'objet du chapitre 4.

Un autre prolongement du travail sur la ré-exécution déterministe est l'étude du passage à l'échelle sur une grappe comportant un grand nombre de processeurs. Si ce travail ne semble pas devoir poser de problèmes théoriques difficiles, il est probable qu'il soulèvera de nombreux problèmes pratiques. Une expérimentation pourra être réalisée dans le cadre du projet 200 PCs mené au laboratoire ID.

3

VISUALISATION D'EXÉCUTIONS PARALLÈLES

Visualisation : « Présentation d'informations sur un écran (de télévision ; d'oscilloscope). » *Le Petit Robert. 1981.*

L'objectif est de fournir aux programmeurs un outil les aidant à identifier les « erreurs de performances » de programmes parallèles comportant éventuellement de nombreux fils d'exécution. Le débogage pour les performances nécessite le calcul d'un grand nombre d'indices de performances [93]. On peut classer ces indices en deux catégories [17] :

Durées d'exécution : que l'on cherche à réduire autant que possible. La principale mesure est la durée d'exécution totale du programme. Cependant on va aussi chercher à mesurer le temps passé dans différentes parties du programme : procédures, protocoles de communication, etc.

Taux d'utilisation des ressources : afin de savoir quel pourcentage de temps a été passé à faire du travail utile. Si on considère les taux d'utilisation des processeurs, les programmeurs doivent savoir quel pourcentage du temps a été perdu dans divers surcoûts tels que l'exécution d'une synchronisation, la création d'un fil d'exécution ou d'une tâche, l'ordonnancement des tâches, les communications, l'oisiveté, etc. Des indices globaux sont susceptibles

d'indiquer des problèmes tels qu'une faible utilisation des processeurs et un taux d'oisiveté important. Cependant la correction de ce type de problème nécessite souvent l'accès à des données plus détaillées. Par exemple, un taux d'oisiveté important peut provenir d'un goulot d'étranglement dont l'origine peut elle-même provenir d'un manque de parallélisme dans le programme, à moins qu'il ne s'agisse d'une erreur d'ordonnancement ou encore d'un usage excessif des synchronisations.

Ici encore nous n'avons pas cherché à développer des outils de détection automatique des erreurs de performances [84]. Ces outils sont susceptibles de rendre des services considérables, particulièrement pour l'élimination des erreurs de performances de programmes de longue durée s'exécutant sur des systèmes de grande taille (voir §3.1.2). Cependant il nous a semblé que les erreurs de performances susceptibles d'être détectées par de tels outils doivent avoir été identifiées au préalable et, en ce domaine, l'exhaustivité nous a semblé un objectif hors de notre portée ! Nous avons donc choisi de développer un outil permettant de présenter de façon aussi claire que possible au programmeur les informations lui permettant d'identifier lui-même ses erreurs, aussi facilement que possible.

La technique utilisée pour collecter les indices de performances est le traçage logiciel des applications. Ainsi que nous l'avons mentionné au § 2.5.1, le traçage consiste à enregistrer dans une trace d'exécution les événements significatifs des exécutions ainsi que leur date d'occurrence : début et fin de fil d'exécution, envoi et réception de message, appel d'une fonction de synchronisation, etc. La raison de ce choix est d'une part que le traçage est la méthode la plus générale de mesure de performances — comparée à l'échantillonnage ou au comptage par exemple — et que d'autre part le traceur logiciel est le plus facile à réaliser — comparé aux traceurs matériels ou hybrides (voir la référence [17] pour plus de détails sur le traçage).

La méthode choisie pour représenter l'activité des processeurs, telle qu'on peut la déduire des traces, est la visualisation des exécutions des programmes. Grâce aux différentes vues (voir figure 3.1) et en particulier aux représentations de l'activité des processeurs en fonction du temps, par processeur (diagramme dit de Gantt) et agrégée (diagramme de Kiviatt), ainsi que grâce aux représentations des communications (diagramme « espace-temps »), il semble possible de mettre en évidence certains goulots d'étranglement provenant par exemple d'un mauvais recouvrement entre calculs et communications, et de les éliminer [101]. Pour limiter la perturbation des programmes observés, ces outils sont le plus souvent utilisés `post mortem` : ainsi il n'est pas nécessaire de transporter les traces hors du système étudié en concurrence avec l'application observée, ce qui serait susceptible de perturber les communications du système observé d'une part ; d'autre part, la visualisation étant plus lente que l'application observée, il serait nécessaire d'as-

servir le déroulement de la seconde au calcul des visualisations.

L'étude des outils de visualisation existants a montré leur inadéquation à la visualisation d'exécution parallèles de programmes à base de fils d'exécution communicants. Les recherches qui ont été menées, dans le cadre de la thèse de Benhur Stein [67], ont conduit à la réalisation de l'environnement de visualisation Pajé. Pajé combine de façon originale plusieurs propriétés essentielles pour un environnement de visualisation (cf. §3.1.3) : l'**extensibilité**, qui permet d'ajouter « facilement » de nouvelles fonctionnalités à l'environnement, l'**interactivité**, qui permet les déplacements dans le temps, l'interrogation des objets visualisés, etc., et enfin la « **scalabilité** »[1] qui permet de représenter un nombre potentiellement grand de fils d'exécution dans une exécution éventuellement longue. Pajé a été utilisé dans le projet Apache, pour visualiser l'exécution de programmes ATHAPASCAN-0 et ATHAPASCAN-1. Sa souplesse permet d'envisager son utilisation dans des contextes variés : des tests sont actuellement en cours dans des domaines aussi différents que la visualisation d'exécutions de programmes Java ou encore l'aide à l'administration de grappes de grande taille.

3.1 INADÉQUATION DES OUTILS EXISTANTS

La visualisation d'exécutions comportant un grand nombre de fils d'exécution posait des problèmes spécifiques, liés au modèle de programmation et non résolus par les outils existants, ainsi que des problèmes plus classiques, plus ou moins bien résolus par ces mêmes outils.

3.1.1 EXEMPLES D'OUTILS DE VISUALISATION

Nous présentons ici rapidement quelques outils de visualisation. Pour une bibliographie plus complète sur les outils de visualisation voir les revues de Kraemer et Stasko [75] ainsi que la bibliographie de la thèse de B. Stein [67].

ParaGraph [74, 101] est un des premiers outils de visualisation à avoir été disponible gratuitement. Il a popularisé l'utilisation de la visualisation pour l'évaluation des performances des applications parallèles. ParaGraph est l'un des outils les plus complets existants actuellement : il propose un grand nombre de visualisations (voir figure 3.1).

[1] Anglicisme utilisé pour traduire *scalability* dont la traduction en bon français serait la périphrase « aptitude à passer à l'échelle » et qui désigne un concept différent de *extensibility* que nous traduisons par extensibilité.

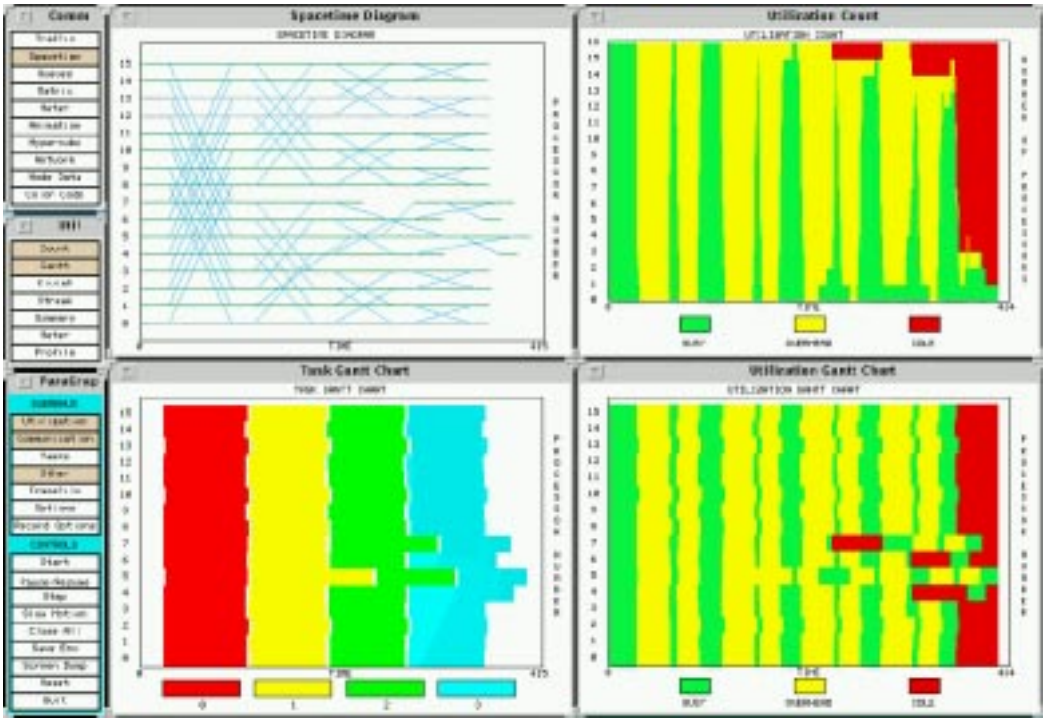

FIG. 3.1 – Quelques visualisations de ParaGraph

Relativement à nos contraintes, les possibilités d'extension de Paragraph restent limitées. Il est possible d'ajouter une visualisation. L'extension du modèle de programmation, ou son adaptation à un modèle différent de celui pour lequel il a été conçu n'est pas simple car l'outil est construit d'une façon monolithique et il n'existe pas de division nette entre la lecture, la simulation et la visualisation des données. L'outil ne supporte pas de concepts comme les fils d'exécution et primitives de synchronisation autres que les échanges de messages. Les visualisations ne sont pas interactives, l'outil n'offrant aucune aide pour trouver une correspondance entre données présentées sur fenêtres différentes ou entre les données présentées et le code source.

L'extensibilité est l'une des principales caractéristiques de Pablo [125] dont l'architecture est basée sur un graphe de composants qui peuvent être connectés par les utilisateur avertis, en utilisant une interface graphique, pour produire un outil de visualisation donné. Pablo est totalement indépendant du modèle de programmation utilisé par le programme tracé. Cette indépendance est obtenue en se limitant à montrer des données statistiques qui ne donnent aucune indication sur le comportement du programme visualisé et ne permettent pas l'interaction avec les visualisations.

Paradyn [84, 94] a été conçu pour identifier les erreurs de performance au-

tomatiquement. Ses modules de collecte de données peuvent se connecter à un programme en cours d'exécution, dont ils extraient des indices de performances qui sont analysés en ligne. Paradyn étant conçu pour l'étude de programmes s'exécutant pendant longtemps sur un grand nombre de processeurs, le volume de données collectées — essentiellement des valeurs de données ou de chronomètres — est modéré. Si un problème de performance est détecté à un endroit donné du programme, des données supplémentaires sont collectées à cet endroit. Des visualisations indiquent à l'utilisateur la progression de la recherche — les hypothèses explorées sont classées selon un arbre — ainsi que données synthétiques, histogrammes par exemple. En revanche, Paradyn ne permet aucune visualisation du comportement des applications.

Annai [118, 120] est un environnement intégré de mise au point de programmes parallèles comportant un débogueur symbolique, un outil d'analyse et de visualisation de traces d'exécution. Le débogueur permet la représentation graphique des structures de données réparties, la détection d'interblocages et la ré-exécution déterministe. Annai n'offre cependant pas de support pour la visualisation de fils d'exécution.

3.1.2 Problèmes spécifiques

Il s'agit essentiellement de la difficulté que pose la visualisation d'un réseau de fils d'exécution de grande taille et évoluant dynamiquement. La donnée d'un nombre éventuellement grand de fils d'exécution est un problème analogue au problème classique de scalabilité que pose la visualisation des exécutions de programmes sur machines « massivement parallèles », comportant plusieurs centaines ou milliers de nœuds. Une approche classique à ce problème est de ne développer que des représentations « scalables », c'est-à-dire dont le « format, la clarté, le sens et la taille sont indépendantes du nombre de nœuds de calcul utilisés » [69]. Si cette approche est utile à un haut niveau d'abstraction, au début d'une recherche d'erreur de performance (voir figure 3.7 par exemple), elle ne permet pas une analyse détaillée, souvent nécessaire pour déterminer l'origine d'un problème. Le choix fait dans Pajé est de permettre à la fois des représentations d'un haut niveau d'abstraction et des visualisations détaillées.

En effet, même avec un nombre limité de nœuds on peut avoir une quantité importante de fils d'exécution. Relativement au problème classique de la « scalabilité », il y a un paramètre supplémentaire provenant de la dynamicité du modèle : le nombre de fils d'exécution est susceptible de varier rapidement durant l'exécution. Le nombre variable et potentiellement important de fils d'exécution pose certains problèmes pour leur visualisation, comme la limitation de l'espace disponible sur un écran pour les afficher, et la difficulté de compréhension d'un tel affichage. Le

système *Gthread* [129] permet la visualisation de plusieurs fils d'exécution pouvant éventuellement être créés et détruits dynamiquement. Cependant, il est conçu pour un système de programmation ne comportant qu'un seul niveau de parallélisme à l'intérieur d'un même nœud, ce nœud étant en général un multiprocesseur symétrique à mémoire partagée.

Les outils de visualisation de programmes parallèles existants comme Paragraph [74], n'ont pas été conçus pour visualiser l'activité de nœuds multiprogrammés. Leurs vues montrent l'activité des nœuds du système, ainsi que leurs communications (voir figure 3.1). Il est bien sûr possible d'utiliser ces systèmes pour visualiser l'activité de fils d'exécution en représentant ces derniers comme des nœuds. Dans ce cas, la durée de vie des fils d'exécution doit être celle du programme, et leur nombre doit être (relativement) limité et rester constant durant l'exécution. Cette solution n'est donc pas adaptée à la visualisation de fils d'exécution dont le nombre varie continuellement et la durée de vie peut être faible.

3.1.3 Problèmes classiques

Afin de se révéler pratiquement utilisables, les outils de visualisation doivent combiner les propriétés d'extensibilité et d'interactivité. L'extensibilité permet d'étendre l'environnement de visualisation en fournissant de nouvelles fonctionnalités : analyse de traces d'un type nouveau, adjonction de nouvelles visualisations ou encore visualisation de nouveaux modèles de programmation parallèle. L'extensibilité est une propriété importante car il est coûteux de développer un outil de visualisation d'une part et les modèles de programmation parallèle ne sont pas encore stabilisés d'autre part. À la différence d'un outil passif où les entités de l'exécution parallèle visualisée — communications, changements d'état des processus, etc. — sont visualisées dès qu'elles sont calculées sans pouvoir être « interrogées » ultérieurement, un outil interactif permet d'« inspecter » tous les objets visualisés. Cette fonctionnalité permet d'obtenir des détails qu'il n'est pas possible d'afficher faute d'espace sur l'écran comme la taille d'un message ou encore le programme source correspondant à un événement.

Là encore, si l'objectif principal d'un outil comme Pablo est d'être facilement extensible [125], cette propriété est assurée par des choix d'architecture incompatibles avec l'interactivité. Cette contradiction a été brisée par Pajé (voir §3.5) et, il n'existe pas à notre connaissance d'outil combinant ainsi les propriétés d'extensibilité et d'interactivité.

3.2 RECHERCHES MENÉES

Comme nous l'avons indiqué plus haut, les recherches qui ont été menées dans le cadre de la thèse de Benhur Stein se sont concrétisées dans l'environnement de visualisation Pajé. La principale qualité de Pajé est une combinaison originale des propriétés d'interactivité, d'extensibilité et de scalabilité [67].

Scalabilité : Pajé offre une forme de « scalabilité » en permettant aux utilisateurs de « zoomer » **interactivement** dans l'espace et dans le temps. Dans l'espace car il est possible d'observer une exécution à différents niveaux d'abstraction : groupes de nœuds, nœuds, fils d'exécution. De façon analogue, il est possible de changer dynamiquement l'échelle de représentation du temps ce qui permet de passer de la représentation d'une longue période de temps à une représentation d'une période plus courte, où plus de détails seront visibles. Le fait que le passage d'un niveau d'abstraction à un autre et que le changement d'échelle du temps soient **interactifs** a une grande importance pour la scalabilité car, si la vision d'ensemble est la seule façon de maîtriser la grande quantité d'informations à traiter, la vision détaillée est la seule façon de mettre en évidence l'origine d'un problème de performances ; si l'environnement n'était pas interactif, il devrait se limiter à des représentations graphiques « scalable » qui sont habituellement calculées à partir de moyennes et ne permettent pas d'analyse détaillée.

Interactivité : Outre les interactions évoquées ci-dessus — possibilités de changer le niveau d'abstraction et l'échelle de temps durant la session de visualisation —, Pajé permet d'inspecter les objets visualisés (fils d'exécution, communications, etc.), de se déplacer en avant et en arrière dans le temps, etc.

Extensibilité : Pajé offre plusieurs caractéristiques destinées à en faciliter l'extension : conception en modules indépendants, indépendance des visualisations relativement à la sémantique du modèle de programmation parallèle visualisé et enfin généricité du module de simulation qui constitue le cœur de Pajé. Cette dernière propriété de Pajé donne en fait aux « programmeurs d'applications » la possibilité de spécifier **dans les programmes tracés** ce qu'ils veulent visualiser et comment la visualisation doit être faite.

3.3 VISUALISATIONS DE PAJÉ

Comme nous l'avons indiqué, l'un des principaux objectifs du projet Apache est de maximiser l'utilisation des processeurs par des techniques de recouvrement des communications par les calculs ou encore en utilisant des techniques d'ordonnancement dynamique. De toutes les vues disponibles dans les outils existants, le diagramme espace-temps (dit aussi « de Feynmann ») semble le plus utile à la compréhension des programmes à parallélisme de contrôle. Par ailleurs le diagramme dit de Gantt est le plus utile quand il s'agit de visualiser l'occupation des processeurs. Pour toutes ces raisons, la principale visualisation de Pajé est une combinaison étendue des diagrammes de Gantt et de Feynmann, qui permet de visualiser l'exécution de programmes comportant un nombre fixe de nœuds dont chacun exécute un nombre non borné de fils d'exécution.

3.3.1 DIAGRAMME ESPACE-TEMPS

Ce diagramme combine en une seule représentation les états successifs, les communications et les événements de chaque fil d'exécution, les opérations et les états des sémaphores. L'espace alloué à chaque nœud s'adapte dynamiquement au nombre de fils d'exécution qui s'y exécutent (voir figure 3.2).

L'axe horizontal représente le temps, et l'axe vertical représente les fils d'exécution, groupés par nœud. Les communications sont représentées par des flèches, les états des fils d'exécution par des rectangles, et les événements par des triangles. Les différentes couleurs représentent le type de communication, d'activité des fils d'exécution ou d'événement. L'affichage de détails supplémentaires sur ces objets est fait en sélectionnant l'objet auquel on s'intéresse.

3.3.2 VISUALISATION DES SYNCHRONISATIONS LOCALES

Dans Pajé, l'activité des nœuds multiprogrammés est représentée par un diagramme combinant en une représentation unique les états et les communications de chaque fil d'exécution. Les états des sémaphores et des verrous sont représentés comme les états des fils d'exécution : à chaque état possible est associé une couleur et un rectangle de cette couleur est représenté dans la position correspondant à la période de temps où le sémaphore se trouvait dans cet état. Chaque verrou est associé à une couleur, et un rectangle de cette couleur est dessiné à proximité du fil d'exécution qui possède ce sémaphore (voir figure 3.3).

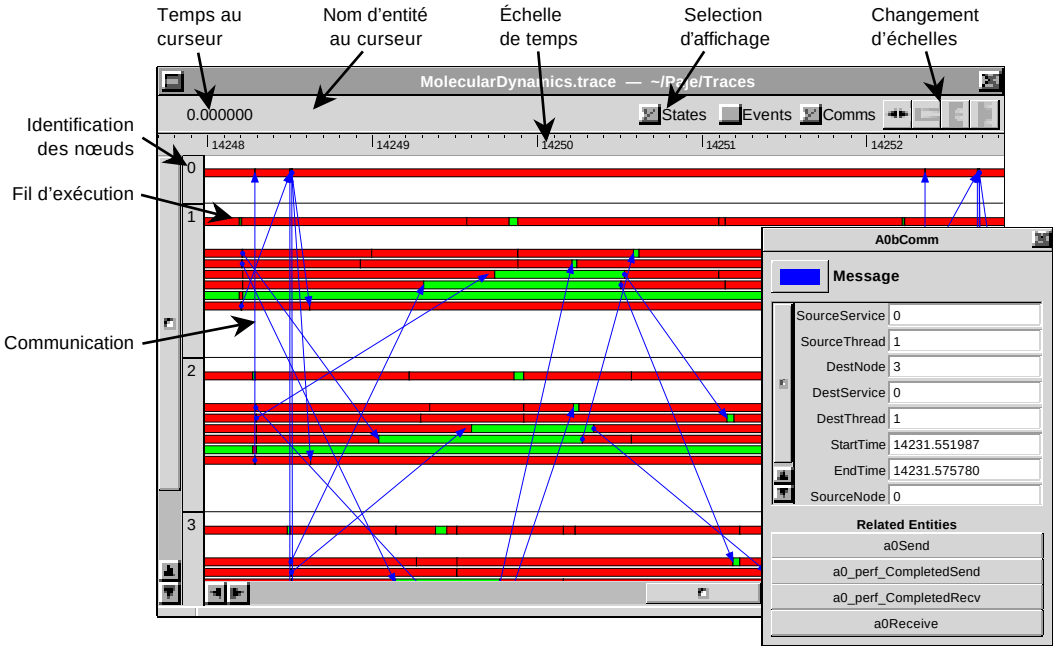

FIG. 3.2 – Diagramme espace-temps étendu

Une couleur foncée indique l'état bloqué d'un fil d'exécution tandis qu'une couleur claire indique un état activable. La fenêtre en premier plan montre l'inspection d'un événement de communication.

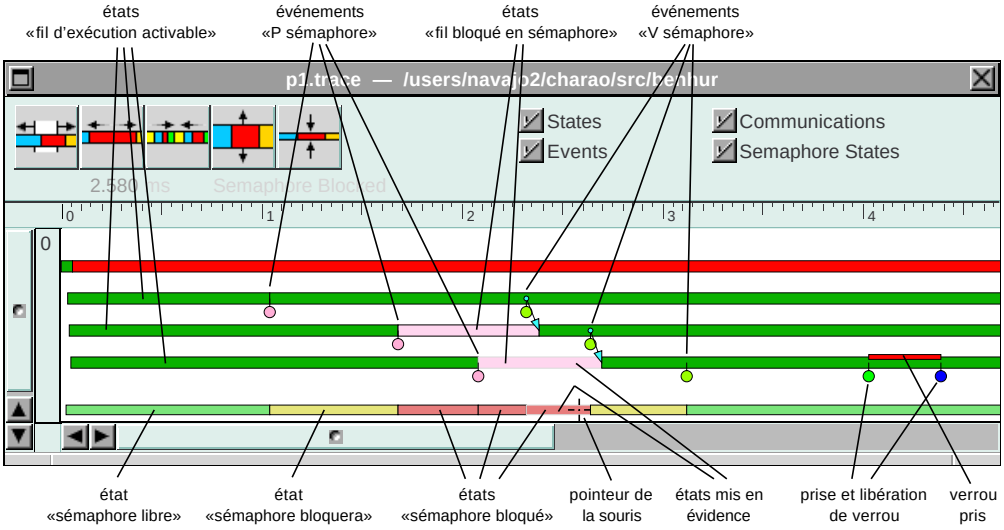

FIG. 3.3 – Visualisation des sémaphores

3.4 EXTENSIBILITÉ

L'extensibilité est une propriété fondamentale pour un outil de visualisation. La principale raison est de prolonger autant que possible la durée de vie d'un outil très complexe à réaliser. Pour survivre, un outil de visualisation doit s'adapter aux évolutions des modèles de programmation parallèle — le domaine est toujours en évolution rapide — ainsi qu'à celles des techniques de visualisation. La conception de Pajé a été très influencée par l'objectif d'extensibilité : architecture modulaire, flexibilité des modules de visualisation et généricité du module de simulation [9, 27, 67].

L'architecture de Pajé est un graphe de composants modulaires dont les seules relations de dépendances sont exprimées par des protocoles de communication bien spécifiés. Cette structuration permet le remplacement ou l'ajout d'un composant, à condition que le nouveau composant respecte le protocole de communication avec les composants auxquels il est connecté. En outre, la plupart des composants sont indépendants de la sémantique du modèle de programmation visualisé. Ces composants sont paramétrés par une description de la hiérarchie des types des objets visualisés. Cette hiérarchie peut être changée par les commandes de « zoom » dont l'effet est l'insertion d'un filtre dans le graphe de composants : le rôle du filtre est de transformer la hiérarchie des types en entrée du composant en une autre hiérarchie, qui est passée aux composants suivants du graphe ; la hiérarchie des types peut également être modifiée par des commandes placées dans le programme visualisé et qui permettent d'instancier Pajé pour un modèle de programmation donné : il s'agit de la généricité de Pajé [27].

3.4.1 GRAPHE FLOT DE DONNÉES DE COMPOSANTS

Pour faciliter l'extensibilité de l'environnement et la réutilisation de ses modules, Pajé a été réalisé sous forme de composants interconnectés sur le modèle de l'environnement Pablo [125]. Chaque composant (module) est un objet indépendant, qui communique avec les autres à travers des ports de communication, de façon à constituer un graphe flot de données, dont les sommets sont les modules d'analyse et les arêtes sont les liens de communication. Les données qui circulent sur les arêtes sont des objets qui représentent les entités du programme analysé. L'indépendance des modules facilite le développement d'un environnement comportant des modules réutilisables. Un environnement de visualisation particulier est alors construit en connectant certains de ces composants.

La figure 3.4 représente un exemple de diagramme flot de données simple comportant un lecteur de traces, un simulateur, un module de statistiques et un module de visualisation (diagramme espace-temps). Le disque contient la trace, qui est lue et interprétée par le lecteur de traces, qui produit des objets qui représentent les événements qui y sont stockés. Le simulateur reçoit ces événements, simule les activités du programme et produit des objets qui représentent les états des fils d'exécution, les communications, les états des sémaphores. Ces objets sont utilisés par le module de statistiques ainsi que par le module de visualisation.

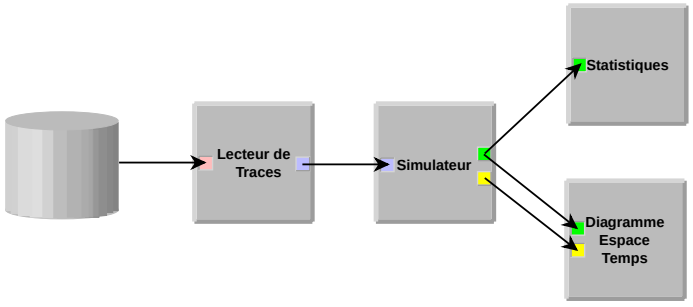

FIG. 3.4 – Exemple de diagramme d'analyse

3.4.2 FLEXIBILITÉ DES MODULES DE VISUALISATION

Les composants de visualisation de Pajé sont totalement indépendants de tout modèle de programmation parallèle. Avant une visualisation, ils reçoivent une description des types des objets à visualiser ainsi que des relations entre ces objets et de la façon de visualiser ces objets (voir figure 3.5). Les seules contraintes portent sur la nature des relations de type entre les objets visualisés et la possibilité de placer chacun de ces objets dans l'échelle de temps de la visualisation. C'est cette description hiérarchique qui est utilisée par les composants de visualisation pour obtenir des autres modules de Pajé la description des objets à visualiser.

La description des types d'objets à visualiser peut être changée pour s'adapter à un nouveau modèle de programmation ou durant une visualisation, pour changer la représentation visuelle d'un objet, suite à une requête d'un utilisateur. Cette propriété, qui rend les composants de visualisation extrêmement flexibles, est également utilisée par les modules de filtrage : quand ces derniers sont insérés dynamiquement dans un diagramme d'analyse tel que celui de la figure 3.4, par exemple pour « zoomer » d'une représentation détaillée à une représentation globale, ils envoient tout d'abord une description de la nouvelle hiérarchie de types d'objets à visualiser au composants de visualisation du graphe flot de données.

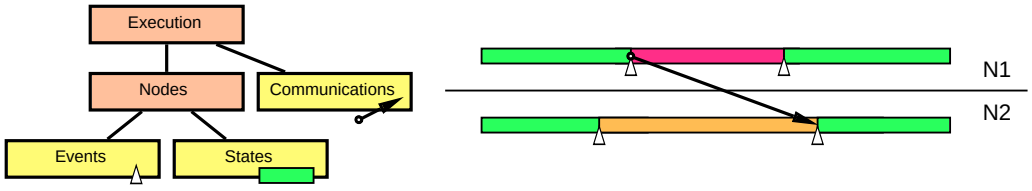

FIG. 3.5 – Utilisation d'une hiérarchie de type simple

Les hiérarchies de types utilisables dans Pajé peuvent être décrites par des arbres dont les feuilles sont appelées entités et les nœuds intermédiaires des *containers*. Les entités sont des objets élémentaires qui peuvent être visualisés tels que des événements, des états de fils d'exécution ou des communications. Les *containers* sont des objets d'un niveau d'abstraction supérieur, utilisés pour structurer la hiérarchie des types (voir figure 3.5). Par exemple tous les événements se produisant dans le fil 1 du nœud 0 appartiennent au *container* « fil-1-du-nœud-0 ».

3.4.3 GÉNÉRICITÉ DE PAJÉ

La structure modulaire de Pajé ainsi que l'indépendance de ses composants par rapport à tout modèle de programmation facilite le développement de nouveaux modules ou l'extension des modules existants par les « développeurs d'outils ». Ces caractéristiques seules ne suffiraient cependant pas à permettre l'utilisation de Pajé pour la visualisation de modèles de programmation variés si le module de simulation, au coeur de l'outil Pajé, était trop étroitement lié à un modèle de programmation donné. Au contraire, Pajé offre la possibilité aux programmeurs d'applications parallèles de définir ce qu'ils aimeraient visualiser et comment les nouveaux objets visualisés doivent être représentés par Pajé. Pour ce faire, la hiérarchie des types des objets à visualiser peut être définie par le programmeur d'application en insérant des définitions et des commandes **dans le programme à tracer et visualiser**. Ces définitions et commandes sont encapsulées dans des « événements utilisateurs » et donc interceptées par le traceur, ce qui permet de les transmettre au module de simulation par l'intermédiaire de la trace. Le module de simulation utilise ces définitions pour construire une nouvelle arborescence de types de données, définissant les relations entre objets visualisés, cette arborescence étant ensuite passée aux modules de visualisation de Pajé (voir [27, 67] pour plus de détails). À notre connaissance, Pajé est le seul outil de visualisation existant à offrir une telle flexibilité, aussi facile à mettre en œuvre.

La généricité de Pajé a permis de visualiser des exécutions de programmes ATHAPASCAN-1, sans qu'il soit nécessaire de faire aucun nouveau développement

dans Pajé. En insérant quelques instructions dans l'implémentation de ATHAPASCAN-1, il a été possible de représenter graphiquement les durées des différentes activités mises en œuvre par l'exécution d'un programme ATHAPASCAN-1 : calcul du programme proprement dit mais aussi gestion et ordonnancement du graphe de tâches défini par l'utilisateur.

3.5 INTERACTIVITÉ

Comme nous l'avons indiqué l'interactivité est une propriété importante de Pajé puisque, combinée avec les possibilités de « zoom » de l'outil, elle en assure la « scalabilité ». Nous avons également indiqué que le déplacement dans le temps était entièrement interactif. Une autre forme d'interactivité permet de faire la liaison entre plusieurs objets représentés : par exemple, le déplacement du pointeur de souris au dessus de la représentation d'un état bloqué de fil d'exécution a pour effet la surbrillance de l'état correspondant du sémaphore sur lequel il est bloqué, permettant à l'utilisateur un rapprochement immédiat ; il est également possible, à partir de la représentation visuelle d'un événement, de faire apparaître la ligne correspondante du programme source en cours de visualisation ; réciproquement, la sélection d'une ligne du programme source dans le « butineur » de code source fait apparaître en surbrillance les événements générés par cette ligne.

La structuration de l'environnement en composants d'un graphe flot de données se prête bien à la réalisation de modules qui, comme c'est le cas de l'outil Pablo [125], n'ont besoin d'accéder qu'une seule fois à chaque information issue de la trace, comme par exemple un module faisant des statistiques globales d'utilisation des nœuds. En revanche, certains modules ont besoin d'accéder simultanément à une quantité importante d'informations, par exemple pour donner une vision historique de la trace (voir section 3.3.1 et figure 3.2). Si un module de ce type reçoit indépendamment les entités transitant sur le graphe flot de données, il devra redemander ou stocker des informations déjà traitées, chaque fois que l'utilisateur désirera revoir un état précédent.

L'interactivité suppose donc que les objets visualisés soient stockés en mémoire. C'est le cas de l'outil *upshot* qui est interactif mais doit stocker l'ensemble de la trace en mémoire [124]. Cette propriété est difficilement compatible avec la « scalabilité » qui a pour conséquence de visualiser un très grand nombre d'objets. La solution adoptée est la définition d'une **fenêtre d'observation**, gérée par un composant particulier du graphe. La **fenêtre de visualisation** visible à l'écran est un sous-ensemble de la fenêtre d'observation et elle se déplace dans le temps en réponse aux requête des utilisateurs. Le glissement vers le futur de la fenêtre de

visualisation, hors des limites de la fenêtre d'observation, a pour conséquence la lecture de nouveaux événements du fichier de traces et éventuellement le stockage d'un état en mémoire. Le glissement vers le passé de la fenêtre de visualisation hors des limites de la fenêtre d'observation a pour conséquence la relecture du dernier état sauvegardé et la re-simulation jusqu'à la date recherchée.

La fenêtre d'observation est gérée par un module particulier appelé **encapsuleur**, qui est inséré dans le graphe flot de données (voir figure 3.6) à la suite du module de simulation. Le module encapsuleur reçoit les objets individuels qui sortent du simulateur et les encapsule dans un objet complexe qui représente donc la fenêtre d'observation courante. Tout accès aux données élémentaires est réalisé à travers cet objet, qui peut être interrogé pour obtenir des informations globales relatives à la fenêtre courante, sélectionner des parties de la fenêtre ou des informations spécifiques relatives à un objet individuel. La fenêtre glisse dans le temps par addition ou retrait d'objets.

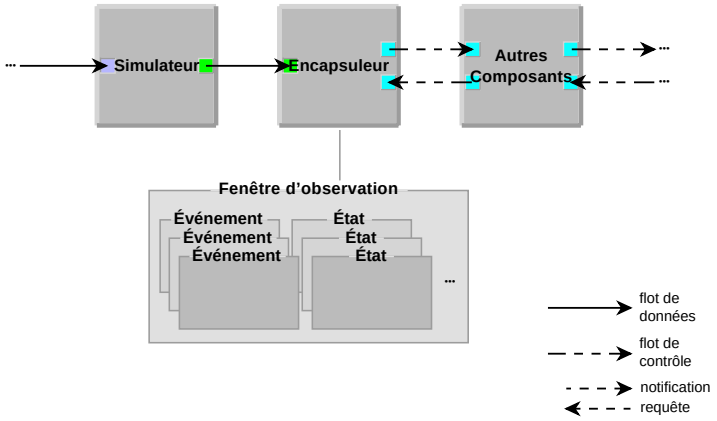

FIG. 3.6 – Encapsuleur et fenêtre d'observation

L'encapsuleur organise l'accès par d'autres composants à toutes les entités produites par le simulateur, en utilisant de la fenêtre d'observation.

Même si l'utilisation de la fenêtre d'observation permet de limiter le nombre d'objets gérés en mémoire à un instant donné, ce nombre n'en demeure pas moins considérable. Sachant que le moindre mouvement de souris nécessite de recalculer la visualisation courante, à cause d'un glissement dans le temps ou pour faire apparaître certains objets en surbrillance, la recherche des objets devant apparaître à l'écran a due être optimisée. La difficulté provient des entités de longue durée de vie — communications, par exemple. Une organisation sophistiquée de ce type de données, à l'intérieur de la fenêtre d'observation, permet de rechercher rapidement

celles qui doivent apparaître dans la visualisation courante (voir [9, 67]).

3.6 « SCALABILITÉ »

L'utilisation de la fenêtre d'observation décrite dans le §3.5 offre une forme de « scalabilité temporelle » en limitant le nombre de données présentes en mémoire durant la visualisation d'exécutions de longue durée. Cependant, les principales visualisations de Pajé ont été choisies pour l'aide qu'elles apportent aux utilisateurs, bien qu'elles ne soient pas `a priori` « scalables » (voir section 3.3) : en effet, l'espace utilisé pour les représenter croît linéairement avec leur nombre, si l'on représente l'ensemble des fils d'exécution mis en œuvre par une exécution de programme. Afin de permettre la visualisation d'exécutions mettant en œuvre un nombre important de fils d'exécution, Pajé permet de regrouper les informations les concernant, soit par nœud, soit par groupe de nœuds. En outre, un module de placement tire parti de la brièveté de l'exécution de certains fils d'exécution.

3.6.1 REGROUPEMENTS DE FILS D'EXÉCUTION OU DE NŒUDS

Le module de regroupement génère, à partir de l'activité des fils d'exécution de la fenêtre courante, une vision plus globale de cette fenêtre qui ne contient qu'un seul fil d'exécution par nœud ou par service créé sur chaque nœud et dont les états peuvent être «en exécution» ou «en attente». Dans la vue groupée, il est plus facile de détecter si un nœud n'a pas assez de travail (voir figure 3.8(c)). Le passage de la vue détaillée à la vue groupée et réciproquement permet un effet de «zoom» sur la visualisation.

En outre, de nouvelles représentations plus synthétiques des exécutions ont été développées et le regroupement de plusieurs fils d'exécution ou de plusieurs nœuds en une représentation unique plus abstraite ont été facilités (voir figure 3.7).

3.6.2 PLACEUR DE FILS D'EXÉCUTION

Les programmes à base de fils d'exécution communicants créent typiquement un grand nombre de fils d'exécution de durée relativement courte. L'objectif de ce filtre est de permettre une meilleure utilisation de la place disponible sur l'écran et une meilleure visualisation des fils, en réutilisant les positions des fils déjà terminés. Chaque fil d'exécution qui commence est placé en fonction des positions

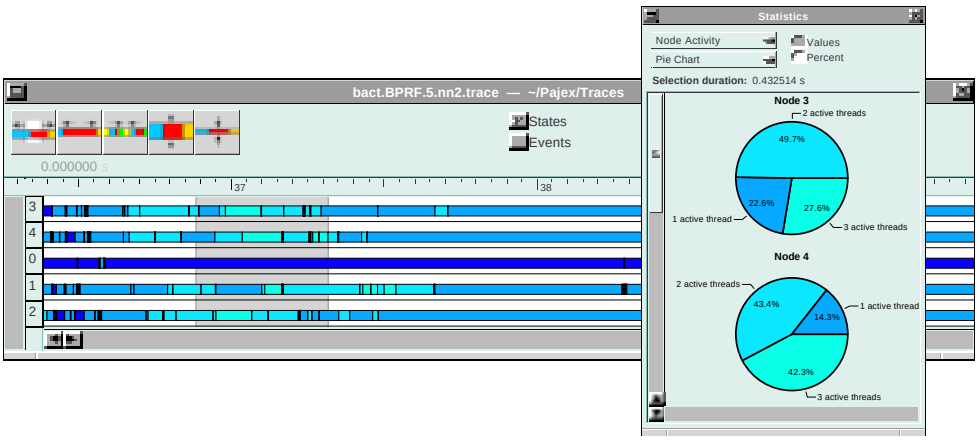

FIG. 3.7 – Utilisation de la « scalabilité » pour identifier l'utilisation des CPU.

disponibles. Quand un fil se termine, sa place est considérée comme disponible. Les figures 3.8(a) et 3.8(b) montrent un ensemble de fils d'exécution avant et après l'action du placeur.

Le placeur supporte la notion de service d'ATHAPASCAN-0[92] ; un fil d'exécution ne peut réutiliser que la position d'un autre fil d'exécution appartenant au même service du même nœud. Ainsi, les fils d'exécution d'un même service restent groupés, pour faciliter leur identification par l'utilisateur.

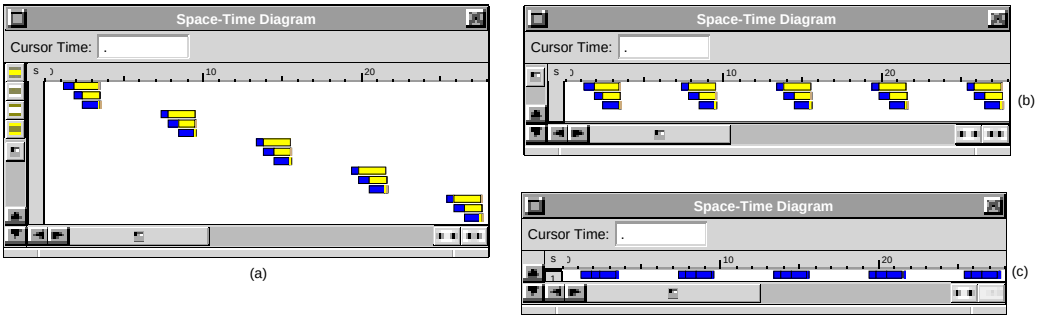

FIG. 3.8 – Exemples de filtres

3.7 PERSPECTIVES

L'une des principales motivations au développement de Pajé a été l'inadéquation des outils existants pour visualiser les exécutions de programmes parallèles à

base de fils d'exécution communicants. Ce problème peut être considéré comme résolu : Pajé permet de visualiser l'exécution de programmes par des fils d'exécution dont le nombre varie dynamiquement ; l'outil adapte dynamiquement l'espace alloué aux nœuds, récupère l'espace laissé libre par les fils d'exécution qui se sont terminés, permettant ainsi la visualisation des exécutions considérées. Pajé comporte en outre un certain nombre de caractéristiques remarquables telles que l'interactivité et l'extensibilité. L'interactivité permet la navigation dans le temps, la mise en évidence de certaines interactions entre objets visualisés ou encore la liaison entre les objets visualisés et les programmes sources. L'extensibilité permet aux experts de rajouter de nouvelles fonctionnalités mais également aux utilisateurs de « programmer » des visualisations au moyen de commandes insérées dans les fichiers de trace : cette fonctionnalité permet d'utiliser Pajé pour de nombreux types de visualisations scientifiques.

La perspective d'utilisation de Pajé « en ligne », pour le débogage, est présentée dans le chapitre 4. Une autre perspective est son utilisation pour visualiser des exécutions parallèles sur un grand nombre de processeurs. A priori, Pajé dispose de toutes les fonctionnalités qui doivent lui permettre ce type de visualisation et en particulier la possibilité d'obtenir dynamiquement un effet de zoom en utilisant des filtres (voir §3.6). Cependant seule une utilisation effective, par exemple dans le contexte du projet Grappe 200 PCs, permettra de valider les solutions existantes, d'en tracer les limites et de proposer les extensions qui pourraient s'avérer nécessaires pour développer des visualisations plus synthétiques.

La principale perspective de recherche offerte par Pajé concerne certainement l'exploitation de la flexibilité de l'outil pour des visualisations variées. Le contexte est en premier lieu celui de la programmation parallèle et des tests ont déjà permis de montrer que Pajé permettait de visualiser des exécutions de programmes utilisant des bibliothèques de communication aussi différentes que ATHAPASCAN-0 et ATHAPASCAN-1. Une étude est actuellement en cours, en collaboration avec le CNET, pour utiliser Pajé pour la visualisation d'exécutions de programmes Java ; sans modification de Pajé il est déjà possible de montrer l'exécution de fils d'exécution parallèles, l'emboitement des appels de méthodes, etc. Une autre étude vient de démarrer pour aider à l'administration d'une grappe de grande taille en représentant graphiquement des informations telles que la charge des processeurs en utilisant Pajé. À travers tous ces travaux, il sera possible d'évaluer et d'étendre si besoin est le langage de commande et le simulateur générique de Pajé, de proposer des extensions à cette forme de généricité pour élargir le spectre d'utilisation de l'outil. Ces études permettront également d'enrichir les fonctionnalités de Pajé : nouveaux filtres, nouvelles visualisations. Le travail d'évaluation des possibilités de Pajé ne vient que de commencer et il semble que ses perspectives soient extrêmement prometteuses.

4

VERS UN ENVIRONNEMENT INTÉGRÉ DE MISE AU POINT

« I have a dream ». M. L. King.

L'utilisation du mécanisme de ré-exécution comme mécanisme de base du débogage est une « vieille » idée qui apparaît par exemple dans ParaRex [106], qui fait coopérer le débogueur de l'iPSC2 avec l'outil de visualisation ParaGraph. L'environnement de programmation parallèle expérimental Annai intègre également outils de débogage et de mesure de performances et visualisation, pour plusieurs modèles de programmation dont HPF et MPI [120]. Cependant cette intégration n'est toujours pas réalisée dans les débogueurs commerciaux existants. TotalView ne comporte pas de mécanisme de ré-exécution ; il permet toutefois la visualisation interactive des données manipulées par les applications, durant les phases de mise au point [121], ce qui peut se révéler utile pour la mise au point des applications de simulation par exemple.

Dans le contexte de l'environnement ATHAPASCAN, l'objectif est d'intégrer tous les outils de mise au point dans un environnement homogène, à savoir les outils de ré-exécution, de traçage et de visualisation avec un débogueur visuel parallèle. En phase de mise au point, toutes les exécutions s'effectuent en mode *record*. En cas d'erreur de logique il est possible de ré-exécuter l'application fautive de façon déterministe relativement à l'exécution initiale. Durant cette ré-exécution on

peut éventuellement tester la présence de conditions de concurrence sur les données en utilisant un système tel que JiTi [115]. Cette ré-exécution est susceptible de mettre en œuvre un débogueur symbolique parallèle et peut-être visualisée en utilisant Pajé. Pajé fournit ainsi une représentation de haut niveau de l'exécution fautive tandis que le débogueur symbolique permet d'examiner dans le détail le comportement du programme exécuté.

La réalisation d'un tel environnement intégré pose des problèmes pour le traçage et la visualisation en ligne ainsi que pour la coopération entre les outils.

4.1 TRAÇAGE EN LIGNE

Pour visualiser une exécution à l'aide de Pajé, il est nécessaire de collecter des traces. Actuellement le traceur est conçu pour limiter autant que possible la perturbation du programme observé, ce qui a pour conséquence de nécessiter le traitement `post mortem` des traces par des outils d'analyse. Le but de ces traitements est de calculer des dates globales cohérentes et de permettre la mise en correspondance des événements de communication tels que l'émission et la réception d'un message [16].

Une première maquette a permis d'explorer la réalisation d'un traceur en ligne, dont les événements sont sérialisés dans chaque nœud puis transmis à l'extérieur au fur et à mesure de leur production [126].

4.1.1 DATATION

Le calcul des dates est nécessité par l'absence de dispositif physique de datation globale dans les systèmes observés. Chaque nœud dispose de sa propre horloge et les dates locales ne sont pas forcément cohérentes en raison du décalage initial et de la dérive entre les horloges des nœuds du système. Ainsi, le traceur ATHAPASCAN-0 utilise le mécanisme de datation globale logicielle SBA (*Sample Before and After*) défini par É. Maillet [80]. L'horloge d'un nœud du système est choisie comme horloge de référence. Des mesures sont pratiquées avant et après l'exécution du programme tracé, afin de calculer le décalage initial et la dérive entre l'horloge de référence et les autres horloges du système. Toutes les dates mesurées durant l'exécution tracée sont ensuite recalées `post mortem` en utilisant les valeurs de décalage et de dérive déduites des mesures.

Le traçage en ligne implique le calcul en ligne de dates globales cohérentes. Il faudra donc calculer les décalages et dérives des horloges locales en fonction

de mesures faites uniquement avant l'exécution, ce qui est moins précis que la technique de É. Maillet (voire [80] pour plus de détails). La correction des dates locales devra se faire en même temps que l'enregistrement des événements, ce qui perturbera plus l'exécution observée que la solution `post mortem` utilisée actuellement.

En fait le calcul de la date est compliqué considérablement par les interactions avec les outils de visualisation et de débogage. Dans de nombreuses situations, le temps physique d'un événement mesuré n'est pas directement utilisable. C'est le cas si l'outil de visualisation est utilisé seul : la production des événements par le traceur est alors asservie à leur consommation par l'outil de visualisation, ce qui a pour effet de ralentir, pas forcément de façon uniforme, l'ensemble des fils d'exécution de l'application. La situation peut être encore plus compliquée si on utilise en outre un débogueur symbolique. Dans ce cas, si un fil d'exécution particulier est exécuté pas à pas, le temps physique enregistré dans les événements tracés n'a plus de sens.

É. Leu traite ce phénomène en recalant les dates de l'exécution visualisée à l'aide de dates enregistrées durant l'exécution *record* [64]. En ce qui nous concerne, une telle solution impliquerait de modifier les techniques utilisées pour la ré-exécution déterministe afin de pouvoir collecter des dates durant les exécutions *record* et *replay* et d'utiliser ces dates pour dater les événements enregistrés pour la visualisation. Une étude est nécessaire pour déterminer si une modification de ce type est compatible avec les choix d'implémentation du mécanisme de ré-exécution déterministe de programmes à base de fils d'exécution communicants : seuls les ordres des synchronisations et des messages conflictuels sont enregistrés d'une part et les enregistrements sont extrêmement compacts d'autre part (voir le §2.4). Il faut donc étudier si la datation des événements enregistrés durant la phase *record* serait suffisante pour permettre de recaler les dates des événements enregistrés pour la visualisation. Par ailleurs, l'enregistrement d'un événement de synchronisation (deux octets) est beaucoup plus compact que celui d'une date physique (huit octets pour *gettimeofday*).

Une autre possibilité que nous comptons explorer consiste à associer aux événements une date logique comme par exemple la valeur courante de l'horloge de Lamport [78]. Une visualisation simplifiée des événements de communication, ordonnés en utilisant une horloge de Lamport, est déjà proposée comme support débogage par l'outil Atempt de l'environnement MAD [76]. Dans le cas de Pajé, il semble en effet que le calcul des dates de Lamport des événements puisse être fait par le module de simulation, à partir des traces collectées pour la mesure de performances. Si cette hypothèse est vérifiée, aucune surcharge des traces collectées pour la ré-exécution ne sera nécessaire, ce qui permettra d'en garder la compacité, indispensable pour pouvoir envisager la ré-exécution déterministe de programmes de grande durée. L'étude devra néanmoins déterminer si les visualisations utilisant

des dates logiques s'avèrent utilisables en pratique. Nous pensons en effet aux fils d'exécution peu ou pas synchronisés, pour lesquels des événements se produisant à des dates physiques très différentes auront des dates logiques proches. Par ailleurs, l'utilisation de dates logiques impliquera des modifications dans l'environnement de visualisation, actuellement conçu pour représenter des dates physiques.

4.1.2 APPARIEMENT DES ÉVÉNEMENTS DE COMMUNICATION

Les choix d'implémentation de ATHAPASCAN-0 — utilisation des standards `Pthreads` et `MPI` sans modification — ne permettent pas de faire simplement l'appariement entre un événement d'émission et l'événement correspondant de réception [16]. Le traceur ATHAPASCAN-0 utilise le fait que les émissions de messages sont sérialisées par ATHAPASCAN-0 d'une part et que `MPI` délivre les messages dans leur ordre d'émission d'autre part pour faire cet appariement `post mortem`, en utilisant des enregistrements de compteurs de messages émis et reçus. La transformation du traceur pour le mettre en ligne implique de faire l'appariement en ligne également.

4.1.3 INTRUSION DUE AU TRAÇAGE

L'intrusion causée par le traceur en ligne sera supérieure à celle du traceur existant. Une des raisons a déjà été indiquée : il s'agit du calcul des dates globales lors de l'enregistrement des événements. Une autre raison sera un accroissement de la concurrence entre le traceur et l'application pour l'utilisation des ressources du système parallèle, ne serait-ce que pour transporter les traces hors du système : dans le traceur actuel, les traces sont stockées en mémoire et si nécessaire sur les disques locaux des nœuds jusqu'à la fin de l'exécution. Dans un traceur en ligne, le transport des traces à l'extérieur du système parallèle se fera en concurrence avec les messages du programme étudié. Pour limiter les effets de l'intrusion sur l'exécution, on peut effectuer ce traçage en ligne durant une ré-exécution, conservant ainsi durant la visualisation en ligne la même causalité que durant une exécution effectuée en mode *record* et donc peu perturbée.

4.2 VISUALISATION EN LIGNE

La visualisation en ligne d'exécutions parallèles ATHAPASCAN-0 pose des problèmes relatifs à la gestion du temps ainsi que des problèmes de coopération entre outils pour le contrôle de l'application. Le problème de gestion du temps concerne également le traçage et a déjà été largement évoqué au §4.1.1. L'environnement de visualisation sera plus ou moins sollicité en fonction des solutions adoptées par le traceur pour la datation des événements. L'utilisation en ligne de Pajé nécessitera également une adaptation de ses modules de lecture de traces et de simulation, afin de pouvoir traiter les événements au fur et à mesure de leur production, ainsi qu'il a été expliqué au §4.1.

4.3 DÉBOGUEUR VISUEL

Le type de débogueur qui nous intéresse est une extension des débogueurs séquentiels « classiques » [18, 121]. L'utilisation de ce type de débogueur impose au programmeur d'être capable de comprendre l'état courant de l'exécution et de contrôler les fils d'exécution. La fourniture d'une interface visuelle constitue une tentative de prise en compte de la complexité d'exécution de programmes parallèles comportant un grand nombre de fils d'exécution évoluant dynamiquement.

Le principal travail entrepris dans ce domaine concerne la réalisation d'une interface graphique pour débogueur parallèle. Cette interface graphique, appelée PADI, est en cours de développement par une doctorante de l'Universidade Federal do Rio Grande do Sul à Porto Alegre, Denise Stringhini, dans le cadre d'une bourse « sandwich » [43]. L'objectif est de faciliter la navigation dans l'ensemble des états de processus, de fils d'exécution et des valeurs de variables que le débogueur parallèle rend possible d'examiner. La principale originalité de PADI est de proposer des mécanismes sophistiqués et flexibles pour regrouper les fils d'exécution en fonction de critères prédéfinis ou en fonction des besoins exprimés par l'utilisateur. L'appartenance à un groupe peut être une notion dynamique même si le groupe est prédéfini : on peut ainsi considérer le groupe des fils d'exécution exécutables ou celui des fils d'exécution actifs. L'interface PADI sera validée en la connectant au débogueur parallèle PDBG [97] développé à l'Universidade Nova de Lisboa.

L'intégration dans un même environnement d'un débogueur symbolique parallèle, contrôlé par une interface graphique telle que PADI, et de l'outil de visualisation Pajé pose des problèmes de coopération entre les outils. En effet, il serait intéressant de pouvoir contrôler l'exécution parallèle à partir de la représentation de haut niveau fournie par Pajé et donc par exemple de permettre la pose éventuelle d'un point d'arrêt, l'examen des données d'un fil d'exécution ou d'une

communication ainsi que l'exécution sélective de quelques fils d'exécution, etc. Une étude fine est nécessaire pour déterminer le type de coopération souhaitable entre les outils, afin de faciliter autant que possible la tâche de l'utilisateur d'une part et d'éviter les interblocages d'autre part. On peut envisager d'utiliser l'infrastructure DAMS [97], développée pour faciliter la coopération entre les outils et utilisée pour le débogueur parallèle PDBG. Un projet de coopération INRIA-ICCTI a été mis en place entre notre équipe et l'Universidade Nova de Lisboa où ont été développés DAMS et PDBG.

4.4 CONCLUSION

Il semble que la mise à disposition d'un environnement intégrant outils de débogage et de visualisation puisse faciliter notablement la mise au point de programmes parallèles à base de fils d'exécution communicants. L'intégration devrait permettre de conserver une vue cohérente des applications visualisées tout au long du processus de mise au point.

Nous disposons actuellement des briques de base principales pour la construction d'un tel environnement : mécanisme de ré-exécution déterministe, outil de visualisation « scalable », interactif et extensible. Une étude est en cours pour définir une interface graphique pour le débogueur. L'intégration semble donc possible mais pose cependant un certain nombre de problèmes pour le traçage et la visualisation en ligne ainsi que pour la coordination entre outils. Par ailleurs, pour que l'environnement intégré puisse remplir son rôle, il est absolument indispensable que cette intégration soit pleinement réussie, à savoir qu'il fonctionne parfaitement et soit simple d'utilisation. Ces deux propriétés ont un coût en temps de conception et de développement : il faut pouvoir dépasser le stade de la maquette qui fonctionne « de temps en temps » ainsi que changer une interface inadéquate autant de fois qu'il est nécessaire pour obtenir un service satisfaisant. L'expérience montre en effet que si les utilisateurs se plaignent d'un manque d'outils alors qu'un grand nombre ont été développés, c'est plus vraisemblablement parce que la majorité des outils existants sont jugés inadéquats que par manque d'outils [90].

5

CONCLUSION

« **Parallèle :** On ne doit choisir qu'entre les suivants : César et Pompée, Horace et Virgile, Voltaire et Rousseau, Napoléon et Charlemagne, Goethe et Schiller, Bayard et Mac-Mahon... » G. Flaubert, *Dictionnaire des idées reçues*.

Pour conclure ce mémoire, nous faisons un bref bilan des travaux qui y sont présentés ainsi que de leurs perspectives immédiates ou à plus long terme.

L'objectif commun des travaux présentés dans ce document est d'aider les programmeurs à mettre au point leurs applications parallèles. Le type d'aide qui est considéré est de permettre aux programmeurs d'observer, aussi précisément et fidèlement que possible, le comportement à l'exécution des applications qu'ils écrivent. On suppose qu'à partir d'une observation exacte et précise, les utilisateurs seront à même de prendre les décisions adéquates permettant de corriger les erreurs de logique et de performances de leurs programmes.

Ces travaux trouvent un terrain d'application privilégié dans le contexte du projet Apache. Dans le but de permettre le recouvrement des communications par les calculs et de faciliter la programmation d'applications irrégulières, un noyau exécutif parallèle appelé ATHAPASCAN-0 a été défini, à base de fils d'exécution communicants. Les travaux présentés ici concernent essentiellement la mise au point des applications utilisant un modèle de programmation de ce type même si leur cadre d'application déborde largement le projet Apache.

5.1 PRINCIPAUX ACQUIS

Deux problèmes ont été abordés principalement : la maîtrise de l'indéterminisme d'une part et la visualisation d'exécutions parallèles d'autre part.

Les programmes parallèles à base de fils d'exécution communicants sont susceptibles de présenter un non déterminisme interne, cette caractéristique permettant par exemple une adaptation dynamique à l'environnement (répartition dynamique de charge). L'indéterminisme interdit le débogage cyclique des applications, tel qu'il se pratique habituellement pour les applications séquentielles. Pour permettre le débogage cyclique des applications, des mécanismes de ré-exécution déterministes ont été développés pour le modèle procédural parallèle et pour le modèle des fils d'exécution communicants.

Dans le cas du modèle procédural parallèle, un mécanisme de ré-exécution déterministe indépendant de la couche de communication a été défini, prouvé formellement et implémenté. Ce mécanisme permet de limiter le nombre d'enregistrements nécessaires à la ré-exécution des programmes. Il est applicable à d'autres modèles de programmation tel le modèle client-serveur.

Un mécanisme de ré-exécution a été défini pour le modèle des fils d'exécution communicants, en collaboration avec des chercheurs de l'université de Gand. Il combine leur technique optimisée, pour prendre en compte l'indéterminisme induit par l'accès des fils d'exécution d'un même nœud à des objets de synchronisation, avec une technique classique pour traiter l'indéterminisme induit par la réception de messages dont la source n'est pas spécifiée et enfin une technique originale pour prendre en compte l'indéterminisme induit par la combinaison de primitives de communications non bloquantes et de tests. Ce mécanisme de ré-exécution est applicable à de nombreux modèles de programmation et en particulier aux bibliothèques de communication standards que sont PVM et MPI.

Une autre application de la ré-exécution déterministe est la lutte contre les l'effet de sonde dans le cadre de la mesure de performances. L'observation d'un système en perturbe habituellement le fonctionnement. Ce classique effet de sonde est particulièrement important lorsqu'il s'agit d'exécutions de programmes parallèles puisqu'il est susceptible de changer la relation de causalité entre les événements des programmes observés. Les travaux réalisés dans ce domaine concernent la correction de l'intrusion due à la prise de traces des applications parallèles, pour en mesurer les performances. Dans ce contexte, des algorithmes de correction `post mortem` des traces ont été définis, pour restituer le comportement qu'auraient eu les programmes observés si l'observation n'avait pas perturbé leur fonctionnement. Ces algorithmes ne fonctionnent que si l'observation n'a pas modifié la

relation de causalité entre les événements tracés. L'utilisation de la ré-exécution déterministe a été expérimentée pour traiter cette situation, dans le cas du modèle procédural parallèle.

L'objectif des outils de visualisation est de faciliter l'observation des exécutions parallèles en présentant de façon aussi intelligible que possible la grande quantité d'information générée par ces exécutions. Les outils existants ne permettaient pas de visualiser un grand nombre de fils d'exécution créés et terminés dynamiquement et communiquant de façons variées. L'outil Pajé combine de façon originale la « scalabilité », l'interactivité et l'extensibilité. La « scalabilité » est fournie par la possibilité de passer d'un niveau d'abstraction à un autre. Cette possibilité n'est vraiment effective qu'en raison de l'interactivité qui permet d'effectuer ces effets de zoom dynamiquement, tout en offrant la possibilité de se déplacer dans le temps. L'extensibilité est la propriété qui permettra à Pajé de suivre facilement les évolutions des modèles de programmation mais aussi d'être utilisé pour visualiser l'exécution d'applications écrites dans d'autres langages tels que Java.

5.2 PERSPECTIVES

Les perspectives ouvertes par les travaux présentés dans ce manuscrit peuvent être divisées en perspectives directement induites et perspectives à plus long terme.

Les perspectives directes ont déjà été indiquées dans les chapitres du manuscrit. Elles ont trait principalement à l'intégration des outils de débogage et de visualisation au sein d'un environnement unifié. Les caractéristiques ainsi que les problèmes de conception et mise en œuvre de cet environnement ont été évoquées dans le chapitre 4. Ce travail ne semble pas poser `a priori` de difficulté scientifique majeure mais seulement nécessiter des développements techniques importants.

Une autre perspective qui a également été évoquée est le passage à l'échelle des outils, en particulier dans le contexte de grappes de plusieurs centaines de PCs. La collecte et l'analyse de traces — pour le ré-exécution déterministe et surtout pour la mesure de performances — devra gérer des volumes de données très importants. Les problèmes que posera cette gestion seront analogues à ceux que posera la gestion des résultats produits par ces architectures parallèles, cette analogie pouvant conduire au déploiement de solutions analogues. Ainsi il est vraisemblable qu'il sera nécessaire dans certains cas de recourir à un traitement parallèle des traces. Par ailleurs, l'outil de visualisation Pajé offre des filtres qui permettent un effet de zoom et donc la combinaison d'une observation globale, synthétique

des exécutions d'applications avec une observation détaillée du comportement de certains nœuds ou de certains fils d'exécution. Reste à savoir si ces visualisations se révéleront utiles lorsque le nombre de processeurs utilisés sera d'un ordre de grandeur supérieur à celui qui a été testé jusqu'à maintenant ? D'autres techniques de filtrage ou d'autres visualisations plus synthétiques se révéleront alors sans doute nécessaires.

Les outils mentionnés ci-dessus, tant pour la ré-exécution déterministe que la visualisation, ne concernent qu'une exécution particulière d'un programme parallèle. Si le programme a un comportement non déterministe, se pose la question de sa validation, même pour un jeu de données particuliers : il n'est pas possible de garantir que le passage par l'un des chemins d'exécution possibles ne mènera pas à une erreur. La problématique du test des applications parallèles est donc différente, plus complexe que celle du test des applications séquentielles. Une approche, développée à l'université de Gdansk [57], est d'identifier un certain nombre de situations (scénarios) remarquables, pour lesquelles on souhaite garantir un certain comportement des applications. Une méthodologie et des outils permettent alors de tester le comportement d'applications parallèles, pour les scénarios remarquables identifiés. Un projet d'action intégrée en coopération avec l'université de Gdansk, a été proposé dans le cadre du programme POLONIUM, afin d'adapter les techniques et outils de test au cas des programmes à base de fils d'exécution communicants.

BIBLIOGRAPHIE PERSONNELLE

Livres et monographies

[1] D. BARTH, J. CHASSIN DE KERGOMMEAUX, J.-L. ROCH, J. ROMAN (réd.), *ICaRE'97 : Conception et mise en œuvre d'applications parallèles irrégulières de grande taille*, CNRS, Décembre 1997, Actes de l'École du GDR CNRS Parallélisme, Réseaux et Systèmes.

[2] J. CHASSIN DE KERGOMMEAUX, P. HATCHER, L. RAUCHWERGER (réd.), *Parallel Computing for Irregular Applications*, *26*, 13-14, Elsevier, Octobre 2000, Special Issue, Parallel Computing.

[3] J. CHASSIN DE KERGOMMEAUX, D. TRYSTRAM (réd.), *European School on Parallel Programming Environments for High Performance Computing, ESPPE'96*, Alpe d'Huez, France, IMAG, Grenoble., avril 1996.

[4] J. CHASSIN DE KERGOMMEAUX (réd.), *Environnements d'Exécution de programmes parallèles*, HERMES, 1995. Numéro spécial de *Calculateurs Parallèles*, Vol. 7, No. 2.

[5] B. FOLLIOT, G. BERNARD, J. CHASSIN DE KERGOMMEAUX, C. ROUCAIROL (réd.), *Placement dynamique et répartition de charge : application aux systèmes parallèles et répartis*, Éditions INRIA, Juillet 1996, Actes de l'École Française de Parallélisme, Réseaux et Systèmes.

[6] B. FOLLIOT, J. CHASSIN DE KERGOMMEAUX, C. ROUCAIROL (réd.), *Le Placement Dynamique et la Répartition de Charge. Application aux Systèmes Répartis et Parallèles*, Université Pierre et Marie Curie, Paris, 1995, Journées de Recherche du GDR Parallélisme Réseaux et Systèmes du CNRS.

Thèses et habilitations à diriger des recherches

[7] J. CHASSIN DE KERGOMMEAUX, *Implémentation et Évaluation d'un Système Logique Parallèle*, thèse de doctorat, Université Joseph-Fourier Grenoble 1, 1989.

Articles publiés dans des revues

[8] J. CHASSIN DE KERGOMMEAUX, P. CODOGNET, « Parallel Logic Programming Systems », *ACM Computing Surveys 26*, 3, september 1994, p. 295–336.

[9] J. CHASSIN DE KERGOMMEAUX, B. DE OLIVEIRA STEIN, P. BERNARD, « Pajé, an interactive visualization tool for tuning multi-threaded parallel applications », *Parallel Computing 26*, 10, aug 2000, p. 1253–1274.

[10] J. CHASSIN DE KERGOMMEAUX, A. FAGOT, « Execution replay of parallel procedural programs », *Journal of Systems Architecture 46*, 10, Juillet 2000, p. 835–849.

[11] J. CHASSIN DE KERGOMMEAUX, P. ROBERT, « An Abstract Machine to implement efficiently OR-AND parallel Prolog », *Journal of Logic Programming 8*, 3, 1990.

[12] J. CHASSIN DE KERGOMMEAUX, M. RONSSE, K. DE BOSSCHERE, « Execution Replay and Debugging of Distributed Multi-Threaded Parallel Programs », *Computers and Artificial Intelligence*, 2000, Special issue on Testing and Debugging, article invité, à paraître.

[13] J. CHASSIN DE KERGOMMEAUX, « Mise au point d'applications parallèles à partir de traces d'exécution », *Calculateurs Parallèles 6*, 4, 1994, p. 21–26, Numéro spécial Actes de l'Ecole SPI Lyon juillet 94.

[14] E. MOREL, J. BRIAT, J. CHASSIN DE KERGOMMEAUX, « Cuts and side-effects in distributed-memory OR-parallel Prolog », *Parallel Computing 22*, 1997, p. 1883–1896.

Chapitres de livre

[15] J. BRIAT, M. FAVRE, C. GEYER, J. CHASSIN DE KERGOMMEAUX, « OPERA : OR-Parallel Prolog System on Supernode », *in : Implementations of Distributed Prolog*, P. Kacsuk et M. Wise (réd.), John Wiley and Sons, 1992, ch. 3, p. 45–63.

[16] J. CHASSIN DE KERGOMMEAUX, B. DE OLIVEIRA STEIN, P. WAILLE, « Mise au point d'applications parallèles irrégulières », *in : ICaRE'97 : conception et mise en oeuvre d'applications parallèles irrégulières de grande taille*, D. Barth, J. Chassin de Kergommeaux, J.-L. Roch, et J. Roman (réd.), CNRS, décembre 1997.

[17] J. CHASSIN DE KERGOMMEAUX, E. MAILLET, J.-M. VINCENT, « Monitoring Parallel Programs for Performance Tuning in Distributed Environments », *in : Parallel Program Development for Cluster Computing : Methodology, Tools and Integrated Environments*, J. Cunha et P. Kacsuck (réd.), Nova Science, 2000, ch. 6, To appear.

[18] J. CHASSIN DE KERGOMMEAUX, « Environnements et outils de mise au point », *in : Ordinateurs et calcul parallèles, ARAGO, 19*, OFTA, 5, rue Descartes, 75005 Paris, 1997, ch. 14, p. 255–271.

[19] P. KACSUCK, J. CHASSIN DE KERGOMMEAUX, E. MAILLET, J.-M. VINCENT, « The Tape/PVM Monitor and the PROVE Visualization Tool », *in : Parallel Program Development for Cluster Computing : Methodology, Tools and Integrated Environments*, J. Cunha et P. Kacsuck (réd.), Nova Science, 2000, ch. 14, To appear.

[20] E. MOREL, J. BRIAT, J. CHASSIN DE KERGOMMEAUX, C. GEYER, « Side-effects in PloSys OR-parallel Prolog on distributed Memory Machines », *in : Parallelism and Implementation of Logic and Constraint Logic Programming*, NOVA Science Publishers, Inc, New York, USA, 1999, ch. 9.

Communications à des manifestations scientifiques

[21] U. BARON, J. CHASSIN DE KERGOMMEAUX, M. HAILPERIN, M. RATCLIFFE, P. ROBERT, J.-C. SYRE, H. WESTPHAL, « The Parallel ECRC Prolog System PEPSys : An Overview and Evaluation Results », *in : Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, ICOT, 1988.

[22] P. BOUVRY, J. CHASSIN, M. DOBRUCKY, L. HLUCHY, E. LUQUE, T. MARGALEF, « Mapping and load balancing on distributed memory systems », *in : Proceedings of the Eight Symposium on Microcomputer and Microprocessor Applications*, A. Pataricza, E. Selényi, A. Somogyi (réd.), *1*, Budapest Technical University, Budapest, Hungary, p. 315–324, octobre 1994.

[23] P. BOUVRY, J. CHASSIN DE KERGOMMEAUX, D. TRYSTRAM, « Efficient Solutions for mapping parallel programs », *in : EuroPar'95 Parallel Processing*, S. Haridi, K. Ali, P. Magnusson (réd.), *LNCS*, *966*, Springer-Verlag, p. 379–390, août 1995.

[24] J. BRIAT, M. FAVRE, C. GEYER, J. CHASSIN DE KERGOMMEAUX, « Scheduling of OR-parallel Prolog on a Scalable, Reconfigurable, Distributed Memory Multiprocessor », *in : Parallel Architectures and Languages Europe, PARLE'91*, E. Aarts, J. van Leewen, M. Rems (réd.), Springer-Verlag, Lecture Notes in Computer Science No. 506, p. 385–402, Eindhoven, The Netherlands, 1991.

[25] J. CHASSIN DE KERGOMMEAUX, U. BARON *et al.*, « Performance Analysis of a Parallel Prolog : a Correlated Approach », *in : Proc. Parallel Architectures and Languages Europe, PARLE'89*, E. Odjik, M. Rem, J.-C. Syre (réd.), *LNCS*, 366, Springer-Verlag, p. 151–163, Eindhoven, 1989.

[26] J. CHASSIN DE KERGOMMEAUX, B. DE OLIVEIRA STEIN, P. WAILLE, « Mise au point d'applications parallèles irrégulières », *in : ICaRE'97 Conception et mise en œuvre d'applications parallèles irrégulières de grandes tailles*, G. thématiques de PRS Capa Rumeur Exec (réd.), École CNRS, 1997.

[27] J. CHASSIN DE KERGOMMEAUX, B. DE OLIVEIRA STEIN, « Pajé : an Extensible Environment for Visualizing Multi-Threaded Programs Executions », *in : Euro-Par 2000 Parallel Processing, Proc. 6th International Euro-Par Conference*, A. Bode, W. Ludwig, T. Karl, R. Wismüller (réd.), *LNCS*, *1900*, Springer, p. 133–140, 2000.

[28] J. CHASSIN DE KERGOMMEAUX, A. FAGOT, « Environnement d'aide à la mise au point de programmes parallèles », *in : Actes des $7^{èmes}$ rencontres sur le parallélisme,RenPar7*, p. 213–216, Faculté Polytechnique de Mons (Belgique), 1995.

[29] J. CHASSIN DE KERGOMMEAUX, M. RONSSE, K. DE BOSSCHERE, « MPL* : efficient record/replay of nondeterministic features of message passing libraries », *in : Proc. EuroPVM/MPI'99*, *LNCS, 1697*, Springer Verlag, p. 141–148, sep 1999.

[30] J. CHASSIN DE KERGOMMEAUX, J.-C. SYRE, H. WESTPHAL, « Implementation of a Parallel Prolog System on a Commercial Multiprocessor », *in : Proceedings European Conference on Artificial Intelligence*, Munich, août 1988.

[31] J. CHASSIN DE KERGOMMEAUX, « (Massively) Parallel Computers need Abstract Machine Models. », *in : Proc. of Workshop on Abstract Machine Models for Highly Parallel Computers*, University of Leeds, mars 1991.

[32] A. FAGOT, J. CHASSIN DE KERGOMMEAUX, « Optimized record-replay for RPC-based parallel programming », *in : Working conference on programming environments for massively parallel distributed systems*, IFIP, WG10.3, Birkh user Verlag, Basel, p. 347–352, Ascona, Switzerland, avril 1994.

[33] A. FAGOT, J. CHASSIN DE KERGOMMEAUX, « Formal and experimental validation of a low-overhead execution replay mechanism », *in : Euro-Par'95 Parallel Processing*, S. Haridi, K. Ali, P. Magnusson (réd.), *LNCS, 966*, Springer-Verlag, p. 167–178, août 1995.

[34] A. FAGOT, J. CHASSIN DE KERGOMMEAUX, « Systematic Assessment of the Overhead of tracing parallel programs », *in : Proceedings of the 4th Euromicro Workshop on Parallel and Distributed processing, PDP'96*, E. Zapata (réd.), IEEE/Computer Society Press, p. 179–186, Braga, janvier 1996.

[35] D. KRANZLMÜLLER, J. CHASSIN DE KERGOMMEAUX, C. SCHAUBSCHLÄGER, « Correction of Monitor Intrusion for Testing Nondeterministic MPI-Programs », *in : Proc. Euro-Par'99*, *LNCS, 1685*, Springer Verlag, p. 154–158, août 1999.

[36] E. MOREL, J. BRIAT, J. CHASSIN DE KERGOMMEAUX, C. GEYER, « Side-effects in PloSys OR-parallel Prolog on distributed-memory machines », *in : Proc. Workshop on Parallelism and Implementation Technologies for (Constraint) Logic Languages*, M. Carro, E. Pontelli (réd.), ESPRIT COMPULOG NET, p. 29–40, Bonn, 1996.

[37] E. MOREL, J. BRIAT, J. CHASSIN DE KERGOMMEAUX, « Overview of the PloSys OR-parallel logic programming system », *in : Proc. 1st Austrian-Hungarian Workshop on Distributed and Parallel Systems, DAPSYS'96*, KFKI-1996-09/M,N, Hungarian Academy of Sciences, Budapest, 1996.

[38] E. MOREL, J. BRIAT, J. CHASSIN DE KERGOMMEAUX, « PloSys : parallélisme OU et effets de bords sur système parallèle *sans* mémoire commune », *in : Proc. JFPLC'96, Journées Francophones de Programmation Logique et Programmation par Contraintes*, J.-L. Imbert (réd.), Hermès, p. 131–145, juin 1996.

[39] M. RONSSE, J. CHASSIN DE KERGOMMEAUX, K. DE BOSSCHERE, « Execution replay for an MPI-based multi-threaded runtime system », *in : Parallel Computing : Fundamentals and Applications*, E. H. D'Hollander, G. R. Joubert, F. J. Peters, H. J. Sips (réd.), *Proceedings of the International Conference ParCo99*, Imperial College Press, p. 656–663, 2000.

[40] M. RONSSE, K. DE BOSSCHERE, J. CHASSIN DE KERGOMMEAUX, « Execution replay and debugging », *in : Proc. of the Fourth International Workshop on Automated Debugging, AADEBUG 2000*, M. Ducassé (réd.), TUM-IRISA, p. 5–18, Munich, 2000. Invited talk.

[41] B. D. O. STEIN, J. CHASSIN DE KERGOMMEAUX, « Environnement de Visualisation de Programmes Parallèles Basés sur les Fils d'Exécution », *in : Actes des* $9^{ièmes}$ *Rencontres Francophones du Parallélisme, RenPar9*, D. Trystram (réd.), Lausanne, mai 1997.

[42] B. D. O. STEIN, J. CHASSIN DE KERGOMMEAUX, « Interactive Visualisation Environment of Multi-threaded parallel programs », *in : Parallel Computing : Fundamentals, Applications and New Directions, Proceedings of the International Conference ParCo97*, Elsevier, p. 311–318, 1998.

[43] D. STRINGHINI, P. NAVAUX, J. CHASSIN DE KERGOMMEAUX, « A Selection Mechanism to Group Processes in a Parallel Debugger », *in : Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'2000*, CSREA Press, p. 2575–2581, Las Vegas, USA, June 2000.

[44] F. TEODORESCU, J. CHASSIN DE KERGOMMEAUX, « Performance Evaluation of Parallel Programs by Multiple Phases of Execution », *in : Proceedings of the 3rd Romanian Conference on Open Systems OSE"95*, p. 11–14, Bucharest, Romania, novembre 1995.

[45] F. TEODORESCU, J. CHASSIN DE KERGOMMEAUX, « On Correcting the Intrusion of Tracing non Deterministic Programs by Software », *in : Proceedings of EuroPar'97*, Springer-Verlag, p. 94–101, août 1997.

[46] P. K. VARGAS, J. L. BARBOSA, C. F. R. FERRARI, D. N.AND GEYER, J. CHASSIN, « Distributed OR Scheduling with Granularity Information », *in : XII Simpósio Brasileiro de Arquitetura de Computadores e Process amento de Alto Desempenho*, p. 253–260, October 2000. São Pedro, SP.

[47] H. WESTPHAL, P. ROBERT, J. CHASSIN DE KERGOMMEAUX, J.-C. SYRE, « The PEPSys Model : Combining Backtracking AND- and OR- parallelism », *in : Proc. 4th Symposium on Logic Programming*, S. Haridi (réd.), IEEE/Computer Society Press, p. 436–448, San Francisco, septembre 1987.

Rapports de recherche

[48] J. CHASSIN DE KERGOMMEAUX, B. DE OLIVEIRA STEIN, « Pajé : an Extensible and Interactive and Scalable Environment for Visualizing Parallel Executions », Rapport de Recherche N° 3919, INRIA, april 2000, http ://www.inria.fr/RRRT/publications-fra.html.

[49] J. CHASSIN DE KERGOMMEAUX, M. RONSSE, K. DE BOSSCHERE, « Efficient execution replay for Athapascan-0 parallel programs », Rapport de recherche N° 3635, INRIA, mars 1999, http ://www.inria.fr/RRRT/publications-fra.html.

[50] A. FAGOT, J. CHASSIN DE KERGOMMEAUX, « Optimized execution replay mechanism for RPC-based parallel programming models », Rapport APACHE N° 18, LMC-IMAG Grenoble France, 1995, Disponible en ftp anonyme à `ftp.imag.fr:imag/APACHE/RAPPORTS`.

Divers

[51] J. CHASSIN DE KERGOMMEAUX, « Mapping and load-balancing in the SEPP-HPCTI projects », 1^{st} SEIHPC Workshop, jan 1996, Braga, Portugal.

[52] J. CHASSIN DE KERGOMMEAUX, « Mapping and load-balancing in the SEPP-HPCTI projects », 2^{nd} SEIHPC Workshop, sep 1996, Miskolc, Hungary.

[53] E. MOREL, J. BRIAT, J. CHASSIN DE KERGOMMEAUX, « PloSys : programmation en logique OU parallèle sur système sans mémoire commune », Actes RenPar'8, 1996, Poster.

[54] F.-G. OTTOGALI, V. OLIVE, B. D. O. STEIN, J. CHASSIN DE KERGOMMEAUX, J.-M. VINCENT, « Visualization of distributed Java applications for performance debugging », Soumis à publication.

[55] D. RIBOT, J. CHASSIN DE KERGOMMEAUX, C. VILLERMAIN, « Object-oriented Development Methodologies », Technical Newsletter, CAP GEMINI SOGETI, Décembre 1989, pages 7–10.

[56] C. TRON, Y. ARROUYE, J. CHASSIN, J.-P. KITAJIMA, E. MAILLET, B. PLATEAU, J.-M. VINCENT, « Parallel Systems Performance Evaluation - The ALPES environment », Parallel Computing ParCo93, 1993, Poster.

BIBLIOGRAPHIE GÉNÉRALE

Livres et monographies

[57] H. KRAWCZYK, B. WISZNIEWSKI, *Analysis and testing of distributed software applications, C^3 Industrial Control, Computers and Communications Series*, Research Studies Press Ltd, Baldock, Hertfordshire, England, 1998, ISBN 0 86380 222 2.

[58] M. SIMMONS, A. HAYES, J. BROWN, D. REED (réd.), *Debugging and Performance Tuning for Parallel Computing Systems*, IEEE Computer Society, 1996.

[59] R. M. STALLMAN, R. PESCH, S. SHEBS, ET AL, *Debugging with GDB : The GNU Source-Level Debugger*, Free Software Foundation, May 2000, For GDB Version 5.

Thèses et habilitations à diriger des recherches

[60] P.-E. BERNARD, *Parallélisation et multiprogrammation pour une application irégulière de dynamique moléculaire opérationnelle*, Thèse de doctorat en mathématiques appliquées, Institut National Polytechnique de Grenoble, France, octobre 1997.

[61] M. CHRISTALLER, *Vers un support d'exécution portable pour applications parallèles irrégulières : Athapascan-0*, thèse de doctorat, Université Joseph Fourier, Grenoble I, 1996.

[62] A. FAGOT, *Réexécution déterministe pour un modèle procédural parallèle basé sur les processus légers*, thèse de doctorat, Institut National Polytechnique de Grenoble, 1997.

[63] T. GAUTIER, *Calcul formel et parallélisme : Conception du Système Givaro et Applications au Calcul dans les Extensions Algébriques*, thèse de doctorat, Institut National Polytechnique de Grenoble, France, juin 1996.

[64] E. LEU, *La réexecution, pierre angulaire de la mise au point des programmes parallèles*, thèse de doctorat, Ecole Polytechnique Fédérale de Lausanne, July 1992, In French.

[65] E. MAILLET, *Traçage de logiciel d'applications parallèles : conception et ajustement de qualité*, thèse de doctorat, Institut National Polytechnique de Grenoble, septembre 1996, http ://callimaque.imag.fr :8081/htbin/docsearch/.

[66] J. M. MELLOR-CRUMMEY, *Debugging and Analysis of Large-Scale Parallel Programs*, Technical report no. 312, University of Rochester, sep 1989.

[67] B. D. O. STEIN, *Visualisation interactive et extensible de programmes parallèles à base de processus légers*, thèse de doctorat, Université Joseph Fourier, Grenoble, France, 1999.

Articles publiés dans des revues

[68] A. BIRRELL, B. NELSON, « Implementing Remote Procedure Calls », *ACM Trans. on Comp. Sys. 2*, 1984, p. 39–59.

[69] A. COUCH, « Categories and Context in Scalable Execution Visualization », *Journal of Parallel and Distributed Computing 18*, 2, 1993, p. 195–204.

[70] J. C. CUNHA, J. LOURENÇO, T. ANTAO, « An Experiment in Tool Integration : the DDBG Parallel and Distributed Debugger. », *Journal of Systems Architecture, 2nd Special Issue on Tools and Environments for Parallel Processing 45*, 11, 1999, p. 897–907.

[71] C. FIDGE, « Logical Time in Distributed Computing Systems », *IEEE Computer*, août 1991, p. 28–33.

[72] M. P. I. FORUM, « MPI : A message passing interface standard », *International Journal of Supercomputer Applications 8*, 3/4, 1994.

[73] I. FOSTER, C. KESSELMAN, S. TUECKE, « The Nexus Approach to Integrating Multithreading and Communication », *Journal of Parallel and Distributed Computing 37*, 1, août 1996, p. 70–82.

[74] M. T. HEATH, J. A. ETHERIDGE, « Visualizing the Performances of Parallel Programs », *IEEE Transactions on Software Engineering 8*, 5, mai 1991, p. 29–39.

[75] E. KRAEMER, J. T. STASKO, « The Visualization of Parallel Systems : An Overview », *Journal of Parallel and Distributed Computing 18*, 2, juin 1993, p. 105–117.

[76] D. KRANZLMÜLLER, S. GRABNER, J. VOLKERT, « Debugging with the MAD environment », *Parallel Computing 23*, 1997, p. 199–217.

[77] T. LAI, T. YANG, « On distributed snapshot », *Information Processing Letters 25*, 1987, p. 153–.

[78] L. LAMPORT, « Time, Clocks and Ordering of Events in a Distributed System », *Communications of the ACM 21*, 7, juillet 1978, p. 558–565.

[79] T. LEBLANC, J. MELLOR-CRUMMEY, « Debugging Parallel Programs with Instant Replay », *IEEE Transactions on Computers C-36*, 4, 1987, p. 471–481.

[80] E. MAILLET, C. TRON, « On efficiently implementing global time for performance evaluation on multiprocessor systems », *Journal of Parallel and Distributed Computing 28*, 1, 1995, p. 84–93.

[81] A. D. MALONY, A. REED, H. WIJSHOFF, « Performance Measurement Intrusion and Perturbation Analysis », *IEEE Transactions on parallel and distributed systems 3*, 4, juillet 1992.

[82] F. MATTERN, « Efficient algorithms for distributed snapshots and global virtual time approximation », *Journal of Parallel and Distributed Computing 18*, 1993, p. 423–434.

[83] C. E. MCDOWELL, D. P. HELMBOLD, « Debugging Concurrent Programs », *ACM Computing Surveys 21*, 4, December 1989, p. 593–622.

[84] B. MILLER *et al.*, « The Paradyn Parallel Performance Measurement Tool », *IEEE Computer*, novembre 1995.

[85] B. P. MILLER, C. MCDOWELL, « Summary of ACM/ONR Workshop on Parallel and Distributed Debugging », *ACM SIGPLAN NOTICES 26*, 12, décembre 1991, p. 1–14.

[86] B. P. MILLER, C. MCDOWELL, « Summary of ACM/ONR Workshop on Parallel and Distributed Debugging », *ACM SIGPLAN NOTICES 28*, 12, mai 1993, p. Vi–XiX.

[87] M. RONSSE, K. DE BOSSCHERE, « RecPlay : A Fully Integrated Practical Record/Replay System », *ACM Transactions on Computer Systems 17*, 2, May 1999, p. 133–152.

[88] V. SUNDERAM, « PVM : A Framework for Parallel Distributed Computing », *Concurrency : Practice and Experience 2*, 4, décembre 1990, p. 315–339.

Chapitres de livre

[89] D. HEMBOLD, C. MCDOWELL, « Race Detection - Ten Years later », in : Simmons *et al.* [58], p. 101–126.

[90] C. PANCAKE, « Collaborative Efforts to Develop User-Oriented Parallel Tools », in : Simmons *et al.* [58], p. 355–366.

[91] D. REED, J. BROWN, A. HAYES, M. SIMMONS, « Performance and Debugging Tools : A Research and Development Checkpoint », in : Simmons *et al.* [58], p. 1–22.

Communications à des manifestations scientifiques

[92] J. BRIAT, I. GINZBURG, M. PASIN, B. PLATEAU, « Athapascan Runtime : Efficiency for Irregular Problems », *in : Proceedings of EuroPar'97, LNCS*, 1300, Springer-Verlag, p. 591–600, août 1997.

[93] J. BULL, « A Hierarchical Classification of Overheads in Parallel Programs », *in : Proceedings of First IFIP TC10 International Workshop on Software Engineering for Parallel and Distributed Systems*, Chapman Hall, p. 208–219, 1996.

[94] H. CAIN, B. MILLER, B. WYLIE, « A Callgraph Based Search Strategy for Automated Performance Diagnosis », *in : Euro-Par 2000 Parallel Processing, Proc. 6th International Euro-Par Conference*, A. Bode, W. Ludwig, T. Karl, R. Wismüller (réd.), *LNCS, 1900*, Springer, p. 108–122, 2000.

[95] K. CHANDY, I. FOSTER, « Parallel language constructs for paradigm integration and deterministic computations », *in : Parallel Computing : Trends and Applications*, G. Joubert, D. Trystram, F. Peters, E. D.J. (réd.), *Advances in Parallel Computing, 9*, North Holland, p. 3–12, 1994.

[96] M. CHRISTALLER, « ATHAPASCAN-0A control parallelism approach on top of PVM », *in : Proc PVM User's group meeting*, University of Tennessee, Oak Ridge, 1994.

[97] J. CUNHA, J. LOURENÇO, J. VIEIRA, B. MOSCÃO, D. PEREIRA, « A Framework to support Parallel and Distributed Debugging », *in : Proc. of High Performance Computing and Networking, HPCN'98*, P. Sloot, M. Bubak, B. Hertzberger (réd.), *LNCS, 1401*, p. 708–717, 1998.

[98] A. FAGOT, J. P. KITAJIMA, « Evaluation de l'intrusion due à la prise de traces dans un programme parallèle », *in : RenPar'6*, 1994.

[99] E. FROMENTIN, M. RAYNAL, V. GARG, A. TOMLINSON, « On the Fly Testing of Regular Patterns in Distributed Computations », *in : Proc. 23rd Int. Conf. on Parallel Processing*, Pennsylvania State University Press, august 1994.

[100] F. GALILÉE, J.-L. ROCH, G. CAVALHEIRO, M. DOREILLE, « Athapascan-1 : On-Line Building Data Flow Graph in a Parallel Language », *in : PACT'98*, Paris, Oct 1998.

[101] M. HEATH, « Recent Developments and Case Studies in Performance Visualization using ParaGraph », *in : Workshop on Performance Measurement and Visualization of Parallel Systems*, Moravany, Czecho-Slovakia, octobre 1992.

[102] M. HURFIN, N. PLOUZEAU, M. RAYNAL, « EREBUS A debugger for asynchronous distributed computing systems », *in : Proceedings of the 3rd IEEE Workshop on Future Trends in Distributed Computing Systems*, Taiwan, avril 1992.

[103] C. JARD, T. JÉRON, G.-V. JOURDAN, J.-X. RAMPON, « A General Approach to Trace-checking in distributed computing systems », *in : Proc. 14th. Int. Conf. on Distributed Computing Systems*, Poznan, Poland, June 1994.

[104] D. KRANZLMÜLLER, J. VOLKERT, « Debugging Point-To-Point Communication in MPI and PVM », *in : Proc. EUROPVM/MPI 98 Intl. Conference*, p. 265–272, septembre 1998.

[105] E. LEU, A. SCHIPER, A. ZRAMDINI, « Execution Replay on Distributed Memory Architectures », *in : Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, p. 106–112, Dallas, USA, décembre 1990.

[106] E. LEU, A. SCHIPER, « Execution replay : a mechanism for integrating a visualization tool with a symbolic debugger », *in : CONPAR 92 - VAPP V*, Y. Robert, L. Bougé, M. Cosnard, D. Trystram (réd.), *LNCS*, *634*, Springer-Verlag, septembre 1992.

[107] L. LEVROUW, K. AUDENAERT, J. VAN CAMPENHOUT, « Execution replay with compact logs for shared-memory programs », *in : Applications in Parallel and Distributed Computing, IFIP Transactions A-44 : Computer Science and Technology*, Elsevier Science B.V., North Holland, p. 125–134, 1994.

[108] L. LEVROUW, K. AUDENAERT, J. VAN CAMPENHOUT, « A New Trace and Replay System for Shared Memory Programs based on Lamport Clocks », *in : Proceedings Euromicro Workshop on Parallel and Distributed Processing, PDP'94*, IEEE Computer Society Press, 1994.

[109] L. LEVROUW, K. AUDENAERT, « An efficient record-replay mechanism for shared memory programs », *in : Proc. Euromicro Workshop on Parallel and Distributed Processing*, IEEE Computer Society Press, p. 169–176, Jan. 1993.

[110] R. NAMYST, J.-F. MÉHAUT, « PM^2 Parallel Multithreaded Machine : a multithreaded environment on top of PVM », *in : Proceedings of EuroPVM'95*, HERMES (ISBN 2-86601-497-9), p. 179–184, 1995.

[111] R. NETZER, B. MILLER, « Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs », *in : Proceedings of Supercomputing '92*, Institute of Electrical Engineers Computer Society Press, Minneapolis, Minnesota, novembre 1992.

[112] M. NEYMAN, « Comparison of Different Approaches to Trace PVM Program Execution », *in : EuroPVM/MPI 2000*, J. Dongarra, et al (réd.), *LNCS*, 1908, Springer-Verlag, p. 274–281, 2000.

[113] D. PAN, M. LINTON, « Supporting Reverse Execution of Parallel Programs », *in : SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, p. 124–129, mai 1988.

[114] C. PANCAKE, R. NETZER, « A Bibliography of Parallel Debuggers, 1993 Edition », *in : Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, B. Miller, C. McDowell (réd.), *ACM SIGPLAN NOTICES*, *28*, 12, ACM Press, p. 169–186, San Diego, California, mai 1993.

[115] M. RONSSE, K. D. BOSSCHERE, « JiTI : Tracing Memory References for Data Race Detection », *in : Parallel Computing : Fundamentals, Applications and New Directions*, Elsevier, 1997.

[116] J. ROOS, L. COURTRAI, J. MÉHAUT, « Execution Replay of Parallel Programs », *in : Proceedings Euromicro Workshop on Parallel and Distributed Processing*, IEEE/Computer Society Press, January 1993.

[117] M. RUSSINOVITCH, B. COGSWELL, « Replay for Concurrent Non-Deterministic Shared-Memory Applications », *in : Proc. of the ACM SIGPLAN'96 Conference*

on Programming Language Design and Implementation, SIGPLAN Notices, 31, p. 258–266, 1996.

[118] B. J. N. WYLIE, A. ENDO, « Annai/PMA multi-level hierarchical parallel program performance engineering », *in : Proceedings 1st Int'l Work. on High-level Programming Models and Supportive Environments (HIPS'96, Honolulu, USA)*, IEEE Computer Society Press, p. 58–67, avril 1996, ftp ://ftp.cscs.ch/pub/CSCS/papers/HIPS96.ps.gz.

Rapports de recherche

[119] A. BENOIT, *Analyse des interactions entre le traceur Athapascan-0 et les applications tracées*, Rapport de dea d'informatique : Systèmes et communications, École Doctorale Mathématiques et Informatique, UJF et INPG, Juin 2000.

[120] C. CLÉMENÇON, A. ENDO, J. FRITSCHER, A. MUELLER, B. J. N. WYLIE, « Annai Scalable Run-time Support for Interactive Debugging and Performance Analysis of Large-scale Parallel Programs », Technical Report N° TR-96-04, Swiss Center for Scientific Computing, Via Cantonale, CH-6928 Manno, Suisse, 1996, http ://www.cscs.ch/Official/Publications/PubTR96.html.

[121] ETNUS, *TotalView User's Guide*, édition Version 4.1, June 2000.

[122] E. GAY, *Réexécution partielle de programmes parallèles*, Rapport de dea, UFR-IMA, Grenoble, 1995.

[123] J.-M. HÉLARY, A. MOSTEFAOUI, M. RAYNAL, « Déterminer un état global dans un système réparti », rapport de recherche N° 2090, INRIA, 1993, http ://www.inria.fr/RRRT/publications-fra.html.

[124] V. HERRARTE, E. LUSK, *Studying parallel program behavior with upshot*, Argonne National Laboratory, 1992, http ://www-fp.mcs.anl.gov/ lusk/upshot/upshotman/upshot.html.

[125] D. A. REED, R. A. AYDT, T. M. MADHYASTHA, R. J. NOE, K. A. SHIELDS, B. W. SCHWARTZ, « An Overview of the Pablo Performance Analysis Environment », rapport de recherche, Department of Computer Science, University of Illinois, Urbana, Illinois 61801, 1992.

[126] F. RIBETTE, *Visualisation en ligne d'applications multi-threadées dans l'environnement d'exécution Athapascan-0*, Rapport de stage de dess, Université Louis Pasteur, Strasbourg, 1999.

[127] F. TEODORESCU, *Utilisation des techniques de réexécution déterministe pour la mesure de performances de programmes parallèles*, Rapport de dea d'informatique, Ensimag, 1994.

[128] Y. VASSILIEFF, *Implémentation d'un mécanisme de sauvegarde pour Athapascan*, Projet de fin d'études, ENSIMAG, Grenoble, 1994.

[129] Q. ZHAO, J. STASKO, « Visualizing the Execution of Threads-based Parallel Programs », rapport de recherche N° GIT-GVU-95-01, Georgia Institute of Technology, 1995.

Divers

[130] C. PANCAKE, R. NETZER, « Bibliography on Parallel and Distributed Debuggers », http ://wheat.uwaterloo.ca/bibliography/Parallel/debug_3.1.html, 1994.