Réexécution déterministe pour un modèle procédural parallèle basé sur les processus légers

Alain Fagot

Réexécution déterministe pour un modèle procédural parallèle basé sur les processus légers

FAGOT Alain

Tout d'abord à Brigitte PLATEAU, qui me fait l'honneur de présider le jury, j'adresse un sincère remerciement. Son intérêt pour mon travail fut constant. En tant que cheffe des Apaches, tous les travaux avaient leur importance à ses yeux. De nombreux conseils avisés m'ont été donnés durant les réunions du groupe de travail sur l'évaluation des performances.

Bien sûr je remercie Mireille DUCASSÉ, Jean-Marc GEIB et Jean-François MÉHAUT, qui ont accepté avec une grande gentillesse d'être les rapporteurs de cette thèse. Une partie de leur été fut consacré à la lecture et à la compréhension de mon travail.

Ik zou Koen DE BOSSCHERE *willen bedanken voor zijn bereidheid om te rapporteren over mijn proefschrift, hoewel de Franse taal niet gemakkelijk voor hem is. Dank voor uw aanwezigheid, als genodigde gedurende de verdediging van dit werk.*

Pour Jacques CHASSIN DE KERGOMMEAUX, qui supporta mes humeurs, mes mousses au chocolat, mon intérêt pour l'espéranto et mes confidences, j'exprime ici toute ma reconnaissance. Pour l'encadrement et la confiance, que tu m'as accordés, tu mériterais bien des éloges. Ton dévouement manqua même te coûter la vie avant la fin de la rédaction de ce manuscrit.

Une petite attention spéciale va à Jean-Marc VINCENT, qui m'a prodigué de nombreux conseils. Outre tes leçons de statistiques et tes conseils de rédaction, j'ai beaucoup apprécié tes recettes. Vivent le homard et la mousse au chocolat.

Le travail de nombreux autres Apaches a influé sur le mien et réciproquement. Des nuits et des jours passés à travailler avec Alexandre, Benhur, Christophe, Éric, Florin, Frédéric, João-Paulo, Johan, Laurent, Martha-Rosa, Michel, Michel, Nathalie, Pascal, Pascal, Paulo, Pierre-Éric, Philippe, Thierry, Thierry, Youri, je garderai d'agréables souvenirs. Bien évidemment je n'oublierai pas non plus ceux avec qui j'ai eu des relations plus détendues !

Je dédie un bisou spécial à Marie-Pierre, qui a consacré ses jours de pluie à la relecture de mon manuscrit. Son point de vue de néophyte en parallélisme m'a aidé à éclaircir certains points pourtant évidents. ☺

Küsse auch für Mutti.

Finfine
al ĉiuj, kiuj senrezulte provis malhelpi,
kaj al ĉiuj, kiuj feliĉe sukcesis helpi,
mi tutkore dankas.

Feliĉeco estas povi sin razi,
kiam oni deziras.

BÉAREZ Oscar

Table des matières

Table des figures

Liste des tableaux

Introduction

La spectaculaire évolution technologique des dernières années a réduit les coûts de production des composants électroniques tout en augmentant les performances en vitesse, en intégration et en fiabilité. En plus du gain de puissance fourni par la diminution du temps nécessaire pour effectuer un calcul élémentaire, la réalisation simultanée de plusieurs calculs augmente la puissance des calculateurs. Les ordinateurs multiprocesseurs sont devenus une partie importante de l'activité des constructeurs informatiques. Les architectures parallèles sont efficacement utilisées pour résoudre un même problème par le travail simultané de plusieurs processeurs. Le traitement parallèle consiste à découper le traitement global en processus indépendants qui se synchronisent par divers mécanismes afin de collaborer à la réalisation d'un travail plus important en taille et en temps de calcul que ne le permettrait une machine séquentielle. Les méthodes et les environnements de programmation doivent évoluer pour s'adapter à l'exploitation de ces nouvelles capacités. Les demandes toujours croissantes en puissance de calcul incitent à développer l'usage du parallélisme dans la plupart des machines actuelles.

But de cette thèse

Le but de cette thèse est de définir et construire un socle pour un atelier de mise au point des applications parallèles basées sur les processus légers. Dans les modèles de calculs non synchronisés, les processus sont en compétition pour l'accès aux ressources partagées. Le comportement d'une exécution dépend plus ou moins fortement de la résolution de ces conflits d'accès. Si son comportement est toujours identique, une application est qualifiée de déterministe. Si son comportement est sensible aux changements dans l'environnement d'exécution, elle appartient au groupe des applications indéterministes. L'indéterminisme du comportement d'une application erronée peut être à

l'origine d'erreurs furtives, qui se manifestent peu fréquemment. Ces erreurs sont donc difficiles à identifier avec les outils traditionnels de la programmation séquentielle. Avec ces techniques, une application erronée est réexécutée autant de fois que nécessaire pour permettre la localisation de la cause de l'erreur. Pour une application parallèle indéterministe, la mise en œuvre de techniques de réexécution déterministe permet d'appliquer une approche cyclique de déverminage. L'élimination des erreurs vise essentiellement les erreurs de programmation mais les problèmes de performance peuvent être traités par une approche similaire.

Environnement de recherche

La répartition efficace du calcul sur les architectures parallèles est à l'heure actuelle très dépendante de l'architecture de la machine, du problème traité et de l'algorithme retenu par le programmeur. Les objectifs du projet APACHE [69] concernent les techniques de programmation des ordinateurs parallèles. Un intérêt tout particulier est porté à l'exécution efficace des applications dites irrégulières. Ces applications manipulent des données faiblement structurées ou dont la représentation peut varier au cours de l'exécution. C'est le cas en particulier des applications qui calculent des déformations, des interactions ou des déplacements d'objets complexes comme des carrosseries, des galaxies ou des molécules.

Pour exploiter au mieux les ressources disponibles, les applications irrégulières ont besoin de virtualiser l'architecture qui supporte l'exécution. Pour offrir au programmeur une interface de haut niveau, qui soit portable et fasse abstraction de la machine, notre équipe développe le noyau exécutif ATHAPASCAN. Cette plateforme est divisée en deux parties : ATHAPASCAN-0 qui réalise une couche de portabilité et ATHAPASCAN-1 qui offre une couche de répartition de charge automatique. La couche de portabilité réalise l'abstraction de la machine, qui supporte la plateforme.

Le paradigme de programmation retenu pour ATHAPASCAN est de type procédural parallèle. L'expression du parallélisme se fait par la déclaration des appels de procédures qui peuvent être réalisés concurremment. Selon l'état de charge du système, la couche de répartition fait appel à la couche de portabilité pour distribuer effectivement le calcul sur la machine. Afin d'évaluer l'efficacité du mécanisme de répartition, d'autres outils sont développés dans le cadre du projet. Un outil de mesures et d'évaluation de performances permet d'analyser les applications écrites pour ATHAPASCAN-0. Enfin un

atelier de mise au point complétera l'environnement de développement d'applications parallèles.

Motivation pour les processus légers

Pendant longtemps le parallélisme a été exploité entre processus lourds (ou processeurs virtuels, en anglais *heavyweight processes* ou *virtual processors*) dans les systèmes d'exploitation traditionnels. Le parallélisme peut être simulé par la coopération de plusieurs processus auxquels le système attribue successivement le processeur. L'introduction des processus légers (ou fils d'exécution, en anglais *lightweight processes* ou *threads*) mène à l'étude d'un parallélisme simulé par des coroutines à l'intérieur même d'un processus lourd. Un processus léger bénéficie d'un contexte qui ne comporte en propre ni segment de code ni segment de données puisque ces segments sont partagés avec les autres processus légers du processus lourd englobant. Si les processus légers ne sont pas gérés directement par le système d'exploitation, deux grains d'ordonnancement cohabitent. Le système d'exploitation ordonnance les processus lourds et ceux-ci ordonnancent les processus légers qu'ils supportent. Selon la technique retenue, l'ordonnancement des processus légers peut imiter celui des processus lourds par attribution de *quantum de temps* ou basculer les processus légers uniquement dans les appels à certaines fonctions.

L'évolution technologique actuelle se traduit par une augmentation plus rapide de la vitesse de calcul des processeurs que de la vitesse de communication des réseaux. Pour exploiter pleinement la puissance de calcul des processeurs, il devient très important de masquer les délais de communication par des calculs. Cette préoccupation concerne bien sûr au premier chef les machines parallèles. Une analogie peut être faite avec les machines séquentielles où ce problème se retrouve aussi pour les accès à la mémoire. Dans ces machines, une solution consiste à mettre en œuvre une hiérarchie de caches. Le parallélisme gagne également l'architecture interne des processeurs avec des conceptions dites à mots d'instruction très longs (en anglais, *Very Large Instruction Word*) qui codent plusieurs opérations réalisables en parallèle. Avec des architectures classiques de processeur, l'utilisation de communications asynchrones permet d'émettre un message pendant que le processeur effectue des calculs. Cette technique de recouvrement des temps de calcul et de communication nécessite de changer rapidement de contexte d'exécution pour profiter au maximum de la puissance du processeur. Les processus légers

présentent l'avantage d'avoir un contexte réduit qui ne nécessite qu'une dizaine à une centaine de cycles pour être basculé. Le basculement du contexte d'un processus lourd nécessite un temps cent à mille fois plus important.

Dans le cadre du calcul parallèle, les processus légers permettent l'expression d'un parallélisme plus fin. Ils rendent possible la manipulation d'un grain d'ordonnancement adapté aux architectures distribuées. Pour celles-ci, ils sont incontournables dès lors qu'il s'agit de masquer par des calculs les délais de communication. Un intérêt supplémentaire consiste à pouvoir créer à distance des charges de travail, par exemple par l'exécution de procédure à distance, sur un processeur distinct de celui où s'exécute le processus appelant. Le contexte réduit d'un processus léger permet de le migrer plus facilement entre processus lourds de même modèle. Ces possibilités peuvent être exploitées pour répartir dynamiquement la charge globale de l'application [32]. La répartition dynamique peut conduire à des comportements différents pour deux exécutions successives d'une même application. Les applications irrégulières illustrent ce phénomène par leur comportement dépendant des données et de l'état du système. Les gains de performance liés au recouvrement des temps de communication et de calcul confirment l'intérêt de ce modèle de programmation. Son utilisation doit s'accompagner de mécanismes permettant un bon ordonnancement global, essentiel pour répartir efficacement la charge des applications irrégulières.

Travail réalisé

Le travail présenté vise essentiellement à aider les programmeurs pour la mise au point d'applications utilisant un paradigme de programmation impérative, où les processus légers sont utilisés pour exécuter concurremment des procédures. L'indéterminisme est utilisé dans ces modèles pour exploiter au mieux les ressources disponibles. Dans ce cadre, la mise au point des applications parallèles se base sur la réexécution déterministe. ATHAPASCAN appartient à cette famille de modèles de programmation par appels concurrents à des procédures à distance. Par rapport aux modèles plus généraux de processus communiquants ou de mémoire partagée, les procédures introduisent des motifs élémentaires de communication qui peuvent être combinés en événements abstraits. L'originalité de notre approche consiste à considérer ces événements abstraits pour simplifier le mécanisme de traçage et de réexécution. Un modèle d'exécution est défini afin d'exprimer les conditions d'équivalence de deux exécutions pour la programmation parallèle procédu-

rale. Ce modèle n'est pas spécifique à ATHAPASCAN et peut s'appliquer à d'autres environnements d'exécution tels que PM^2 [67]. Le prototype ATHAPASCAN-0a [14] était en développement durant notre étude. Nous avons donc pu collaborer à la conception du noyau exécutif pour le doter d'un mécanisme de réexécution déterministe. L'implantation de la réexécution déterministe dans le noyau exécutif ATHAPASCAN-0a dérive de cette étude théorique. Elle a nécessité la modification du noyau exécutif pour y inclure l'enregistrement de traces et la réexécution déterministe d'applications ATHAPASCAN-0a. Une expérimentation par une application très indéterministe valide la réalisation pratique du mécanisme. La validation du fonctionnement correct est complétée par une estimatoin du surcoût en temps, obtenue par une série de mesures. Celles-ci, menées selon une méthodologie systématique, visent à étudier, sur un large spectre de programmes parallèles, le surcoût dû à l'enregistrement des traces pour la réexécution déterministe.

Organisation de ce document

Dans cette thèse, nous traitons de la réexécution déterministe pour un modèle procédural parallèle basé sur les processus légers. Ce document est structuré en six chapitres.

Dans le premier chapitre sont exposés les problèmes de la mise au point d'applications parallèles. Après un rappel des définitions de base, nous présentons les opérations nécessaires à un programmeur pour contrôler et comprendre le comportement d'une application. L'indéterminisme des exécutions est un obstacle à la compréhension du comportement d'une application erronée. Des outils de mise au point d'applications indéterministes sont présentés à la fin du chapitre. Ces outils sont basés sur la réexécution déterministe.

Dans le second chapitre sont exposés les principes de la réexécution déterministe sur laquelle se basent les outils présentés dans le premier chapitre. Après une description du principe originel, nous présentons les extensions définies pour différents modèles de programmation. Une exécution équivalente d'une exécution parallèle peut être produite à partir de traces d'événements. Les événements enregistrés sont des événements de synchronisation, qui permettent de réexécution l'application de manière déterministe relativement à l'exécution enregistrée.

Dans le troisième chapitre sont exposés un modèle procédural parallèle et les conditions d'équivalence de deux exécutions dans ce modèle. Après la définition du modèle de base et la preuve d'une condition suffisante pour

l'équivalence de deux exécutions, nous présentons l'extension du modèle à la communication par échange de messages et à la synchronisation entre processus. Le concept d'appel de procédure permet d'englober tous ces mécanismes au sein du modèle de base.

Dans le quatrième chapitre sont exposés les travaux relatifs à la réalisation d'un mécanisme de réexécution déterministe pour le noyau exécutif ATHAPASCAN-0a. Après une description rapide du noyau exécutif ATHAPASCAN-0a, nous présentons l'intégration de la fonctionnalité de réexécution déterministe. La construction de ce mécanisme de réexécution déterministe est guidée par l'étude théorique présentée dans le troisième chapitre. Un exemple d'application indéterministe illustre le fonctionnement du mécanisme.

Dans le cinquième chapitre sont exposées les mesures du surcoût en temps d'exécution introduit par l'enregistrement des traces nécessaires à la réexécution. Après une description du cadre méthodologique, nous présentons l'adaptation des outils pour le noyau exécutif ATHAPASCAN. Des programmes parallèles synthétiques ATHAPASCAN-0a sont générés à partir de modèle d'algorithmes. Un banc d'essai est constitué de plusieurs instances d'un groupe de modèles choisis. Les mesures font apparaître une faible intrusion du mécanisme d'enregistrement des traces.

Dans le sixième chapitre sont exposés un bilan de l'utilisation de la réexécution déterministe pour ATHAPASCAN-0a et les perspectives d'application des travaux de cette thèse. Après une introduction à l'évaluation de performances lors d'une réexécution déterministe, nous présentons la démarche conduisant à la réalisation d'un mécanisme de réexécution déterministe pour la plateforme ATHAPASCAN-0b. Celle-ci remplace le noyau exécutif ATHAPASCAN-0a en améliorant les possibilités et les performances offertes aux programmeurs. Le modèle exploité est toujours celui d'appels de procédures à distance, pour lequel cette thèse définit la réexécution déterministe. Une discussion sur la nécessité et les difficultés de réaliser la réexécution déterministe clôt ce document.

Chapitre 1

Mise au point d'applications parallèles

Nous nous limiterons dans ce document à la mise au point dynamique. Celle-ci nécessite l'exécution de l'application, en opposition à l'analyse statique qui utilise le programme source pour en extraire les comportements possibles, sans l'exécuter. Des méthodes hybrides pratiquent une analyse statique sur des traces d'exécution. La mise au point dynamique d'une application nécessite d'en comprendre le comportement.

Dans le cadre de la construction d'un atelier de mise au point dynamique, nous nous intéresserons essentiellement à la réalisation d'outils d'aide pour le programmeur plutôt qu'à la conception d'outils de détection automatique des erreurs. Une approche classique pour la mise au point dynamique consiste à observer le comportement d'une application erronée au cours de réexécutions successives. Durant chaque exécution, le programmeur tente de comprendre davantage le comportement de l'application. Dans le cas d'une application parallèle, le comportement peut varier d'une exécution à l'autre à cause de l'indéterminisme des exécutions. La technique de réexécution déterministe permet de se ramener au cas de la mise au point d'une application déterministe, en produisant des exécutions équivalentes à partir de traces d'exécution.

Dans un premier temps, les outils formels essentiels sont rappelés. Ils permettent de définir les notions importantes comme le comportement observable d'une application parallèle, l'équivalence de deux exécutions et l'indéterminisme. Les phases de la mise au point cyclique dressent le cadre de travail d'un atelier de mise au point dynamique. Les éléments de réalisation d'un tel atelier sont ensuite présentés.

1.1 Rappels d'algorithmique parallèle et distribuée

Cette section présente les définitions adoptées pour les notions essentielles d'algorithmique parallèle et distribuée. L'observation du comportement d'une application parallèle se base sur la définition des événements d'une exécution. L'intrusion de l'observation induit l'indéterminisme des exécutions, par l'effet de sonde sur les conflits d'accès. L'utilisation de la réexécution déterministe impose de définir l'équivalence de deux exécutions. Pour cela des outils comme les horloges vectorielles nous seront nécessaires. Les définitions sont regroupées par thème : exécution, temps, causalité, intrusion et recouvrement calcul/communication.

1.1.1 Exécution

Pour mettre en œuvre une approche cyclique de mise au point dynamique, il faut pouvoir observer, lors de chaque cycle, une exécution équivalente à celle ayant manifesté le comportement erroné. Les définitions de ce paragraphe nous permettent de caractériser deux exécutions équivalentes.

Définition 1.1 *Pour Chandy et Lamport [12], un* événement *est une action atomique qui peut changer l'état du processus. Il est noté e.*

Définition 1.2 L'état *d'un processus est déterminé par son activité (en cours d'exécution ou suspendu) et par les valeurs de ses variables locales.*

Définition 1.3 *Un* processus *est caractérisé par une séquence totalement ordonnée d'événements locaux. Il est noté :*

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

Un événement e peut être *interne* à un processus ou impliquer plusieurs processus dans une *synchronisation*. Par exemple si e_i^p est un envoi de message et e_j^q la réception de ce message, il y aura une relation entre ces événements. Le lien de causalité entre les événements sera présenté en 1.1.3.

Définition 1.4 *Dans une synchronisation, un événement* source e_i^p *est lié causalement à un événement* cible e_j^q *(ou éventuellement plusieurs*[1]*).*

1. Dans le cas de l'écriture et de la lecture d'une zone de mémoire partagée, un événement d'écriture peut être lié à plusieurs événements de lecture.

Le *comportement* d'une application est donné par les réactions de chaque processus à ses propres événements. L'enregistrement d'informations relatives aux événements constitue une trace de l'exécution.

Définition 1.5 *Une* trace *est formée d'une séquence d'enregistrements relatifs aux événements d'un processus. Par commodité, on confond souvent la notation d'un événement et de l'enregistrement qui le décrit. Une trace se note :*

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

Définition 1.6 *Une* exécution *est caractérisée par l'ensemble E des événements de chaque processus de l'application et par un ordre partiel P sur ces événements. Elle est noté :*

$$X = \langle E, P \rangle$$

Les ordres partiels définis sur les événements d'une application parallèle sont présentés en 1.1.3. La trace d'une exécution d'une application est l'ensemble des traces des processus qui ont participé à cette exécution.

Définition 1.7 L'instrumentation *consiste à introduire, dans l'exécution de l'application, des actions pour construire une trace.*

Les actions introduites ajoutent des événements à l'exécution. Ces événements ne sont pas captés par l'instrumentation mais induisent une perturbation. Les effets en sont présentés en 1.1.4.

Définition 1.8 *Selon Mellor-Crummey [62], deux exécutions sont* équivalentes *si et seulement si elles ont le même ensemble partiellement ordonné d'événements.*

$$X \approx X' \Longleftrightarrow X = \langle E, P \rangle \wedge X' = \langle E, P \rangle$$

1.1.2 Temps

Pour en présenter les relations, les événements d'une exécution peuvent être reportés sur un *diagramme espace-temps*. La figure 1.1 présente quatre processus, leurs événements et les relations entre ceux-ci. Le temps s'écoule de gauche à droite selon une convention généralement admise (lire la discussion de Mattern dans [61]). Les processus sont donc placés sur l'axe vertical. Les événements sont désignés par les points sur les lignes qui représentent

les processus. Les relations entre les événements de synchronisation sont représentées par des flèches. Celles-ci sont dirigées de l'événement source vers l'événement cible (par exemple envoi et réception d'un message ou écriture et lecture d'une zone de mémoire partagée).

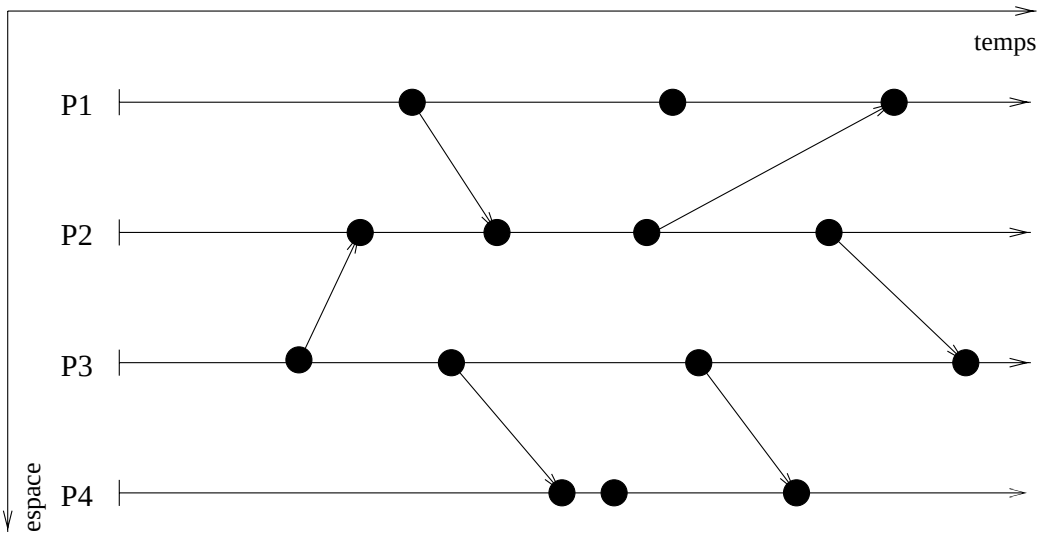


FIG. 1.1 – *Diagramme espace-temps.*

Les *horloges physiques* des processeurs peuvent être utilisées pour estampiller les événements. Dans une trace, l'enregistrement de chaque événement peut comporter une *estampille* qui représente la valeur de l'horloge physique du processeur à l'instant où a eu lieu la capture de l'événement. Cette estampille peut servir à calculer des intervalles de temps entre des événements.

Comme les processus s'exécutent sur des processeurs distincts, les valeurs des horloges physiques locales ne peuvent pas toujours être utilisées pour ordonner globalement les événements. Chaque horloge physique a une fréquence légèrement différente de celle d'un autre processeur de la même machine parallèle. Cette dérive peut être plus ou moins sensible selon les systèmes. Afin de disposer d'un *temps global cohérent*, des mécanismes permettent de corriger algorithmiquement les dérives des horloges. À partir des travaux de Malony, Reed et Wijshoff [58], Maillet et Tron [56] proposent d'autres algorithmes pour effectuer efficacement la correction.

L'usage principal du temps physique intéresse les mesures de performance des applications. Aussi précis soit-il, le temps physique ne suffit pas pour exprimer toutes les relations entre les événements. Des travaux, comme ceux de Jard, Jeron, Jourdan et Rampon [42], se basent sur la précédence entre

les événements. Ils exploitent la relation de causalité pour déterminer des propriétés sur les exécutions.

1.1.3 Causalité

Sur la figure 1.1, les flèches entre les événements notent la causalité directe entre les événements de processus différents. Les événements successifs d'un même processus sont églement en relation de causalité directe (par exemple, sur la figure 1.2, les événements e_1^1 et e_2^1 du processus P1).

Définition 1.9 *Deux événements e_i^p et e_j^q sont en relation de* causalité directe *(notée $e_i^p \prec_D e_j^q$) dans deux cas :*

1. *$p = q$ et $i = j - 1$, ou*
2. *$p \neq q$, e_i^p et e_j^q sont des événements de synchronisation où e_i^p est l'événement source et e_j^q l'événement cible.*

Définition 1.10 *La* précédence causale *est la fermeture transitive de la relation de causalité directe. Deux événements e_i^p et e_j^q sont en relation de précédence causale (notée $e_i^p \prec e_j^q$) dans deux cas :*

1. *$e_i^p \prec_D e_j^q$, ou*
2. *$\exists e_k^r, e_i^p \prec e_k^r \wedge e_k^r \prec e_j^q$.*

La relation de précédence causale définit un ordre partiel P entre les événements d'une exécution. Cet ordre caractérise l'exécution $X = \langle E, P \rangle$ (avec l'ensemble des événements E). L'horloge de Lamport [48] exprime cet ordre P, noté aussi par $\xrightarrow{HL}$. Il est possible de définir sur E un ordre total T compatible avec P.

Définition 1.11 *Pour un événement e_i^p, la valeur de* l'horloge de Lamport *$HL(e_i^p)$ est calculée à partir de la valeur locale hl_p de l'horloge du processus p et de la valeur $HL(e_j^q)$ de l'horloge d'un éventuel événement e_j^q associé. La mise à jour de la valeur de l'horloge suit l'algorithme suivant :*

1. $HL(e_i^p) = \max(hl_p, HL(e_j^q)) + 1$
2. $hl_p = HL(e_i^p)$

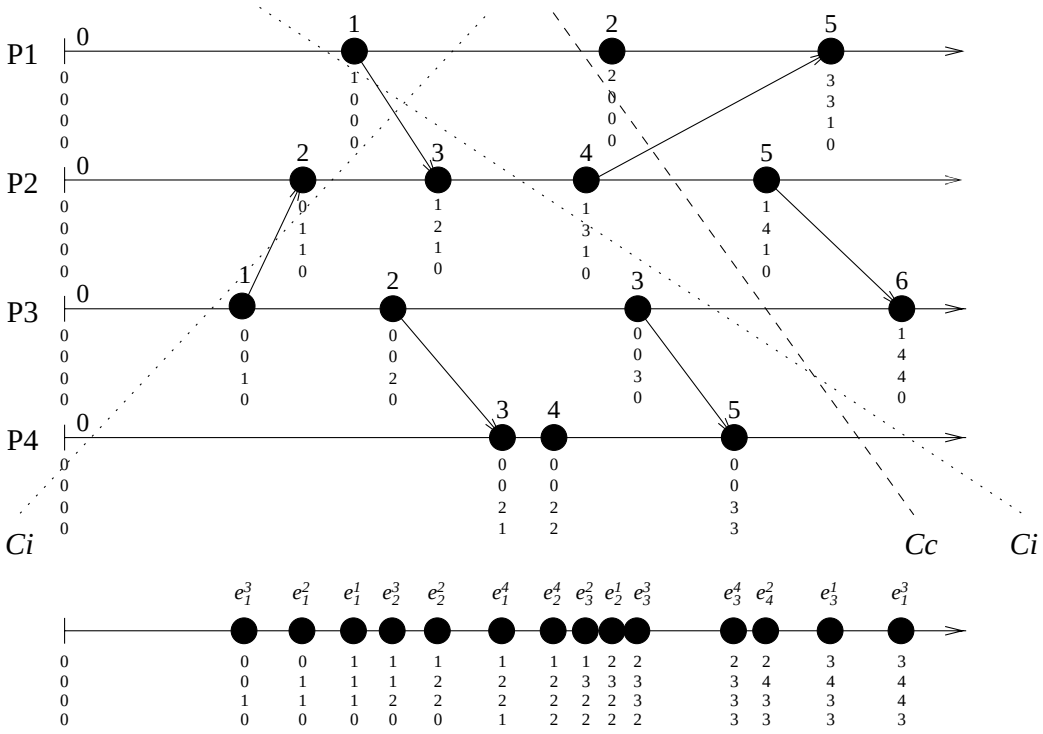


FIG. 1.2 – *Temps logiques.*

Sur la figure 1.2, les valeurs de l'horloge de Lamport sont placées au dessus de chaque événement. Il apparaît clairement que l'horloge de Lamport n'implique pas toujours une dépendance causale entre deux événements dont les valeurs d'horloge sont différentes. Par exemple, l'événement e_3^2 porte la valeur 4 et l'événement e_3^4 la valeur 5 alors qu'il n'y a aucun lien causal entre ces deux événements (de même pour e_2^4 et e_4^2).

L'ordre partiel P entre les événements peut être codé par une horloge vectorielle. Fidge [25] et Mattern [60] en ont donné indépendemment une définition. La valeur d'une horloge vectorielle est un vecteur dont la dimension est le nombre de processus dans l'application. L'horloge indique pour chaque processus sa dépendance par rapport aux événements des autres processus. Elle indique, pour chaque processus partenaire, le dernier événement qui a pu influer sur le processus.

Définition 1.12 *Pour un événement e_i^p, la valeur de* l'horloge vectorielle *$HV(e_i^p)$ est calculée à partir de la valeur locale hv_p de l'horloge du processus p et de la valeur $HV(e_j^q)$ de l'horloge d'un éventuel événement e_j^q associé. La*

mise à jour de la valeur de l'horloge suit l'algorithme suivant :

1. $HV(e_i^p) = \sup(hv_p, HV(e_j^q))$
2. $HV(e_i^p)[p] = HV(e_i^p)[p] + 1$
3. $hv_p = HV(e_i^p)$

où l'opérateur sup *réalise un maximum composant par composant.*

Sur la figure 1.2, les valeurs des horloges vectorielles sont placées sous chaque événement. Sous la figure se trouve la séquence que verrait un *observateur idéal* conscient instantanément de tous les événements de chaque processus.

Charron-Bost [13] a démontré que l'horloge vectorielle est la représentation la plus précise et la plus compacte de la causalité. L'horloge de Lamport dont le codage est plus simple ne permet en effet pas d'ordonner causalement les événements qui portent des valeurs d'horloge différentes.

L'utilisation d'une horloge vectorielle impose de connaître *a priori* tous les processus d'une exécution car les horloges vectorielles ne sont pas extensibles. Fidge propose une autre forme d'horloge logique [26], qui croît au cours d'une exécution. Cette horloge code explicitement les processus alors qu'ils sont implicitement représentés dans une horloge vectorielle. Cette nouvelle horloge est un ensemble de couples $(processus, valeur)$. Sa gestion est assez difficile à mettre en œuvre.

Sur le diagramme espace-temps, il est possible de tracer une ligne qui intersecte la ligne de chaque processus. Elle sépare ainsi le diagramme entre le passé et le futur. Une telle ligne est appelée une *coupe* (voir les lignes en pointillés sur la figure 1.2).

Définition 1.13 *Pour Chandy et Lamport [12], un* état global cohérent *d'une application distribuée est donné par une coupe dans le diagramme espace-temps telle que toutes les flèches soient dirigées du passé vers le futur.*

La cohérence ainsi établie garantit qu'il n'y a pas de message reçu avant d'être émis ou de valeur lue avant d'être écrite. Sur la figure 1.2, la coupe marquée Cc est cohérente car la flèche entre e_3^2 et e_3^1 est dirigée du passé vers le futur. Les coupes marquées Ci sont *incohérentes* car la précédence entre certains événements n'est pas respectée (de e_1^1 vers e_2^2 ou de e_1^3 vers e_1^2). L'état global cohérent d'une application est constitué des états de chaque processus

avant la coupe et des valeurs portées par les synchronisations qui traversent la coupe. Il s'agit d'un état possible de l'application qu'un observateur idéal pourrait voir.

1.1.4 Intrusion

La construction d'une trace d'exécution nécessite d'ajouter des instructions dans le code de l'application s'il n'y a pas de dispositif matériel disponible pour cet usage. L'effet de ces instructions supplémentaires est de produire de nouveaux événements ou de rallonger la durée des états entre deux événements.

Définition 1.14 *Dans un processus,* l'intrusion *ajoute des événements ou rallonge la durée des états.*

Définition 1.15 *Un* conflit d'accès *se produit lorsque plusieurs événements donnés peuvent se produire dans un ordre indéterminé.*

Définition 1.16 *La* résolution *d'un conflit d'accès est l'ordre effectif selon lequel les événements se succèdent dans le temps.*

L'intrusion peut perturber la résolution des conflits d'accès. Ceci vient de *l'indéterminisme* de certaines actions des processus. Les actions de synchronisation sont les plus sujettes à l'indéterminisme. Le choix du (ou des) processus partenaire(s) dans une synchronisation peut être facultatif ou impossible. Par exemple, pour une lecture d'une zone mémoire partagée, il peut être impossible de spécifier après quelle écriture elle doit avoir lieu. Ou encore pour une réception de message, il peut être admis de prendre le premier message en provenance de l'un quelconque des autres processus de l'application.

Définition 1.17 L'effet de sonde *est la modification du comportement d'une application par l'intrusion.*

Toute *observation* par traçage logiciel est intrusive. Elle ne présente de l'exécution qu'une vue déformée par l'effet de sonde (en anglais *probe effect* [31]). Cette propriété n'est pas exclusive au parallélisme. En programmation séquentielle aussi, des erreurs disparaissent lorsque des traces sont ajoutées.

1.2 Importance de l'indéterminisme

L'indéterminisme des exécutions est le principal problème qui se pose pour la mise au point dynamique d'applications parallèles. Il est toutefois utile pour assurer aux applications de bonnes performances indépendemment de l'environnement d'exécution. Une application, qui utilise l'indéterminisme pour améliorer ses performances, peut s'adapter à la rapidité relative de chaque processeur exploité par une exécution. Elle peut aussi plus aisément être portée d'une architecture vers une autre.

1.2.1 Changement de comportement

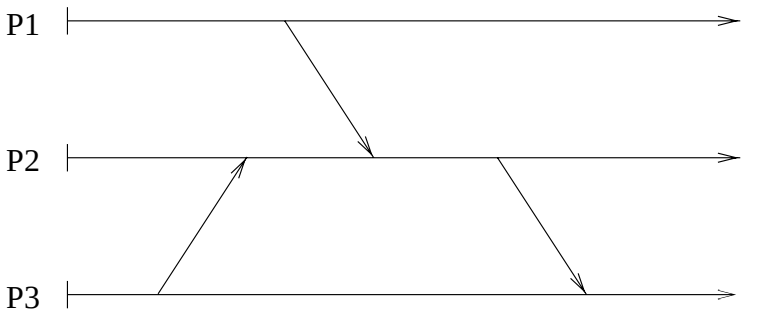
L'indéterminisme peut conduire une application à changer son comportement selon les délais de synchronisation. Pour plusieurs exécutions d'une même application indéterministe, l'ordre de réception de messages varie. Cet ordre est sensible aux variations des temps de calcul ou de communication. L'ajout d'instructions dans le code ou d'informations dans les interactions influe sur ces temps.

La figure 1.3 présente l'effet d'une intrusion sur l'un des processus. Le processus P2 exécute le code suivant :

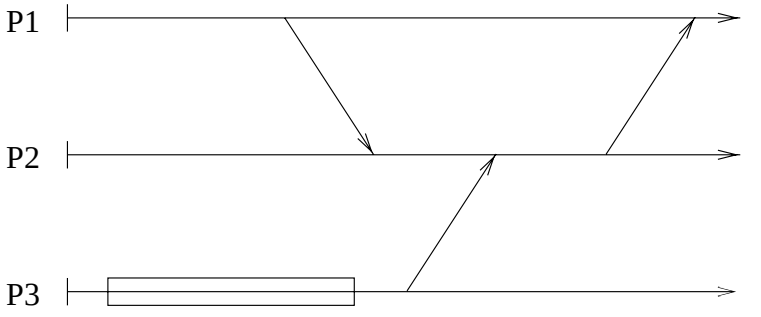
```
message1 = Recevoir(INDIFFÉREMMENT)
Traitement(message1)
message2 = Recevoir(INDIFFÉREMMENT)
Traitement(message2)
Envoyer(ExpéditeurDe(message1),réponse)
```

Le cas (a) montre une exécution sans intrusion de ce code. Si le processus P3 subit une intrusion, le comportement est modifié en celui présenté par le cas (b).

Ce type d'indéterminisme est exploité par les *applications opportunistes* qui utilisent les ressources selon leur disponibilité. L'opportunisme peut être passif, comme dans le cas de la réception d'un message quelconque, ou actif si l'application répartit sa charge dynamiquement. La *répartition dynamique de charge* consiste à déporter les calculs vers les ressources les plus rapides. Les applications indéterministes changent donc leur comportement selon l'importance de l'intrusion.



(a) exécution sans intrusion



(b) exécution avec intrusions

FIG. 1.3 – *Perturbation du comportement par des intrusions.*

1.2.2 Recouvrement calcul/communication

Une notion importante pour améliorer l'efficacité d'une application parallèle est le recouvrement entre calculs et communications. Le temps nécessaire à une communication avec un processus partenaire peut être mis à profit pour réaliser des calculs locaux. Si un processus ne peut se tenir actif en attendant la fin d'une communication, il doit laisser un autre processus occuper le processeur.

Définition 1.18 *Le* recouvrement des communications par des calculs *est l'utilisation des délais de communication pour effectuer des calculs sur le processeur.*

Des bibliothèques spécialisées sont développées pour intégrer efficacement les communications et les calculs [10]. Elles sont surtout dédiées aux applications régulières. Les applications irrégulières ont recours à des systèmes

dynamiques où les entités sont créées et détruites tout au long de l'exécution.

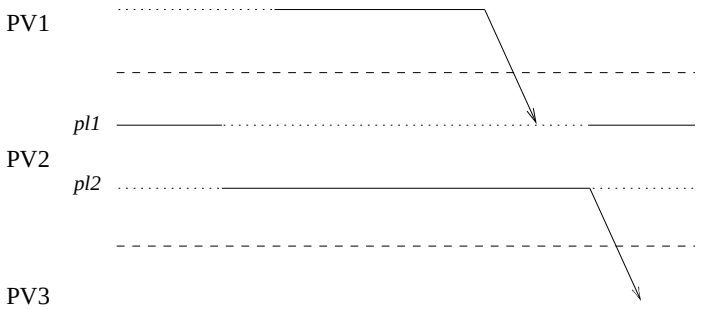


FIG. 1.4 – *Recouvrement calcul/communication.*

La figure 1.4 présente le cas où le processus $pl1$ du processeur virtuel PV2 attend une réception de message. Durant son attente, le processus $pl2$ occupe le processeur pour avancer son propre calcul. La coopération entre les processus peut faire perdre du temps à certains puisqu'ils risquent d'attendre un peu plus longtemps (comme le processus $pl1$ sur la figure 1.4). Toutefois, cela amène à augmenter le taux d'occupation globale de chaque processeur.

1.3 Mise au point dynamique

Pour mettre au point une application selon une approche dynamique, un programmeur doit en comprendre le comportement. L'aspect fonctionnel concerne l'adéquation du résultat produit avec les spécifications. Si le résultat obtenu ne correspond pas à celui attendu, l'application comporte une erreur fonctionnelle. L'aspect des performances concerne les temps de calcul nécessaires pour obtenir le résultat. La réalisation de l'application peut comporter un goulot d'étranglement. Le programmeur souhaitera alors en amoindrir les effets.

Pour les deux types d'erreurs, il faut extraire des informations d'une exécution, les analyser et en présenter l'analyse. Selon ses connaissances et la présentation du résultat de l'analyse, le programmeur peut demander une autre présentation, effectuer une autre analyse, extraire d'autres informations ou modifier l'application. À l'issue de cette démarche cyclique, les erreurs rencontrées sont éliminées.

Pour assister un programmeur dans la mise au point d'une application, des outils permettent d'observer le comportement fonctionnel et les perfor-

mances. Nous présenterons les moyens disponibles pour la mise au point d'une application parallèle.

1.3.1 Phases de l'approche cyclique

L'approche cyclique de mise au point vise à comprendre un peu mieux le comportement d'une application erronée lors de chaque itération. Une itération se compose de quatre phases : extraire des informations, analyser les traces, présenter le résultat de l'analyse et modifier l'application. La quatrième phase est optionnelle. Après la présentation du résulat de l'analyse le programmeur peut choisir de commencer une nouvelle itération en revenant sur l'une des trois premières phases. Il peut aussi choisir de terminer l'itération par la quatrième phase ; l'itération suivante commencera alors par la première phase.

Dans les débogueurs séquentiels usuels comme *gdb* [74], l'extraction d'informations, l'analyse et la présentation sont souvent combinées en une seule opération. L'extraction d'informations sur l'application mise au point se fait par des primitives du système d'exploitation. Un ensemble fixé de schémas d'extraction-analyse-présentation est offert au programmeur.

D'autres approches plus flexibles sont proposées, en particulier dans l'environnement Opium [20]. Dans ce cas, un programmeur peut profiter des scénarios déjà réalisés ou alors en définir de nouveaux, spécifiques à son application ou à sa démarche personnelle de recherche des erreurs. Pour chaque scénario, il spécifie les traces extraites, l'analyse réalisée et la présentation souhaitée. Cette séparation des différents niveaux permet d'étendre l'utilisation du même outil à différents modèles de programmation [21].

Nous détaillons ici les quatre phases d'une approche dynamique, indépendemment de la réalisation d'un outil particulier.

L'extraction d'informations s'effectue par des mécanismes du système d'exploitation ou par une version instrumentée de l'application mise au point.

Les informations obtenues du *système d'exploitation* sont de très bas niveau. Il est difficile au programmeur de lier ces informations au comportement observable de l'application. Une analyse automatisée permet de les lier à des entités du source.

L'instrumentation peut être réalisée de plusieurs manières :

– Dans le source de l'application ;

- Par le compilateur ;
- Dans une bibliothèque fonctionnelle ;
- Dans le code exécutable.

Les instructions de traçage sont insérées *dans le source de l'application* soit par un programmeur, soit par un préprocesseur. Le programmeur peut contrôler librement ce qu'il souhaite tracer. Le préprocesseur traduit le code source en un code équivalent instrumenté. Pablo [70], Tape/PVM [54] et POM [34] utilisent cette techique.

L'instrumentation *par le compilateur* nécessite de modifier le compilateur. En contrepartie, elle facilite l'intégration d'informations sémantiques dans les traces. Annai [79] et *qpt* [49] utilisent cette technique.

Selon les besoins, il peut suffir d'avoir des traces seulement des opérations *d'une bibliothèque fonctionnelle*. La fonctionnalité de traçage est latente dans la bibliothèque. La recompilation de l'application n'est pas nécessaire dans ce cas puisque le traçage est activé par des options d'exécution. Annai, PICL [33] et XPVM [46] offrent cette possibilité.

Enfin l'instrumentation *dans le code exécutable* ne requiert pas non plus de recompilation de l'application. Toutefois, tout comme l'instrumention d'une bibliothèque fonctionnelle, elle ne fournit pas d'accès aux informations sémantiques de l'application. PARADYN [65] exploite ce type d'instrumentation.

L'analyse des traces se base sur des filtres ou des requêtes qui recherchent des propriétés dans le comportement observé. Les filtres expriment des propriétés prédéfinies alors que les requêtes permettent d'exprimer également des propriétés spécifiques à la sémantique de l'application mise au point.

Bates et Wileden [5, 4] ont introduit l'abstraction comportementale (en anglais *behavioral abstraction*) pour la mise au point de programmes distribués. À l'aide d'un langage de description des événements, un programmeur peut définir à partir d'événements de bas niveau des événements de plus haut niveau afin de réduire le volume d'informations à présenter. Pour faciliter la description des abstractions, Kunz [47] propose un regroupement assisté d'informations concernant les processeurs et les événements. Les regroupements réalisés par des heuristiques peuvent être modifiés par un programmeur pour s'adapter plus finement à l'application mise au point.

La détection de propriétés globales se heurte au problème des propriétés instables. Ces propriétés peuvent redevenir fausses après avoir été vérifiées dans un état global cohérent. Cooper, Marzullo et Neiger [16, 59] ont proposé une méthode pour la détection de propriétés dans le treillis des états globaux cohérents. Ils définissent les modalités *POS* (s'il est possible de vérifier la propriété pour au moins une exécution) et *DEF* (si la propriété est vérifiée pour toutes les exécutions). Fromentin [30] définit pour les propriétés comportementales les modalités *SOME* (s'il existe existe un chemin dans le treillis des états globaux qui vérifie la propriété) et *ALL* (si tous les chemins dans le treillis des états globaux vérifient la propriété). À partir de ces modalités, il élargit les classes de propriétés détectables.

La présentation du résultat de l'analyse peut être simplement textuelle, utiliser le son ou afficher des graphiques (statiques ou animés).

Bien que la plus simple, la présentation textuelle peut attirer l'attention du programmeur sur certains détails par le recours aux couleurs ou à la surbrillance. Elle est surtout utilisée par les outils de correction des erreurs de programmation.

La présentation sonorisée, proposée par Francioni, Albright et Jackson [29], se base sur la capacité de l'oreille à reconnaître des motifs musicaux. Cette technique peut être utilisée pour attirer l'attention sur un ensemble restreint d'événements. Les caractéristiques de hauteur, de timbre et de durée du son associé à chaque événement doivent être choisies avec précaution pour ne pas conduire à un vague bruit. En raison de cette difficulté d'ajustement, elle est rarement proposée dans les outils de mise au point.

La présentation la plus fréquente après la présentation textuelle est la présentation graphique. L'outil Paragraph [36] s'est rapidement imposé comme référence dans le domaine. Parmi la trentaine de vues qu'il propose, les plus importantes sont le diagramme espace-temps, le diagramme de Gantt (qui résume l'activité des processeurs) et le diagramme de Kiviatt (qui résume la charge des processeurs). Toutefois un programmeur est limité à l'usage des vues prédéfinies. L'outil Pablo [70] a été développé dans l'objectif de permettre l'extension des vues aux besoins rencontrés pour chaque application mise au point.

Comme les outils Paragraph et Pablo, l'outil Paje [18] est orienté vers la correction des performances. Il permet aussi la définition de nouvelles visualisations selon les besoins rencontrés. Cet outil décompose clairement les différentes étapes exposées ici, en distinguant des modules spécialisés pour

l'acquisition de traces (souvent par lecture d'un fichier), pour l'analyse de ces traces et pour la présentation. Un scénario définit l'enchaînement des modules utilisés pour un besoin donné. Les scénarios peuvent être modifiés et enregistrés pour d'autres usages ultérieurs.

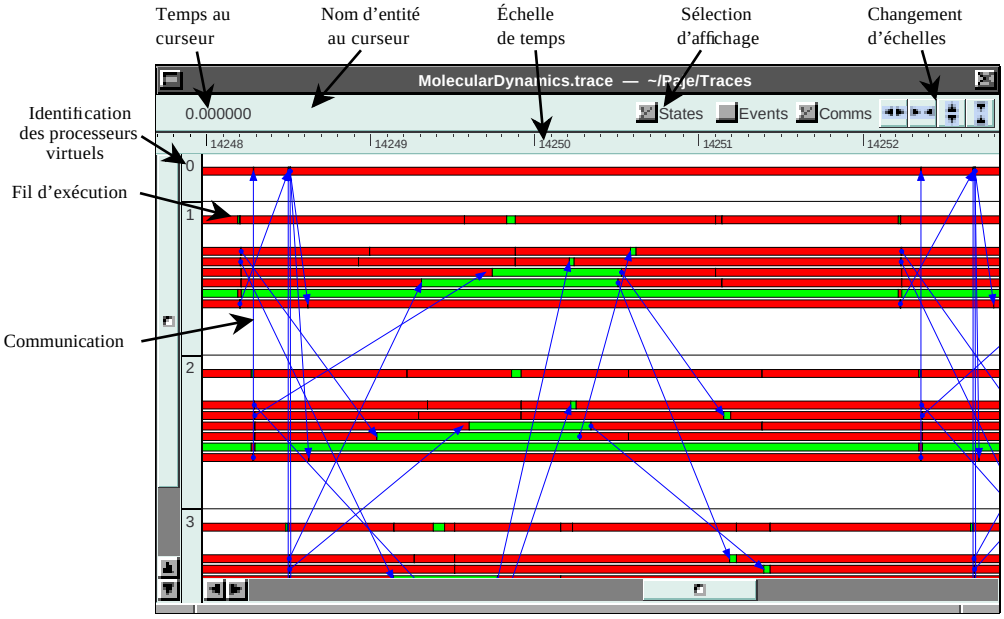


FIG. 1.5 – *Visualisation d'un diagramme espace-temps.*

La figure 1.5 présente une visualisation d'un diagramme espace-temps pour une application ATHAPASCAN-0. Chaque fil d'exécution est représenté par une barre horizontale dont la couleur reflète l'état du fil. Les processus légers sont regroupés selon le processeur virtuel surlequel ils s'exécutent. Le regroupement est réalisé par un module de placement des processus sur la vue. Les flèches entre les processus représentent les communications. Bien qu'elle constitue une aide à la compréhension, la visualisation n'est qu'un outil de présentation des résultats des analyses sur les informations extraites d'une exécution.

La modification de l'application permet de corriger une erreur détectée ou d'explorer la validité d'une hypothèse. Selon les outils et l'importance de la modification, l'application devra être recompilée. Un outil comme Thésée [41] (présenté au paragraphe 2.2.4) permet en outre de conserver un historique des versions du code et des traces obtenues pour chacune des versions.

1.3.2 Éléments de réalisation d'un atelier de mise au point

Dans ce paragraphe, nous esquissons les éléments à rassembler pour construire un atelier de mise au point dynamique d'applications parallèles. Deux aspects complémentaires sont présents dans un tel atelier. Il faut d'une part corriger les erreurs de programmation et d'autre part corriger les problèmes de performances. L'utilisation d'un débogueur séquentiel pour chaque processus lourd doit s'accompagner d'une extension des points d'arrêt. L'analyse de performances doit présenter des informations fiables et effacer l'intrusion induite par l'acquisition de ces informations.

Pour corriger les erreurs de programmation, un débogueur séquentiel peut être utilisé indépendemment chaque processus lourd. Le choix du débogueur doit prendre en compte le support des processus légers par l'outil. Pour mettre au point simultanément tous les processus lourds d'une application parallèle, plusieurs débogueurs séquentiels doivent collaborer. Pour réaliser des actions sur les processus, un programmeur peut soit donner indépendamment à chaque débogueur les commandes à exécuter, soit les distribuer par un mécanisme central. Les processus concernés par chaque commande sont alors regroupés dans un *contexte*[2]. Le contexte exprime ainsi la concentration du programmeur sur l'ensemble des processus de l'application ou sur un groupe dont il veut plus particulièrement comprendre le comportement.

Les points d'arrêt sont un moyen classique de contrôler le déroulement de l'exécution d'une application mise au point. L'utilisation d'un point d'arrêt a pour but de suspendre l'exécution d'une application lorsque certaines conditions sont vérifiées. La suspension peut concerner l'ensemble des processus d'une application, un groupe de processus (par exemple défini par un contexte) ou un seul processus (sur lequel la condition a été vérifiée). Lorsque l'ensemble des processus est concerné, l'application peut être suspendue soit dans un état précédent immédiatement la validation de la condition, soit dans un état suivant immédiatement la détection de la condition, soit dans un état ultérieur.

Lorsque l'exécution d'une application est suspendue dans un état suivant la détection de la condition, Miller et Choi [64] proposent, pour la programmation par échange de messages, un algorithme qui garantit la suspension

2. Il ne faut pas confondre le contexte d'évaluation des commandes de mise au point avec le contexte d'exécution d'un processus.

dans un état global cohérent. Pour sa part, Fromentin [30] base un mécanisme de points d'arrêt sur la détection à la volée de propriétés globales. Le *point d'arrêt causal*, défini par Fowler et Zwaenepoel [28], suspend l'application dans un état global cohérent précédent un événement. L'événement de suspension de chaque processus est calculé à partir de traces. Durant une exécution suivante de l'application, les processus sont suspendus dans l'état global désiré. Leu définit le *point d'arrêt vectoriel explicite* [51], où l'événement de suspension de chaque processus est explicitement donné par une horloge vectorielle. Cette horloge vectorielle correspond à celle d'une coupe dans le diagramme espace-temps. La responsabilité de la cohérence revient au programmeur qui définit le vecteur.

Pour corriger les problèmes de performances, le programmeur doit pouvoir connaître des quantités telles que les temps de calcul, les temps de communication, les fréquences d'appels des procédures et les goulots d'étranglement des communications. Dans une application parallèle, de mauvaises performances peuvent provenir aussi bien de procédures mal optimisées que des synchronisations entre processus. La gestion de l'ordonnancement des processus peut aussi conduire à des pertes d'efficacité.

Classiquement, les outils d'aide à l'amélioration de l'efficacité des applications présentent au programmeur un bilan du temps passé à exécuter chacune des procédures appelées. Cette analyse permet de classer les procédures selon l'ordre souhaitable de leur optimisation. De manière empirique, plus la procédure a consommé de temps sur l'ensemble d'une exécution, plus il semblera prioritaire de tenter de l'améliorer. Pour extraire une information fiable, les outils mis en œuvre sont obligés de procéder à de nombreux sondages qui perturbent fortement les temps d'exécution des procédures de l'application. Malony [57] et Maillet [55] proposent des méthodes de correction des perturbations induites par le traçage pour l'évaluation de performances.

Dans une application parallèle interviennent aussi les communications entre les processus. Même si le support d'exécution tente de les masquer par du calcul grâce à l'utilisation des processus légers, quelques aménagements peuvent être nécessaires de la part du programmeur. L'atelier de mise au point idéal doit pouvoir montrer comment le support d'exécution a réussi à masquer les délais de communication. Une mise en évidence des communications non-masquées aide le programmeur à apporter les modifications nécessaires à son code. Il peut jouer sur l'éclatement des trop grosses interactions ou le regroupement des trop petites.

1.3.3 Limitations dues aux intrusions

Comme le présente le paragraphe 1.2, les applications indéterministes sont sensibles aux intrusions. Celles-ci peuvent amener une application à changer complètement son comportement. Les intrusions peuvent venir de l'usage de mécanismes de traçage ou d'outils interactifs qui suspendent (ou ralentissent) l'exécution de certains processus d'une application mise au point. Sur une application parallèle indéterministe, une forte intrusion peut complètement bouleverser le comportement logique par des changements d'interactions entre certains processus. Selon les analyses effectuées, certains processeurs virtuels subiront plus d'intrusion que d'autres. Le déséquilibre introduit sera exploité par une application opportuniste qui disposera ses calculs de manière différente.

Les intrusions sont plus ou moins importantes selon les phases de l'approche cyclique où elles interviennent. L'extraction d'informations est obligatoire et doit donc tenter de minimiser son intrusion. Les phases suivantes peuvent se dérouler après l'exécution ou à la volée. Lorsque l'analyse des informations est réalisée à la volée, elle doit aussi minimiser son intrusion. Enfin la présentation est la phase la plus critique car elle peut nécessiter de suspendre l'exécution de l'application mise au point, pour synchroniser l'affichage des informations avec la capacité de compréhension d'un programmeur. Des choix effectués pour la réalisation de chaque phase dépendent les importances relatives des intrusions induites.

Si à chacune des exécutions qu'il observe, un programmeur se trouve confronté à des valeurs différentes, il ne pourra en déduire que l'indéterminisme de l'application. Toutefois, il sera totalement dépourvu de moyens pour en comprendre l'origine. Que ce soit pour corriger les erreurs de programmation ou pour corriger les problèmes de performances, un programmeur a besoin d'observer un même comportement à chaque itération de la mise au point cyclique. Les outils, qui ne garantissent pas l'équivalence des exécutions, ne permettent pas de mettre au point facilement les applications indéterministes.

La technique de réexécution déterministe, présentée au chapitre suivant, permet de reproduire des exécutions équivalentes à une exécution sur laquelle des traces peu intrusives ont été enregistrées. Seule la première phase d'extraction d'informations minimales est réalisée sur une exécution initiale. La démarche cyclique se poursuit dans un mode d'exécution particulier où toutes les exécutions sont équivalentes à l'exécution initiale. Les intrusions induites

n'ont alors plus de conséquence sur le comportement des exécutions observées. Des ateliers de mise au point exploitent la réexécution déterministe pour offrir aux programmeurs la possibilité de mettre au point également les applications indéterministes. Deux exemples d'ateliers sont présentés au paragraphe 2.4.

1.4 Conclusion

La mise au point dynamique selon une approche cyclique nécessite d'observer à chaque itération le même comportement de l'application à mettre au point. L'indéterminisme des exécutions limite les possibilités d'utiliser les outils présentés au paragraphe 1.3. Pour se ramener au cas de la mise au point d'applications déterministes, il faut employer la technique de réexécution déterministe. Cette thèse est centrée sur la réalisation d'un mécanisme de réexécution déterministe pour ATHAPASCAN-0a. Ce noyau exécutif exploite un modèle procédural parallèle. Après la présentation de la réexécution déterministe dans le chapitre suivant, le chapitre 3 présente notre formalisation pour le modèle procédural parallèle.

Chapitre 2

Réexécution déterministe

Parmi les principaux problèmes que pose la mise au point de programmes parallèles, l'indéterminisme des exécutions rend la correction des erreurs extrêmement problématique. Une erreur peut en effet n'apparaître que de façon occasionnelle ou encore disparaître dès qu'un outil de mise au point ou de traçage est mis en œuvre. De telles erreurs peuvent également se produire pour les programmes séquentiels, bien que moins fréquemment. En effet, la distribution des exécutions ajoute des sources d'indéterminisme. Ce type d'erreurs, appelées erreurs furtives, est très sensible aux intrusions et aux perturbations de l'environnement. La sensibilité aux intrusions rend souvent difficile l'utilisation de la mise au point dynamique présentée au paragraphe 1.3.

Une méthode classique pour résoudre ce problème est d'enregistrer *l'ordre* des événements de synchronisation et de communication durant une exécution puis de réexécuter le programme parallèle en respectant l'ordre enregistré, c'est-à-dire de façon *déterministe* relativement à l'enregistrement initial. Cette méthode est appelée «*Instant Replay*» par ses auteurs, LeBlanc et Mellor-Crummey [50]. Elle a été utilisée dans plusieurs environnements dont les modèles de programmation sont différents. Quatre réalisations sont présentées, qui traitent la programmation par mémoire partagée, par échange de messages, par automates distribués et par objets persistants partagés. Notre adaptation au modèle procédural parallèle est présentée au chapitre 3. La réexécution déterministe sert de base à certains outils de mise au point comme ceux présentés au paragraphe 2.4.

2.1 Principe général

La reproduction d'une exécution peut se faire selon deux techniques : dirigée par les données ou dirigée par le contrôle. Dans les deux cas, deux phases se succèdent : lors de la première phase des informations sont collectées, lors de la seconde ces informations permettent de reproduire le comportement observé. Pour la première, ce sont les données enregistrées au cours d'une exécution qui dirigent les réexécutions. Pour la seconde, c'est l'ordre d'événements qui permet de diriger les réexécutions. Après une description sommaire des deux techniques, suit une présentation détaillée du principe de la reproduction dirigée par le contrôle qui a servi de base à cette étude. Par rapport à la reproduction dirigée par les données, elle permet de limiter la perturbation provoquée par l'observation de l'application. Elle a été adaptée à des modèles de programmation aussi différents que ceux présentés au paragraphe 2.2. Une formalisation permet d'étudier des améliorations au principal initial. Elle est présentée au paragraphe 2.3 ainsi que deux améliorations par Netzer et Miller [68] et Levrouw, Audenaert et Van Campenhout [53].

2.1.1 Rappel historique

La réexécution dirigée par les données Dès 1982, Curtis, Jones, Barkan et Wittie [17, 44] proposent le système BugNet basé sur une reproduction dirigée par les données. Il est conçu pour la mise au point d'applications réparties sur un réseau de machines communiquant par messages. Il enregistre dans un fichier de traces le contenu des entrées et des messages réalisés par les processus de l'application. Ce fichier peut être analysé par le programmeur ou fournir les valeurs des entrées pour réexécuter séparément chaque processus. Cette technique permet de mettre au point des programmes distribués en isolant les processus les uns des autres. Un programmeur peut ainsi analyser le comportement d'un processus qui s'était avéré erroné sans surcharger le réseau avec les autres processus. Comme les réexécutions reçoivent les mêmes entrées que l'exécution erronée, le même comportement est observé à chaque réexécution. Ainsi BugNet permet de mettre au point des programmes indéterministes selon une approche cyclique de mise au point dynamique. Jusqu'alors, les programmeurs n'avaient pas d'outils pour traiter de tels programmes.

Cette méthode s'applique à des programmes distribués sur des architectures où le coût des communications est élevé et les échanges rares. Avec des

architectures où les communications deviennent plus fréquentes, le fichier de traces peut grossir très vite et introduire une dégradation notable des performances. La forte augmentation de l'intrusion ne permet pas toujours de piéger les erreurs furtives. Le principal inconvénient à retenir contre cette méthode est qu'elle ne fournit que la possibilité de réexécuter un processus en isolation du reste de l'application. Cet inconvénient peut cependant devenir un véritable avantage car il permet de se concentrer sur le comportement d'un processus suspect. Pour réexécuter un groupe de processus, il est nécessaire de disposer d'un temps global cohérent sur le réseau. Ce temps global permet de synchroniser l'avancement indépendant de chaque processus.

La réexécution dirigée par le contrôle Le grand changement introduit par LeBlanc et Mellor-Crummey [50] fut de ne conserver que *l'ordre* des événements de synchronisation et non plus leur contenu. C'est le principe de la reproduction dirigée par le contrôle : plutôt que de conserver les valeurs utilisées par chaque processus c'est l'ordre des opérations de synchronisation qui est conservé. Les valeurs sont recalculées lors de chaque réexécution de l'application en forçant l'ordre des synchronisations selon l'ordre enregistré. Les exécutions reproduites sont équivalentes à l'exécution enregistrée, en l'absence de conflit ouvert pour l'accès aux données. Un conflit ouvert est un accès concurrent à des données, non protégé par un appel à des primitives de synchronisation entre les processus [1]. Pour les programmes qui protègent leurs accès aux données partagées, les événements observés pour une réexécution sont les mêmes que ceux observés pour l'exécution initiale. Contrairement à la méthode précédente où chaque processus devait être réexécuté séparément, cette méthode de reproduction concerne tous les processus qui doivent recalculer les valeurs qui ne sont plus conservées. Cette forme de reproduction est en fait une réexécution de toute l'application, même pour observer un seul processus suspect. En contrepartie de cette contrainte, le volume des traces enregistrées est réduit à une identification de chaque événement de synchronisation enregistré. L'indentification d'un événement n'occupe que quelques octets. Le volume enregistré n'est effectivement réduit que si la taille des échanges est supérieure à l'information d'identification. La réduction peut être négligeable si les échanges sont de petite taille (quelques octets) mais elle peut être considérable pour des messages de grande taille (quelques dizaines d'octets ou plus).

1. Audenaert et Levrouw [2] ont étudié de tels systèmes.

2.1.2 Principe initial de la réexécution

La réexécution déterministe se base sur l'hypothèse de déterminisme des processus séquentiels s'ils sont soumis aux mêmes données en entrée. Il est admis que l'indéterminisme pourchassé est celui introduit par les interactions entre les processus. Durant l'exécution, l'ordre relatif des événements significatifs est enregistré mais pas les données associées à ces événements. Les événements significatifs sont définis selon le modèle de programmation. Il s'agit des réceptions de message, des accès à des variables partagées ou des activations de transitions. Pour chaque modèle présenté dans le paragraphe 2.2, nous définissons ces événements. Les événements significatifs sont les événements de synchronisation entre les processus. C'est donc l'ordre des interactions qui est enregistré. Comme il n'est pas nécessaire de conserver le contenu des interactions entre processus, cette approche est plus économique en temps et en place pour sauver les informations nécessaires à la réexécution qu'une approche basée sur l'enregistrement du *contenu* des interactions. Pour réaliser l'enregistrement et la réexécution, le mécanisme est réparti entre les différents processus, sans utilisation d'horloges synchronisées ni de temps logique global évitant ainsi une synchronisation globale des événements. Chaque processus est responsable de l'enregistrement de l'ordre des événements significatifs qu'il observe puis de leur réexécution suivant cet ordre initial. Pour effectuer l'enregistrement, chaque processus est doté d'une structure de données organisée comme une bande. Sur cette bande, il note la séquence des événements de synchronisation auxquels il participe. Cette bande servira de guide lors de la réexécution.

2.1.3 Extension du principe de réexécution

Le modèle initial considère comme des objets partagés des structures de données dont l'accès doit être atomique. Tous les éléments composants la structure doivent être lus ou écrits au cours d'une même opération. L'atomicité des accès garantit la cohérence des informations conservées dans la structure. Selon ce principe, les objets partagés sont des entités passives qui subissent les actions des flots d'exécution sans exercer de contrôle. Dans ce cas, les flots d'exécution sont responsables du contrôle de l'accès aux objets partagés. Ce sont eux qui enregistrent l'ordre des événements nécessaires à la réexécution.

Il est possible de considérer des modèles de programmation dans lesquels les objets partagés sont actifs dans le sens où ils sont des espaces d'exécution

pour des flots parallèles. Ces objets se comportent comme des processeurs virtuels qui vont supporter l'exécution des flots logiques d'une application. Ils offrent certains services aux applications. Dans ce cas, les objets partagés sont partiellement responsables du contrôle des flots d'exécution. Ces derniers peuvent être considérés comme des clients par l'objet dont le rôle est alors de garantir à chaque flot l'accès aux données qu'il contient. L'objet distribue des autorisations aux flots, provoquant ainsi une forme de sérialisation partielle des accès. L'attribution d'autorisation d'accès peut se faire selon deux politiques : l'attribution de tickets ou l'alignement dans une file d'attente. Dans la solution avec attribution de tickets, les flots d'exécution sont capables de mémoriser la valeur du ticket distribué pour chaque accès. Dans la solution avec des files d'attente, le flot ne peut savoir quelle est sa position, il est alors de la responsabilité de l'objet de mémoriser l'ordre d'accès des différents flots clients. Cette solution est, d'une certaine manière, la symétrique du principe initial où les objets sont passifs et les flots responsables de la mémorisation de l'ordre des accès.

2.2 Quelques exemples de réalisations

La technique de réexécution déterministe a déjà été utilisée dans plusieurs environnements différents. Elle a été appliquée à des modèles où les interactions entre processus se font par mémoire partagée [50] ou par échange de messages [52, 68]. Elle a été appliquée aussi au modèle du langage de spécification Echidna [38] et au modèle Guide [41] avec partage d'objets[2]. Nous avons sélectionné les réalisations pour quatre modèles de programmation. Cette sélection montre la possibilité d'adapter la technique à des cas très différents. Notre adaptation au modèle procédural parallèle contribue à la diversité des modèles traités.

2.2.1 «Instant Replay» (Partage de mémoire)

La première réalisation [50] a été présentée par LeBlanc et Mellor-Crummey en 1987. Elle est appliquée à un modèle d'interaction entre les processus par partage de mémoire. Chaque structure partagée est dotée d'un numéro de version et d'un compteur du nombre de lectures effectuées. L'historique des modifications sur cette structure est représenté par la séquence totalement

2. Il s'agit ici des objets de la programmation par objets et non des objets de LeBlanc et Mellor-Crummey, qui sont des structures de données dont l'accès doit être atomique.

ordonnée des numéros de versions. Un protocole de sérialisation des opérations de lecture et d'écriture permet de garantir un résultat valide pour chaque opération sur une structure partagée. Un tel protocole peut être celui des lecteurs-rédacteurs (CREW : Concurrent Read Exclusive Write) utilisé par LeBlanc et Mellor-Crummey, mais d'autres peuvent être définis selon le contexte.

Durant l'enregistrement, un ordre partiel (car il n'y a pas de sérialisation des lectures concurrentes) des accès à chaque structure est conservé. Chaque processus enregistre, sur sa bande, le numéro de la version de la structure qu'il accède et, pour un accès en écriture, le nombre de lectures effectuées sur cette version. Un lecteur devra attendre que la version de la structure soit celle qu'il a accédée lors de l'exécution enregistrée. Un rédacteur devra attendre que tous les lecteurs de la version qu'il veut modifier aient effectué leur lecture de la structure. Ce traitement de l'historique des accès pour chaque processus permet de garantir, à la réexécution, l'accès à la même valeur de la structure.

2.2.2 Échange de messages

Pour réaliser l'interaction entre les processus, il est possible d'utiliser l'échange de messages. Pour les différencier, Leu et Schiper [52] ont besoin qu'un identificateur de message unique soit attribué de manière déterministe à chaque message émis. Selon les environnements, l'ordre de livraison des messages peut être identique à l'ordre d'émission ou n'avoir aucune contrainte. Les événements à tracer sont uniquement les réceptions pour le cas où les ordres d'émission et de réception sont identiques. dans l'autre cas, il faut traiter aussi les événements d'émission.

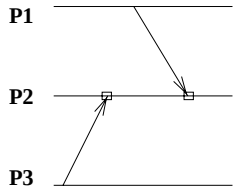


FIG. 2.1 – *Enregistrement des réceptions de messages.*

Plutôt que d'enregistrer le numéro de version d'une structure, il faut enregistrer l'identificateur de message. Un identificateur est attribué à chaque message par le processus émetteur. Il caractérise, de manière unique et déterministe, le message qui le porte. Dans l'hypothèse d'ordres d'émission

et de réception identiques, l'ordre effectif des réceptions est enregistré pour chaque processus (voir figure 2.1). L'historique des réceptions contient les identificateurs des messages dans l'ordre observé lors de l'exécution initiale. La réexécution consiste à forcer les processus à considérer la réception des messages selon l'ordre donné par l'historique.

2.2.3 Erebus (Automates distribués)

Erebus est un outil de mise au point réalisé par Hurfin, Plouzeau et Raynal [38] pour le langage Echidna [43], un sous-ensemble du langage de spécification Estelle [9]. Par rapport à la programmation par échange de messages, le langage Echidna structure explicitement les processus comme des automates. Chaque processus Echidna est un automate décrit par un ensemble de transitions. Il dispose aussi de variables locales et de files de messages reçus (voir figure 2.2). Chaque transition est composée d'une garde et d'un bloc d'actions. L'exécution de chaque processus est contrôlée par un ordonnanceur dont le rôle consiste à répéter la séquence suivante :

1. évaluation de toutes les gardes,
2. choix d'une transition à activer, parmi celles dont la garde est évaluée à *vrai*,
3. exécution du bloc d'actions de la transition choisie.

Si la garde de la transition choisie comporte la réception d'un message en provenance d'un processus déterminé sur une file spécifiée, le premier message de la file est consommé. L'activation d'une seule transition parmi les transitions prêtes introduit de l'indéterminisme. Selon le résultat de ce choix, l'enchaînement des états, peut varier d'une exécution à l'autre. Cette séquence des états devra être reproduite lors des réexécutions.

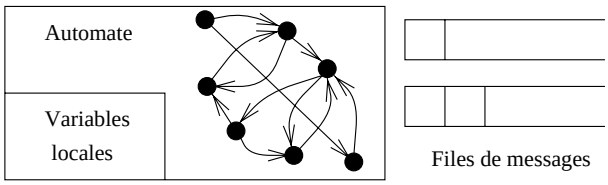


FIG. 2.2 – *Un processus Echidna.*

Durant la phase d'enregistrement, chaque ordonnanceur conserve l'ordre selon lequel les transitions sont activées. La reproduction est guidée par les

identités des transitions activées lors de l'exécution initiale. Chaque ordonnanceur essaie d'activer la transition donnée par la séquence enregistrée. Si la transition n'est pas activable (garde évaluée à *faux* lors de la reproduction), l'ordonnanceur attend qu'elle le soit. Cette contrainte permet de respecter l'ordre initial des interactions entre processus. Elle est suffisante car la réception d'un message indique explicitement l'expéditeur dont la transition attend une communication. Il n'est donc pas nécessaire de reproduire également l'ordre de réception des messages.

2.2.4 Thésée (Partage d'objets persistants)

Un autre mode d'interaction entre processus est le partage d'objets persitants dans le système Guide [3, 19]. Jamrozik a développé la réexécution déterministe pour ce modèle dans le noyau Thésée [41]. Dans le modèle Guide, les processus utilisent des objets persistants. La durée de vie de ces objets est supérieure à celle des applications, qui les manipulent. Les événements du modèle d'exécution de Guide concernent les opérations sur les objets, les créations et disparitions d'entités (objets, activités et domaines principalement, voir figure 2.3), la migration d'objets et l'extension d'activités sur d'autres sites. La migration d'un objet intervient si une activité de l'application veut l'utiliser alors qu'il est libre mais sur un autre site. L'extension d'activité intervient dans le cas où l'objet convoité est lié sur un autre site, l'activité s'étend alors sur ce site.

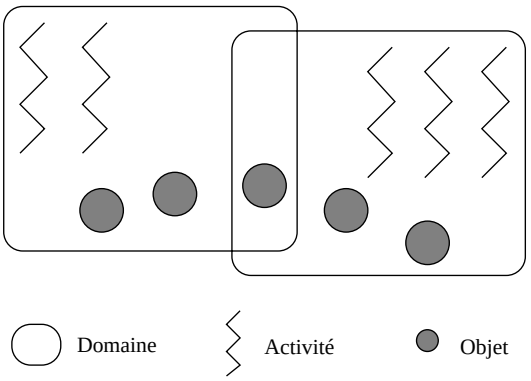


FIG. 2.3 – *Structure d'une exécution dans le système Guide.*

Le partage d'objets entre applications impose de placer l'application à mettre au point en isolation par rapport aux autres applications du système. Comme elle manipule des objets perssitants, le risque existe de la voir dé-

truire ou corrompre des objets essentiels à d'autres applications. Les objets partagés sont remplacés par des copies de travail pour éviter de perturber l'exécution des autres applications et, réciproquement, éviter la perturbation due à l'exécution des autres applications. La perturbation considérée ici concerne le comportement logique qui risque de subir les conséquences d'actions sur des objets partagés.

Le noyau de réexécution Thésée permet de conserver automatiquement l'état des objets du contexte initial d'exécution et leur localisation initiale. L'historique de l'exécution est composé des historiques locaux des nombreuses primitives du système Guide, sur chaque site visité par l'exécution. Des processus spécialisés, différents de ceux de l'application, récupèrent les événements signalés par le système Guide sur chaque site visité. Cette forme d'instrumentation permet de tracer toute application, sans recompilation.

Thésée permet la reproduction de l'exécution à partir des informations conservées dans l'historique. Le traçage des appels de méthode permet connaître les appels effectués et les objets utilisés. Lors de la première utilisation d'un objet, Thésée enregistre son état initial et crée une copie sur laquelle travaillera l'application mise au point. L'enregistrement des accès aux objets permet de reproduire les migrations d'objets et les extensions d'activités entre les sites. La disparition des objets se reproduit au même moment de leur cycle de vie lors des réexécutions.

2.3 Formalisation et améliorations

2.3.1 Relations entre les événements

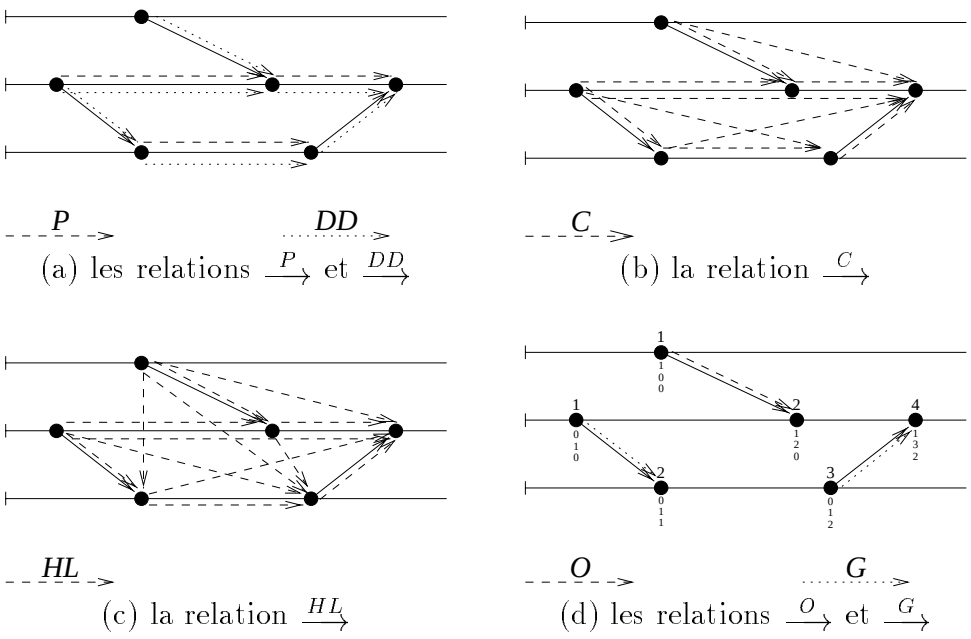
En fait, chacun des systèmes présentés au paragraphe précédent implante le traçage d'une même relation particulière entre les événements. Il s'agit de la relation de causalité directe $\prec_D$ (définition 1.9, notée dans ce paragraphe par $\xrightarrow{DD}$) qui définit la dépendance directe entre deux événements. Deux événements e_i^p et e_j^q sont *successifs* sur un même processus ($e_i^p \xrightarrow{P} e_j^q$) si et seulement si $e_i^p \xrightarrow{DD} e_j^q$ avec $p = q$ et $i = j - 1$. Pour certains modèles, la relation $\xrightarrow{P}$ entre les événements successifs d'un même processus n'a pas besoin d'être enregistrée. L'enregistrement de la relation $\xrightarrow{DD} \setminus \xrightarrow{P}$ permet de reproduire des exécutions équivalentes à l'exécution enregistrée.

Cependant, il est possible de réduire encore la taille de la relation enregistrée. Netzer et Miller [68] ont montré que la relation minimale, pour réexécuter un programme parallèle, est la relation $\xrightarrow{O}$. Elle se calcule à par-

tir de la relation de causalité $\prec$ (définition 1.10, notée dans ce paragraphe par $\xrightarrow{C}$). On a $e_i^p \xrightarrow{O} e_j^q$ si et seulement si il existe e_k^q et e_h^r tels que $e_j^q\ e_k^q$, $e_h^r \xrightarrow{DD} e_k^q$ avec $e_h^r \not\prec e_i^p$ et $e_i^p \not\prec e_h^r$. La relation $\xrightarrow{O}$ est donc incluse dans $\xrightarrow{DD} \setminus \xrightarrow{P}$. Son calcul nécessite le recours à une horloge vectorielle (définition 1.12) pour obtenir la relation de causalité $\xrightarrow{C}$.

Toutefois, la relation optimale $\xrightarrow{O}$ est coûteuse à calculer. Levrouw, Audenaert et Van Campenhout ont étudié une relation $\xrightarrow{G}$ calculée à partir de la relation $\xrightarrow{HL}$, donnée par l'horloge de Lamport (définition 1.11). Pour e_i^p et e_j^q, $e_i^p \xrightarrow{HL} e_j^q$ si et seulement si $HL(e_i^p) < HL(e_j^q)$. La relation $\xrightarrow{G}$ lie deux événements e_i^p et e_j^q si et seulement si il existe e_k^q tel que $e_k^q \xrightarrow{P} e_j^q$ avec $HL(e_k^q) \leq HL(e_i^p) - 2$. La relation $\xrightarrow{G}$ est aussi incluse dans $\xrightarrow{DD} \setminus \xrightarrow{P}$.

Ces différentes relations sont représentées pour une exécution simple sur la figure 2.4.



(a) les relations $\xrightarrow{P}$ et $\xrightarrow{DD}$ (b) la relation $\xrightarrow{C}$

(c) la relation $\xrightarrow{HL}$ (d) les relations $\xrightarrow{O}$ et $\xrightarrow{G}$

FIG. 2.4 – *Différentes relations d'ordre entre les événements.*

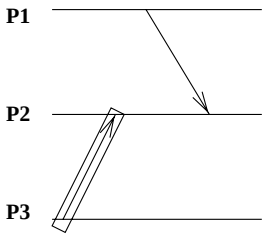
2.3.2 Amélioration basée sur les horloges vectorielles

Dans leur modèle, Netzer et Miller [68] se sont attachés à réduire le volume des traces prises en détectant les conflits de réception entre messages. Au lieu de tracer toutes les réceptions de messages (comme cela est fait classi-

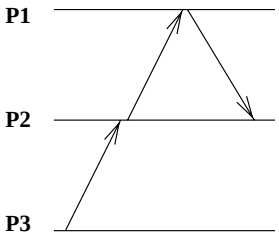
quement), leur solution consiste à détecter les conflits entre messages et à ne tracer que ces messages en conflit. Cette détection permet de ne retenir que les messages dont l'ordre de réception crée l'indéterminisme de l'exécution.

Chaque message est porteur d'une horloge vectorielle qui sert à éliminer l'enregistrement de messages non concurrents. La figure 2.5 présente trois instantanés de l'exécution. Elle illustre le comportement du processus **P2** lors de la réception du dernier message (celui le plus à droite). Le cas (a) est un conflit car l'un des deux messages pouvait arriver avant l'autre. Dans le cas (b) il existe une chaîne causale [48] entre les deux réceptions de messages. Les messages tracés sont ceux qui auraient pu être reçus par le processus selon un ordre différent.

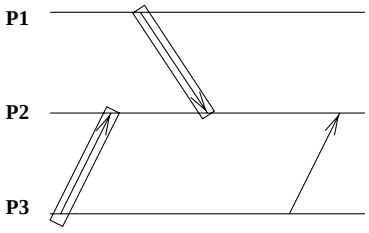
Ainsi pour (a), la réception du message en provenance de **P1** entraîne l'enregistrement de la réception du message en provenance de **P3**, puisque ces deux messages sont en conflit. Pour (b), il n'existe pas de message en conflit et donc il n'y a pas de trace générée. Le cas (c) reprend l'exécution présentée par le cas (a). La réception du second message en provenance de **P3** provoque le traçage du message en provenance de **P1**. Comme toute technique basée sur les horloges vectorielle, cette technique impose de connaître *a priori* le nombre de processus participant à l'exécution de l'application parallèle.



(a) le message encadré est tracé



(b) aucun message n'est tracé



(c) le message en provenance de **P1** est tracé

FIG. 2.5 – *Enregistrement des messages en conflit.*

2.3.3 Amélioration basée sur l'horloge de Lamport

Pour leur part Levrouw, Audenaert et Van Campenhout [53] ont choisi une voie opposée puisqu'ils considèrent la relation $\xrightarrow{HL}$ définie par les horloges de Lamport [48]. Cette approche va à l'encontre de l'intuition car la relation $\xrightarrow{HL}$ est moins précise que la relation $\xrightarrow{C}$ et contient donc plus d'événements en relation (en effet $\xrightarrow{DD} \subset \xrightarrow{C} \subset \xrightarrow{HL}$). Ce choix introduit davantage d'événements en relation.

La réduction du volume des traces est obtenue en traçant la relation $\xrightarrow{G}$ (voir figure 2.4). Dans cette relation, se trouvent les événements qui introduisent une rupture de la séquence locale des valeurs de l'horloge de Lamport sur un processus. Les valeurs de l'horloge ont une différence strictement supérieure à 1. Cette méthode limite, bien plus que la méthode de Netzer et Miller, le nombre de traces nécessaires pour conserver l'ordre défini par la relation $\xrightarrow{DD}$. Elle présente de plus le grand avantage de l'économie de la gestion de l'horloge. Au lieu d'un vecteur dont le nombre d'éléments est égal au nombre de processus du programme, il n'y a plus qu'un seul scalaire à manipuler. Le nombre de processus peut varier au cours de l'exécution de l'application sans nécessiter d'augmenter ou de réduire le volume des informations de contrôle.

2.3.4 En résumé

La réexécution déterministe est basée sur la reproduction de la $\xrightarrow{DD}$ entre les événements d'une exécution. Alors que la première amélioration garantit une réexécution au même coût que l'enregistrement, la deuxième approche dégrade légèrement les performances de la réexécution. L'introduction des contraintes supplémentaires se remarque lors de la réexécution puisque chaque événement ne peut se produire que si tous les autres processeurs ont produit tous les événements dont l'horloge de Lamport est inférieure. Sur une architecture à mémoire partagée, le surcoût dû à la réexécution reste toutefois limité. Les auteurs rapportent un surcoût inférieur à 9 %, ce qui est tout à fait acceptable lors d'une session de mise au point.

Durant la première réexécution, il est en fait possible de tracer à nouveau le programme afin d'en retirer une trace plus importante qui permettra de rejouer la relation $\xrightarrow{DD}$ directement. Il est à noter enfin que les contraintes supplémentaires introduites par la relation $\xrightarrow{HL}$ seraient très pénalisantes pour la réexécution d'un programme sur architecture à mémoire distribuée en raison de la synchronisation globale de tous les processus après chaque événement. Cette amélioration n'a pas encore été tentée pour la programmation par

échange de messages, en raison du coût attendu.

2.4 Exemples d'ateliers de mise au point

Cette partie présente succintement deux ateliers de mise au point basés sur la réexécution déterministe. Ils intègrent la correction des erreurs et la correction des performances. L'atelier ParaRex est dédié à la programmation par échange de messages sur machine Intel iPSC/2. L'atelier d'Annai traite les programmations par parallélisme de données (en anglais *data parallelism*) et par échange de messages. Il est disponible sur plusieurs plateformes.

ParaRex [51] a été développé par l'intégration d'un dévermineur symbolique (DECON [39] sur machine iPSC/2) et d'un outil de visualisation (ParaGraph [36]) autour du mécanisme de réexécution déterministe. *Annai* [15] est un environnement de programmation qui comporte deux outils intégrés, *Performance Monitor and Analyser* (PMA) et *Parallel Debugging Tool* (PDT). PMA traite de la visualisation et de l'analyse des performances. PDT permet quant à lui le déverminage de programmes HPF [37, 1], C ou Fortran avec des appels aux primitives de communication MPI [63]. Nous avons retenu ces deux ateliers car ils présentent deux approches possibles. ParaRex intègre des outils disponibles pour une machine spécifique alors que pour Annai l'atelier est intégré à l'environnement de programmation.

2.4.1 ParaRex

Cette approche réutilise des outils existants sur la machine cible pour les intégrer dans un atelier de mise au point. L'originalité réside dans la manière de les amener à coopérer à partir du mécanisme de réexécution déterministe. Le programmeur peut ainsi suivre la réexécution déterministe de son programme à la fois au niveau macroscopique (étapes des algorithmes) par l'animation ParaGraph et au niveau microscopique (instructions exécutées) par DECON.

Le dévermineur DECON a été développé par Intel spécifiquement pour sa machine. Outre les classiques possibilités de débogage séquentiel des processus, il offre des contrôles spécifiques au déverminage de programmes parallèles. Il intègre la gestion des contextes pour appliquer les commandes à un ensemble de processus. Comme il est dédié à la machine iPSC/2, il permet également d'accéder aux tampons de communication du système pour en analyser les informations relatives aux messages. Les processus suspendus

en attente de messages peuvent être connus de la même manière. Cet outil permet de mettre au point efficacement les programmes déterministes selon une approche cyclique dynamique.

ParaGraph est un outil de visualisation qui permet l'animation et l'analyse des performances de programmes parallèles basés sur l'échange de messages. La visualisation exploite une trace récoltée, par enregistrement dans un fichier ou à la volée, lors de l'exécution du programme. Les événements (envois et réceptions de messages), qui composent cette trace, sont essentiellement relatifs à la communication entre les processus. L'interface graphique propose une large palette de vues différentes des informations extraites de la trace (entre autres diagrammes espace-temps et diagrammes de Kiviatt). À tout instant, le programmeur peut suspendre et redémarrer l'animation ou encore opter pour le mode pas-à-pas. La trace peut être transmise à ParaGraph par un fichier ou bien à la volée, durant l'exécution du programme. Cette dernière possibilité est utilisée par ParaRex. Ceci permet la visualisation au cours d'une session de déverminage.

ParaGraph requiert un ordre total sur les estampilles de temps des événements visualisés. Le temps global du système est obtenu par synchronisation des horloges des nœuds avant l'exécution du programme. Comme la dérive des horloges est infime sur la machine iPSC/2, cette solution convient à la plupart des programmes. ParaGraph peut donc présenter des vues cohérentes des événements des programmes. Le mécanisme de traçage enregistre avec chaque événement sa date d'occurence. Lors de la réexécution, cette date est l'estampille de l'événement. Après l'envoi de chaque événement à la visualisation, le mécanisme de réexécution déterministe suspend le processus jusqu'à la réception de l'acquittement de la visualisation de l'événement. Afin d'ordonner les événements à visualiser à la volée, ParaGraph doit connaître les estampilles des deux derniers événements de chaque processus. Le mécanisme de réexécution déterministe lui fournit avec chaque événement l'estampille du prochain événement sur le processus (cette estampille est dans la trace). Le programmeur peut donc suspendre l'exécution de son programme par arrêt de la visualisation ou par un point d'arrêt positionné en utilisant le dévermineur.

2.4.2 Annai

La conception des outils de mise au point d'Annai supporte un modèle de programmation de «haut niveau» avec HPF et un modèle de programmation

de «bas niveau» avec la communication par MPI. L'environnement Annai n'est pas dédié à une machine particulière. Outre les fonctionnalités classiques des débogueurs séquentiels, PDT (Parallel Debugging Tool) offre la réexécution déterministe, des points d'arrêt distribués et la visualisation des données distribuées. Pour sa part, PMA (Parallel Monitoring and Analysis) gère les aspects liés à l'évaluation de performances.

Un processus central pilote pour chaque processus lourd de l'application un débogueur séquentiel. Il présente au programmeur une interface unique pour l'ensemble. Les données distribuées peuvent être présentées selon plusieurs vues mettant en relief la répartition sur les différents processus. Les conflits d'accès sont détectés lors de l'enregistrement des traces pour la réexécution déterministe. Plusieurs mécanismes de points d'arrêt sont implantés. Tout d'abord les points d'arrêt locaux qui ne mettent en jeu qu'un processus. L'exécution du processus est suspendu lorsque la condition est vérifiée, comme pour un point d'arrêt dans un programme séquentiel. Les points d'arrêt distribués sont de deux types : $\exists$ pour détecter une condition sur un des processus et $\forall$ pour détecter une condition globale sur tous les processus. Pour un point d'arrêt de type $\exists$, l'exécution du groupe de processus est suspendue lors de la validation de la condition d'arrêt sur l'un des processus. La condition doit être vraie sur tous les processus dans le cas d'un point d'arrêt de type $\forall$.

L'instrumentation de l'exécution peut être choisie selon les besoins du programmeur par l'intermédiaire de l'interface commune. Différentes vues sont présentées, de la vue générale de l'ensemble de l'application aux vues détaillées concernant uniquement un processus ou même une procédure. Les communications font partie des sources de perte d'efficacité les plus fréquentes. C'est pourquoi des vues particulières leur sont dédiées. Outre la présentation des événements survenus au cours de l'exécution, l'outil permet la synthèse de statistiques sur les temps de calcul, de communication, de gestion du parallélisme et d'inactivité.

2.5 Conclusion

Ces différentes réalisations montrent que la réexécution est possible avec des modèles de programmations très divers. La reproduction dirigée par le contrôle, ou réexécution, présente l'avantage d'introduire une faible perturbation de l'exécution observée. L'essentiel de la méthode de reproduction dirigée par le contrôle est de conserver la trace de l'ordonnancement de tous

les événements participant à la synchronisation des activités parallèles. L'effort essentiel réside dans l'identification des événements de synchronisation entre processus dont il est nécessaire de conserver *l'ordre* pour le reproduire. Le chapitre suivant présente l'étude d'un modèle de programmation parallèle par appel de procédure à distance. Les événements de synchronisation de ce modèle sont liés au traitement des appels de procédures. Une application concrète est proposée pour le noyau exécutif ATHAPASCAN-0a dans le chapitre 4.

Chapitre 3

Formalisme procédural

Notre travail de formalisation d'un modèle de programmation procédural parallèle constitue la base de la thèse. Il permet de donner une définition de l'équivalence de deux exécutions pour la construction d'un mécanisme de réexécution déterministe. Ce chapitre a fait l'objet d'une publication [23]. Elle est ici réécrite et augmentée de compléments au modèle de base. Les concepts de ce modèle sont très largement inspirés des concepts ATHAPASCAN.

L'appel de procédure à distance permet d'exécuter une fonction sur un *processeur virtuel* différent de celui qui traite le processus léger appelant. L'exécution de la fonction appelée est prise en charge par un processus léger créé spécifiquement pour ce traitement. Les fonctions publiées pour être appelées à partir d'autres processeurs virtuels sont encapsulées dans des *points d'entrée*. Ces points d'entrée couplent la réception d'une requête avec la création d'un processus et la terminaison du processus avec l'envoi du résultat.

Nous avons vu au chapitre 2 que la réexécution déterministe consistait toujours à tracer l'ordre des accès aux ressources partagées (mémoire ou réseau de communication) et à reproduire cet ordre. Dans le cas d'un modèle procédural parallèle, les points d'entrée peuvent être vus comme des ressources partagées. Cette intuition guide vers le traçage de l'ordre de traitement des requêtes arrivant sur les points d'entrée. À partir d'un modèle procédural de base (défini au paragraphe 3.1), qui ne comporte que des appels de procédures, il est possible de définir les conditions d'équivalence de deux exécutions. La démonstration de l'équivalence se base sur celle de Mellor-Crummey dans [62]. Au modèle de base il est nécessaire ou confortable d'ajouter quelques compléments (communication par messages et synchronisation entre processus) afin de faciliter la programmation. Ces extensions

(traitées au paragraphe 3.3) ne remettent pas en cause la démonstration de l'équivalence.

3.1 Modèle procédural de base

À l'inverse des modèles par échange de messages, le modèle procédural parallèle couple généralement l'activation et la terminaison de processus avec la communication et la synchronisation. La communication est bidirectionnelle et, par essence, asymétrique. Le processus appelant et le processus appelé ont des rôles distincts. L'appelant est client d'un service exécuté par l'appelé. C'est une généralisation des langages fonctionnels séquentiels où l'appel d'une fonction est effectué le plus tôt possible avant que son résultat ne soit utilisé.

Ce concept repose sur les implantations courantes de l'appel de procédure à distance (en anglais *Remote Procedure Call*) de Birrel et Nelson [7]. Le mécanisme d'appel de procédure à distance offre la possibilité à un processus léger d'exécuter une fonction se trouvant dans un processeur virtuel distant (voir figure 3.1). Dans le cadre de l'appel léger de procédure à distance [6] (en anglais *Lightweight Remote Procedure Call*), l'exécution de cette fonction est prise en charge par un processus léger créé spécifiquement pour celle-ci. Ce processus léger peut être démarré aussi bien localement qu'à distance. La durée de vie du processus léger est limitée à la durée d'exécution de la fonction. Ces appels peuvent être synchrones ou asynchrones, c'est-à-dire suspendre ou non l'exécution de l'appelant.

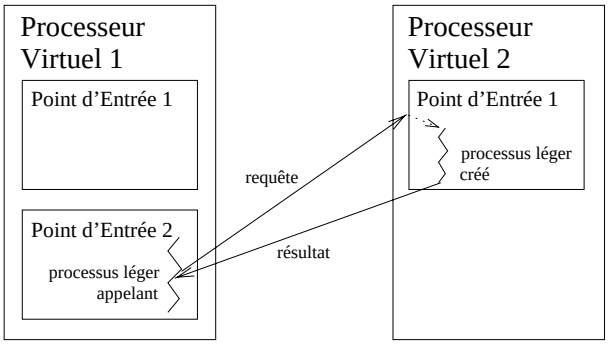


FIG. 3.1 – *Modèle procédural parallèle.*

La sémantique de l'appel synchrone (voir figure 3.2(a)) correspond à celui de l'appel de procédure classique. La seule différence est que l'exécution de

la procédure s'effectue sur un autre processeur virtuel. L'appel bloquant de procédure à distance n'augmente pas en soi le parallélisme. Le flot d'exécution se déplace sur un site distant. L'exécution du processus appelant ne reprend qu'après le retour du résultat. L'appel asynchrone correspond à la création d'activités parallèles. Cela permet de continuer, sans attendre, le calcul jusqu'à la partie où le résultat est nécessaire. Le processus créé à distance s'exécute en parallèle du processus appelant, qui peut alors effectuer d'autres appels. Un mécanisme de points de synchronisation permet d'attendre explicitement le résultat d'un appel non-bloquant. L'attente peut suspendre l'exécution du processus si le résultat n'est pas encore reçu (voir figure 3.2(c)). Dans l'autre cas, la prise en compte du résultat a lieu après sa réception (voir figure 3.2(d)). Il n'est pas obligatoire de respecter pour ces attentes un ordre lié à l'ordre des appels. Un cas particulier de l'appel asynchrone est l'appel sans attente de résultat (voir figure 3.2(b)). Ce type d'appel fait l'économie d'un point de synchronisation non nécessaire lorsque le principal effet attendu est la création de parallélisme.

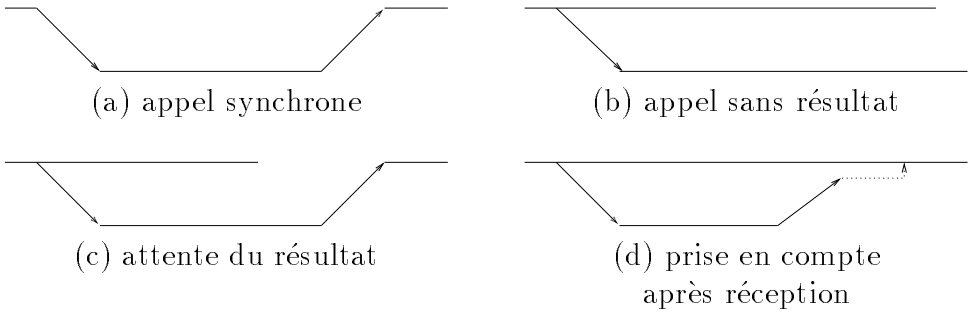


FIG. 3.2 – *Appels légers de procédure à distance.*

Dans le cas d'un modèle procédural de base, les effets de bord par affectation à des variables globales sont interdits. De même les seules communications entre deux processus sont les transferts de paramètres et de résultats. Sur un réseau qui respecte l'ordre des messages entre deux points, il est autorisé d'effectuer ces transferts à la volée plutôt qu'en un seul bloc. Ceci permet le démarrage de la procédure appelée avant la disponibilité de tous ses paramètres. De même les premiers résultats peuvent être retournés avant la fin du calcul de la procédure. La seule limite est que le dernier paramètre doit être reçu avant l'émission du premier résultat (voir figure 3.3).

Dans notre étude nous allons considérer un modèle où chaque procédure candidate à l'appel à distance est encapsulée dans un *point d'entrée.*

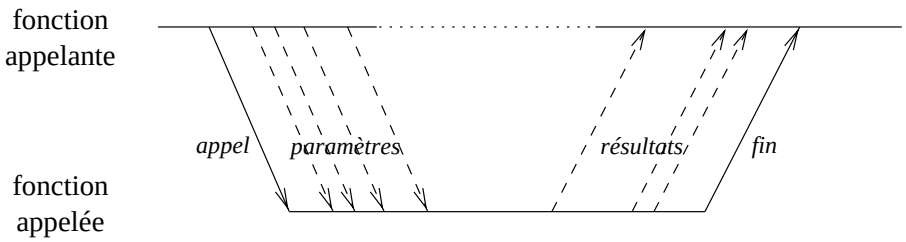


FIG. 3.3 – *Transmission des paramètres et des résultats d'une fonction.*

Chaque point d'entrée permet l'appel local ou distant de la procédure encapsulée. Les points d'entrée sont regroupés dans des *processeurs virtuels*. L'appel d'un point d'entrée sur un processeur virtuel donne lieu à la création sur ce processeur virtuel d'un *processus* (léger) qui calcule la fonction. Le *degré de concurrence* de chaque point d'entrée peut être borné de manière à limiter le nombre de processus actifs simultanément sur ce point d'entrée sur chaque processeur virtuel. Cela permet au programmeur de réaliser des mécanismes de synchronisation.

3.2 Équivalence d'exécutions

Un modèle d'exécution pose un cadre formel pour définir l'équivalence de deux exécutions du même programme. La notion d'équivalence est définie du point de vue du programmeur afin que le comportement observé soit identique pour deux exécutions équivalentes. La démonstration est similaire à la démonstration de l'équivalence donnée par Mellor-Crummey [62]. Un programme parallèle est composé d'un ensemble de processus (légers) qui exécutent chacun une fonction pour calculer la réponse (le résultat) à une requête. Une relation de placement P définit la bijection entre les requêtes émises et les processus de calcul. La démonstration montre qu'il suffit de forcer la même relation de placement P dans deux exécutions pour les rendre équivalentes.

3.2.1 Modèle d'exécution

Dans la suite nous considérerons que :

1. Le noyau de communication préserve l'ordre des messages. Pour les communications entre deux processeurs virtuels donnés, l'ordre de réception est identique à l'ordre d'émission.

2. Les requêtes sont traitées selon l'ordre de réception sur chaque point d'entrée.

3. L'exécution *enregistrée* et les exécutions *rejouées* d'un programme parallèle utilisent un ensemble fixé et ordonné de processeurs virtuels PV. Chaque exécution rejouée est lancée avec les mêmes paramètres initiaux que l'exécution enregistrée. Chaque processeur virtuel offre un ensemble fixé et ordonné de points d'entrée PE_{pv} qui permettent chacun l'appel d'une fonction.

4. Toutes les réexécutions disposent au moins des mêmes ressources, en espace mémoire et disque, que l'exécution enregistrée.

5. Les programmes n'utilisent pas de primitives non-déterministes du système. Le résultat d'équivalence pour de tels programmes peut être étendu aux programmes utilisants des primitives non-déterministes, pourvu que les résultats de ces primitives soient enregistrés au cours de la phase d'enregistrement et lus par les phases rejouées qui suivent (voir les détails de réalisation en 4.2.1).

6. L'utilisation des entrées/sorties par les programmes est limitée. Le programmeur doit s'assurer que les entrées des réexécutions sont les mêmes que celles de l'exécution enregistrée. Pour les périphériques de sortie partagés, qui manipulent des flots de données séquentiels, les accès doivent être encapsulés dans des points d'entrée à concurrence bornée à 1. Il n'y a ainsi pas de mélange des sorties de plusieurs processus légers concurrents.

7. Les temps de transmission des requêtes et des résultats sont finis.

Chaque ensemble PE_pv est un sous-ensemble de l'ensemble PE des points d'entrée de l'application. Par la suite, nous distinguerons le processeur virtuel sur lequel est défini chaque point d'entrée. Nous notons pv, pe le point d'entrée pe défini sur le processeur virtuel pv.

Définition 3.1 *Un* historique des réceptions de requêtes $hr_{pv,pe}$ *est associé à chaque point d'entrée de chaque processeur virtuel. Il définit l'ordre selon lequel les requêtes reçues sont traitées. La notation de l'historique des réceptions de requêtes du point d'entrée pe du processeur virtuel pv est la suivante :*

$$hr_{pv,pe} = c_0^{pv,pe}, c_1^{pv,pe}, c_2^{pv,pe}, \ldots$$

Le modèle d'exécution se réfère aux historiques[1] des réceptions de requêtes $hr_{pv,pe}$ qui sont définis comme la séquence des processus exécutés sur le point d'entrée. Chaque calcul $c_{pl}^{pv,pe}$ est réalisé par le processus identifié par le triplet unique $\langle pv, pe, pl\rangle$ désignant le processus pl exécutant le point d'entrée pe sur le processeur virtuel pv.

Définition 3.2 *Un* historique des émissions de requêtes he_p *est associé à chaque processus* $p = \langle pv, pe, pl\rangle$. *Il définit la séquence des requêtes émises par ce processus durant l'exécution. La notation de l'historique des émissions de requêtes du processus* p *est la suivante :*

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

Chaque émission de requête $e_i^p = \langle pv, pe\rangle$ est destinée au point d'entrée pe sur le processeur virtuel pv. Pour chaque processus p, l'historique des émissions de requêtes he_p reflète les interactions de ce processus avec le reste du programme.

Définition 3.3 *La* relation de placement P *est un ensemble de triplets de la forme* $\langle p_1, i, p_2\rangle$ *indiquant que l'émission de requête* $e_i^{p_1} = \langle pv, pe\rangle$ *est traitée par le processus* $p_2 = \langle pv, pe, pl\rangle$.

La relation P réalise une bijection entre l'ensemble des requêtes émises et l'ensemble des calculs (requêtes reçues). Cette bijection garantit que chaque requête émise est traitée et que chaque calcul correspond à une requête émise.

D'après ces définitions, l'exécution X est caractérisée par le triplet $\langle H, E, P\rangle$, où H est l'ensemble des historiques des réceptions de requêtes, E est l'ensemble des historiques des émissions de requêtes et P est la relation de placement.

3.2.2 Conditions d'équivalence d'exécutions

Définition 3.4 *Deux exécutions* X *et* X' *sont dites équivalentes si pour chaque processus* $p = \langle pv, pe, pl\rangle$ *les deux exécutions assignent le même historique des émissions de requêtes au processus* p $(E = E')$. *L'équivalence de deux exécutions* X *et* X' *est notée* $X \approx X'$.

1. La confusion entre *historique* et *histoire* est facile pour les anglophones. L'histoire d'un événement est l'ensemble des événements qui le précède causalement (voir 1.1). L'historique est une séquence ordonnée sur un même objet.

Cette définition de l'équivalence d'exécutions convient pour le déverminage d'un programme puisque le comportement de chaque processus est individuellement identique dans toutes les exécutions équivalentes. Ces comportements identiques permettent au programmeur de raffiner sa compréhension de l'exécution d'un programme par la répétition de réexécutions. Une technique de déverminage cyclique peut être appliquée.

Lemme 1 *Les processus séquentiels n'ayant aucune interaction extérieure sont déterministes.*

Ce lemme exprime l'hypothèse de base de tous les mécanismes de réexécution déterministe. Il s'applique aussi à la réexécution de programmes séquentiels.

Conséquences :

1. Quelle que soit l'exécution d'un programme parallèle, un processus lancé avec les mêmes conditions initiales émettra la même première requête ou le même résultat, s'il n'émet pas de requête.

2. Quelle que soit l'exécution d'un programme parallèle, un processus lancé avec les mêmes conditions initiales et dont les requêtes précédemment émises étaient identiques et ont retourné les mêmes résultats, émettra la même requête suivante ou résultat, s'il n'émet pas d'autre requête. Ici la différence est que le processus interagit avec son environnement. Toutefois ses interactions demeurent les mêmes quelles que soient les exécutions.

À partir de la définition 3.4 de l'équivalence et du lemme 1, nous allons montrer les conditions d'équivalence. Nous utiliserons la relation de causalité entre les événements de notre modèle. Une horloge vectorielle, adaptée des définitions de Fidge et Mattern [25, 60], est l'outil formel adéquat. Le marquage des messages par ce mécanisme permet d'ordonner les messages. Cet ordre sera utilisé pour la démonstration de l'équivalence de deux exécutions.

Définition 3.5 *Une* horloge vectorielle *hv est définie pour notre modèle comme un vecteur dont la dimension est le nombre de points d'entrée utilisés par une exécution d'un programme. Elle est mise à jour ainsi :*

1. *Le $pe^{ième}$ composant de l'horloge vectorielle d'un point d'entrée pe est incrémenté à chaque fois qu'une requête reçue $c_{pl}^{pv,pe}$ est traitée par le*

point d'entrée. C'est à dire que pour chaque nouveau processus pl créé, l'horloge est incrémentée :

$$hv_{pe}[pe] := hv_{pe}[pe] + 1$$

Le processus créé reçoit le numéro $pl = hv_{pe}[pe]$ *après l'incrémentation de la valeur de l'horloge du point d'entrée.*

2. *La valeur de l'horloge vectorielle* hv_{pe} *de pe est accolée à chaque message envoyé par un processus de pe, qu'il s'agisse d'un message de requête ou de résultat.*

3. *L'horloge vectorielle* hv_{pe} *de pe est mise à jour à la réception de chaque message par le point d'entrée : si le message est une requête, c'est au lancement d'un nouveau processus, juste avant d'incrémenter le* $pe^{ième}$ *composant de l'horloge vectorielle (voir ci-dessus) ; sinon, si le message est une réponse, l'incrémentation a lieu quand le message est passé au processus requérant. La mise à jour réalise l'opération suivante :*

$$hv_{pe} := \sup(hv_{pe}, hv_{mes})$$

sup *étant une opération de maximum composant par composant.*

Nous utilisons maintenant cette horloge vectorielle pour définir un ordre partiel entre les messages émis durant une exécution. Soit n le nombre de points d'entrée dans chaque exécution. Soient m_i et m_j deux messages émis par des processus exécutant les points d'entrée distincts PE_i et PE_j durant l'exécution d'un programme et hv_i et hv_j les valeurs des horloges vectorielles accolées à m_i et m_j.

Définition 3.6 *L'ordre partiel entre messages induit par l'horloge vectorielle sera noté par* $\prec_{HV}$*:*

$$m_i \prec_{HV} m_j \quad - \quad hv_i \prec hv_j$$

avec

$$\begin{aligned} hv_i \prec hv_j \quad - \quad & hv_i[k] \leq hv_j[k], \forall k \in [1,n] \\ & et \exists l, hv_i[l] < hv_j[l] \end{aligned}$$

$\prec_{HV}$ est un ordre partiel car des messages non liés causalement ne peuvent être ordonnés. Plusieurs requêtes émises par le même processus p peuvent porter la même horloge vectorielle puisque celle-ci n'est mise à jour que lors de la création de nouveaux processus. Toutefois elles sont ordonnées par l'indice i de l'émission de requête e_i^p pour étendre l'ordre $\prec_{HV}$. Si deux requêtes e_i^p et e_j^p du processus p ne peuvent être ordonnées par $\prec$, alors $e_i^p \prec_{HV} e_j^p$ si $i < j$.

Théorème 1 *Soit $X = \langle H, E, P \rangle$ une exécution d'un programme. Soit X' une exécution du même programme sous les hypothèses exposées ci-dessus (voir début de la partie 3.2.1).*

Pour que X' soit équivalente à X, toutes les requêtes de X' doivent être placées dans les historiques de réception en utilisant P ($P' = P$).

Pour démontrer ce théorème, nous allons faire une preuve par l'absurde. Nous supposerons qu'il existe dans une exécution un message qui n'a pas d'équivalent dans l'autre. Un tel message peut être soit le premier émis par un processus, soit une interaction ultérieure. S'il s'agit du premier message émis, il faut distinguer le cas particulier du premier message émis par le programme. Un message sans équivalent ne peut exister avec un placement identique. La démonstration du théorème 1 découle directement du lemme 1, comme l'illustre la preuve suivante.

Preuve : Supposons qu'il existe au moins un message de X' sans message identique correspondant dans X. S'il existe plusieurs de ces messages, il existe un ensemble de plus petits messages, selon la relation $\prec_{HV}$. Soit $r_j'^i$ un des messages de cet ensemble, le $j^{ème}$ message émis par le processus i. Deux cas peuvent se produire :

1. Soit $j = 1$, ce qui signifie que $r_j'^i$ est la première requête émise par le processus i, ou la réponse émise par le processus i à la fin de son calcul, s'il n'émet aucune requête. De nouveau deux cas sont possibles :

   (a) Soit $i = 1$, $r_1'^1$ est la première requête émise durant l'exécution du programme. Le programme a été lancé avec les mêmes paramètres d'entrée dans X et X' et donc les calculs séquentiels avant la première émission de requête doivent être identiques dans X et X' (conséquence du lemme 1). Ainsi les requêtes r_1^1 et $r_1'^1$ sont identiques.

(b) Soit $i > 1$, comme le processus i

- a été créé comme conséquence de la même requête dans les deux exécutions X et X', cette requête étant plus petite que r'^i_1 dans l'ordre $\prec_{HV}$,
- a commencé avec les mêmes conditions initiales à cause de l'utilisation durant l'exécution X' du placement P défini durant l'exécution X,
- n'a pas reçu d'autre entrée externe avant d'émettre r'^i_1,

il réalisera le même calcul séquentiel entre son initialisation et l'émission de r'^i_1 (conséquence du lemme 1). Ainsi r'^i_1 émise durant X' est identique à r^i_1 émise durant X.

2. Soit $j > 1$. Mais avant l'émission de r'^i_j, le processus i

- a commencé avec les mêmes conditions initiales à cause de l'utilisation durant l'exécution X' du placement P défini durant l'exécution X,
- a reçu les mêmes entrées, dans le même ordre que dans l'exécution X, car sinon il y aurait un message m'_i de X', tel que $m'_i \prec_{HV} r'^i_j$, sans message identique correspondant m_i dans X, ce qui contredit l'hypothèse ci-dessus.

À cause du lemme 1, il n'est pas possible pour r'^i_j d'être différente de la $j^{ème}$ requête du processus i dans X et l'hypothèse de la preuve est contradictoire.

Dans tous les cas, l'hypothèse de l'existence d'une requête de X' sans équivalente dans X est contredite. Un raisonnement similaire prouve qu'il est impossible pour une requête de X de ne pas avoir sa contrepartie dans X'. Ainsi l'utilisation de la même relation de placement P dans deux exécutions garantit l'équivalence de ces exécutions. □

3.2.3 Réduction des traces

Une contribution importante de cette étude est la limitation des types d'événements considérés pour définir l'équivalence de deux exécutions. Cette limitation découle de l'abstraction de plusieurs événements de bas niveau en un événement du niveau applicatif. La concentration s'attache uniquement

aux événements d'interaction entre les processus. Parmi ces interactions, seules les réceptions de requêtes sont tracées. Cette réduction s'oppose au traçage de tous les événements du modèle (émission et réception de requête et émission et réception de résultat).

La réduction obtenue ne fait appel à aucune des améliorations décrites au paragraphe 2.3. Le calcul de la relation $\xrightarrow{O}$ en cours d'exécution est compliqué et coûteux. En effet les horloges vectorielles peuvent devenir très volumineuses et impraticables à gérer efficacement. Nous utilisons ici les horloges vectorielles comme simple (et puissant) outil de démonstration. La relation $\xrightarrow{G}$ poserait pour notre modèle des problèmes lors de la réexécution. Il faudrait synchroniser tous les processeurs virtuels lors de la réception de chaque requête.

3.3 Compléments au modèle de base

Le modèle de base considère les appels de procédure comme les seules interactions entre les processus. D'autres modèles utilisent les communications et les synchronisations entre processus indépendemment de la création de parallélisme. Ces modèles peuvent être intégrés dans le modèle présenté ici. La notion de point d'entrée peut être appliquée aux primitives considérées dans les autres modèles.

3.3.1 Communication par messages

Les modèles de communication par échange de messages utilisent plusieurs représentations de la communication entre les processus. Des hypothèses générales sur le réseau décrivent les conditions de distribution des messages. Ces hypothèses concernent la fiabilité du réseau de communication et l'ordre de distribution des messages selon l'ordre d'émission. Une première approche s'abstrait totalement de la réalisation physique de la communication. Les seules entités manipulées sont les messages, qui voyagent directement de l'émetteur au récepteur. Une seconde approche décrit le réseau de communication comme un ensemble de liens (ou canaux) établis entre les processus. Ces liens peuvent être directs ou indirects. Les messages voyagent sur ces liens (ou dans ces canaux). Une troisième approche ne voit du réseau de communication que les interfaces avec les processus. Ceux-ci peuvent accueillir des messages dans des ports ou expédier des messages vers des ports lointains. La navigation des messages n'est pas décrite par le modèle, mais

simplement leur disponibilité.

De toutes ces approches, les ports sont les abstractions qui se rapprochent le plus des points d'entrée. Ils représentent une encapsulation de la fonction d'accès aux messages. Habituellement, les ports sont considérés comme des entités passives, manipulées par les processus. Un port peut aussi être envisagé comme une entité active à qui les processus adressent des requêtes. Selon cette approche, un port est un service de stockage des messages en transit. Les requêtes sont donc de deux types : dépôt ou retrait de message. Puisqu'un port est désormais un point d'entrée, il correspond parfaitement à notre modèle procédural. Ainsi nous enrichissons le modèle de base par la communication par échange de messages.

3.3.2 Synchronisation entre processus

Les mécanismes de synchronisation les plus courants sont les verrous, les sémaphores et les signaux. Les sémaphores et les verrous sont utilisés avec des primitives de parenthésage pour les accès aux ressources partagées. Les signaux agissent comme des messages brefs diffusés à un ensemble de processus. Cette dernière forme peut difficilement être intégrée à notre modèle. Les verrous et les sémaphores peuvent être considérés comme des manières particulières de réaliser un point d'entrée.

Les primitives de synchronisation par sémaphores et par verrous agissent comme des distributeurs d'autorisations d'accès. L'entrée dans la section critique ne se fait que si une autorisation a été donnée au processus. Durant cette partie de son exécution, un processus se trouve dans un contexte particulier.

Si l'autorisation a été donnée par un verrou, le processus est seul à exécuter des instructions qui peuvent modifier certaines zones partagées de la mémoire. Il se trouve donc dans la situation d'un processus exécutant la fonction d'un point d'entrée à concurrence bornée à 1. En poursuivant l'analogie, les instructions exécutées entre l'autorisation d'accès délivrée par le verrou et la libération forment le corps de la procédure appelée. Le verrou est acquis puis libéré par le même processus. Dans ces conditions, un parenthésage par un verrouillage et un déverrouillage équivaut à un appel de procédure sur un point d'entrée. Nous créons donc un point d'entrée virtuel pour la procédure d'acquisition de verrou.

L'analogie peut aussi être appliquée aux sémaphores en relâchant les contraintes d'unicité de processus concurrent et d'identité du processus re-

quérant et du processus libérant. Ces contraintes correspondent au fonctionnement des verrous. Les sémaphores ont un fonctionnement plus souple. La procédure P joue ici le rôle de parenthèse ouvrante pour la définition du point d'entrée virtuel. Ainsi nous enrichissons le modèle de base par la synchronisation entre processus.

3.4 Perspectives

Le modèle présenté dans ce chapitre s'applique aux appels de procédures à distance. Bien qu'il ait été étudié pour une réalisation basée sur les processus légers, il reste applicable dans le cadre des implantations classiques telles que celles de Birrel et Nelson [7]. Ce modèle s'applique également aux modèles *Clients-Serveur*, à la terminologie près. Les processus appelants sont les clients des processus créés pour traiter leurs requêtes. Chaque point d'entrée est un service mis à disposition des clients potentiels au sein de l'application.

Enfin les modèles d'objets actifs ou d'acteurs peuvent exploiter ce modèle procédural. Le processeur virtuel représente un objet actif qui propose ses points d'entrée comme méthodes. Les fils d'exécution sont les différentes activités en cours au sein de l'objet. Pour un acteur, les comportements sont similaires à des points d'entrée. Dans notre approche, il n'y a pas de traçage des événements au niveau des messages échangés et du fonctionnement interne du noyau exécutif comme dans le mécanisme de réexécution déterministe défini pour le langage BOX [72]. Dans notre modèle, plusieurs événements de base sont regroupés au sein d'un événement de plus haut niveau d'abstraction.

En limitant les types d'événements tracés, un mécanisme de réexécution déterministe plus efficace peut être réalisé. L'intrusion nécessaire à l'enregistrement des traces qui guideront la réexécution est ainsi aisément réduite. La mesure de cette intrusion (voir chapitre 5) montre qu'elle reste faible pour une utilisation de ce modèle dans la réalisation d'un mécanisme de réexécution déterministe. Le chapitre suivant présente l'application de ce modèle au noyau exécutif ATHAPASCAN-0a.

Chapitre 4

Réexécution pour Athapascan

Ce chapitre présente la mise en œuvre d'un mécanisme de réexécution déterministe pour le noyau exécutif ATHAPASCAN-0a [14]. Ce noyau exécutif est basé sur le modèle procédural parallèle défini dans le chapitre 3. Un prototype du noyau exécutif ATHAPASCAN-0a a été instrumenté avec le mécanisme décrit. Ce prototype instrumenté a été expérimentalement validé. Le déterminisme des réexécutions a été testé avec un programme très indéterministe, créant un grand nombre de processus légers. L'enregistrement des traces nécessaires à la réexécution déterministe introduit un surcoût en temps d'exécution. La mesure systématique de ce surcoût dû au traçage est présentée dans le chapitre 5.

4.1 Présentation d'Athapascan-0a

4.1.1 Modèle de programmation

Le modèle de programmation ATHAPASCAN-0a est un modèle procédural parallèle. Une application est composée de processeurs virtuels distribués sur une machine parallèle. Chaque processeur virtuel porte un ensemble de services (ou points d'entrée). L'appel d'un service, à destination d'un processeur virtuel déterminé, crée un processus léger sur ce processeur virtuel pour exécuter la procédure associée au point d'entrée. Sur le processeur virtuel porteur de ce point d'entrée, le nombre d'exécutions simultanées de la procédure associée est le degré de concurrence d'un service. Il est possible de limiter le degré de concurrence, par exemple pour réaliser un moniteur.

Le noyau exécutif ATHAPASCAN-0a [14] réalise les variétés synchrone et asynchrone des appels légers de procédure à distance. La primitive *Call* as-

sure l'invocation bloquante d'un service. Lorsque le processus appelant poursuit son exécution après l'appel à cette primitive, il a reçu le résultat. Pour continuer ses calculs avant de recevoir le résultat, un processus peut appeler la primitive *Spawn*. Cette primitive assure l'invocation non bloquante d'un service. Pour utiliser pleinement l'appel asynchrone, le processus appelant peut tester la présence d'un résultat attendu sans se bloquer en attente. Si le résultat n'est pas encore arrivé, il est possible de continuer d'autres calculs avant de tester de nouveau. La primitive *TestSpawn* assure cette fonctionnalité de test. Son résultat est soit positif si le résulat est arrivé, soit négatif si le résultat est encore attendu. Lorsqu'un processus n'a plus de calculs locaux à effectuer, il peut se mettre en attente bloquante du résultat d'un appel asynchrone antérieur. Cette fonctionnalité est assurée par la primitive *WaitSpawn*.

ATHAPASCAN-0a prévoit la possibilité de définir dynamiquement un point d'entrée et la fonction associée, qui sera exécutée lors du traitement de prochaines requêtes. Pour cela, le programmeur dispose de la primitive *NewEntryPoint* pour définir un point d'entrée et de la primitive *RemoveEntryPoint* pour retirer un point d'entrée du catalogue d'un processeur virtuel. Le changement de procédure associée à un point d'entrée s'effectue en deux étapes. Le point d'entrée est désactivé par *RemoveEntryPoint* puis la nouvelle définition est donnée avec *NewEntryPoint*. Deux points d'entrée sont obligatoirement définis sur chaque processeur virtuel. Le point d'entrée *InitTask* est appelé pour réaliser l'initialisation du processeur virtuel. Le point d'entrée *TermTask* assure une terminaison propre du processeur virtuel.

La limitation du degré de concurrence sur un point d'entrée permet contrôler le nombre de fils d'exécution simultanément créés sur ce point d'entrée. La limite est déclarée lors de la définition du point d'entrée. Pour réaliser un mécanisme de verrou, il suffit de définir un point d'entrée avec une limite fixée à 1. Pour un tel point d'entrée, le noyau exécutif garantit qu'à tout instant au plus un processus léger est créé pour exécuter la procédure associée. Les autres mécanismes de synchronisation doivent être réalisés par le programmeur sur ce principe minimaliste. En particulier, les sémaphores ne sont pas définis dans la spécification initiale d'ATHAPASCAN-0a.

4.1.2 Implantation

Un prototype du noyau exécutif ATHAPASCAN-0a a été implanté sur un réseau de stations de travail ou d'ordinateurs personnels et sur un IBM SP-1.

Le noyau exécutif se présente sous forme d'une bibliothèque de fonctions, liée aux codes exécutables d'une application ATHAPASCAN-0a. La bibliothèque de communication PVM [76] et plusieurs bibliothèques de processus légers servent de base à la construction de ce noyau exécutif. Selon l'architecture cible, la bibliothèque de processus légers peut être au choix une bibliothèque au standard DCE sur AIX et OSF/1, LWP [22], REX [66] ou une bibliothèque locale (conçue et développée par Briat de l'équipe APACHE). Les processeurs virtuels ATHAPASCAN-0a sont des tâches PVM qui peuvent être dynamiquement ajoutées. Les appels de procédure à distance et le transfert des résultats s'effectuent par des primitives PVM de passage de message. Les bibliothèques de processus légers sont utilisées pour créer et manipuler les processus ATHAPASCAN-0a.

Un processus léger initial, nommé *chien de garde*, est créé au lancement de chaque processeur virtuel. Ce processus gère les relations avec la bibliothèque de communication. Il traite chaque message arrivant qui peut être soit un appel de procédure à distance soit une réponse. Dans le cas d'un appel de procédure, il crée et initialise un nouveau processus léger. L'ordonnancement des processus est non préemptif. Les processus sont suspendus uniquement lorsqu'ils attendent une réponse (*Call* ou *WaitSpawn*). Lorsqu'une réponse arrive, le chien de garde réveille le processus en attente. Cette politique d'ordonnancement

Initialement, les spécifications prévoyaient que tous les mécanismes de synchronisation devaient être réalisés à partir de la limitation de concurrence. Les accès aux zones de mémoire commune pouvaient être protégés par des points d'entrée dont la limite de concurrence était fixée à 1. Toutefois cette forme de partage de mémoire est très contraignante pour la programmation. Elle est d'autre part très inefficace car elle nécessite la création, l'activation et la destruction d'un processus léger uniquement pour réaliser une synchronisation entre processus. L'implantation de la limitation de concurrence utilise des sémaphores pour sérialiser les créations de processus légers. Il est plus simple de mettre ceux-ci à la disposition des programmeurs par des primitives de l'interface de programmation. Lorsqu'il s'agit de fils d'exécution sur un même processeur virtuel, des primitives réalisent entre les processus différentes formes de synchronisation plus rapides.

4.2 Instrumentation du noyau exécutif

Pour le modèle d'exécution présenté au chapitre précédent, une exécution équivalente peut être construite en assurant que des propriétés pertinentes d'ordonnancement des traitements des requêtes de l'exécution originale sont préservées. Le paragraphe 3.2.2 montre qu'il suffit d'utiliser le même placement des requêtes dans les historiques de traitement pour obtenir une exécution équivalente. Si l'ordre dans lequel les requêtes sont traitées peut être enregistré durant une exécution et si le même ordre peut être imposé durant des exécutions suivantes, alors ces exécutions sont équivalentes. Les seules interactions tracées sont les appels de point d'entrée. La stratégie résultante évite l'enregistrement de chaque réception de message. Avec cette stratégie, il suffit d'un seul enregistrement pour chaque traitement de requête, alors que l'utilisation d'un modèle de plus bas niveau nécessiterait de deux à six enregistrements par requête.

4.2.1 Cas particuliers

Limitation de concurrence La limitation de la concurrence sur chaque point d'entrée ne change pas la définition de l'équivalence. Cette limitation ne nécessite pas de traitement particulier. Dans le cas particulier d'un point d'entrée défini en exclusion mutuelle, un espace mémoire peut lui être associé. Cet espace mémoire servira à conserver un état entre deux appels successifs. Cette introduction des effets de bords est toujours supportée par le modèle présenté. En effet on peut considérer que l'état mémorisé est un paramètre (resp. résultat) implicite de la procédure. Il n'y a aucune protection des accès à la mémoire dans le cas d'une limitation strictement supérieure à un.

Primitives de synchronisation En considérant que les primitives de synchronisation réalisent une sérialisation des appels comme le fait un point d'entrée à concurrence bornée, nous nous rapprochons du cas présenté ci-dessus. Souvent les primitives fonctionnent par paire. L'une réalise la réservation de ressource et l'autre effectue la libération. La primitive de réservation correspond à l'appel d'un point d'entrée, la primitive de libération au retour du résultat et le code entre les deux réalise la fonction de traitement de l'espace mémoire partagé. Ces fonctions peuvent être utilisées pour synchroniser les processus indépendamment d'un traitement sur la mémoire partagée.

Les primitives de réservation de ressource sont des points d'entrée virtuels

pour le modèle d'équivalence. Elles auront sur chaque processeur virtuel une bande associée à chacune pour les traiter comme des points d'entrée. Ces points d'entrée sont définis localement par le système mais ils restent inaccessibles pour les processus extérieurs au processeur virtuel. Leur particularité est de ne pas créer de processus léger pour s'exécuter. Ces points d'entrée virtuels s'exécutent dans le contexte du processus appelant. La particularité de leur réalisation n'induit pas de changement sur la manière de les considérer pour le modèle. La bande de trace associée sert pour enregistrer l'ordre de traitement des appels (assimilés à des requêtes) à ces procédures.

Primitives indéterministes Le traitement des fonctions indéterministes nécessite un recours à la réexécution dirigée par les données. La valeur du résultat d'une fonction indéterministe ne peut être recalculée. Elle doit donc être enregistrée pour être reproduite lors d'une réexécution. Une telle fonction peut être considérée comme un point d'entrée particulier qui associe la valeur du résultat à chaque calcul de son historique des réceptions de requêtes. Durant l'enregistrement, le résultat est conservé avec l'identité de la requête à laquelle il a été donné. Pour la réexécution, le résultat enregistré est délivré au lieu d'exécuter réellement le calcul.

Pour les fonctions indéterministes dont le domaine de résultat ne comporte que deux valeurs, il est possible de n'enregistrer que les requêtes qui ont reçu une des deux valeurs (par exemple la moins fréquente). Toute requête qui ne figure pas dans l'historique recevra l'autre valeur (par exemple la plus fréquente). Comme la valeur à retourner se déduit de la présence ou de l'absence de la requête dans l'historique, cette valeur n'a pas besoin d'être enregistrée. La valeur qui provoque l'enregistrement peut être imposée par la nécessité de modifier le comportement de la procédure lors de la réexécution. Ainsi pour le test de complétion d'un appel asynchrone (primitive *TestSpawn*), une réponse positive (le résultat est arrivé) nécessite d'attendre la réalisation effective de la condition lors de la réexécution. Le comportement de test est remplacé par le comportement d'attente.

4.2.2 Modification du noyau pour l'instrumentation

Le prototype initial d'ATHAPASCAN-0a ne prévoit pas un nommage reproductible des requêtes émises. Celles-ci portent des identifiants liés au noyau de communication et au mécanisme interne du noyau exécutif. L'utilisation de PVM comme noyau de communication favorise le changement d'identité

selon le nombre de *tâches* exécutées auparavant sur la machine virtuelle. Au niveau interne, l'identité d'une requête est donnée par un numéro de point de synchronisation disponible sur le processeur virtuel émetteur. Ce numéro dépend de l'état de la liste des points de synchronisation disponibles. La liste évolue au cours de l'exécution selon les requêtes émises et les résultats attendus.

Pour le mécanisme de réexécution déterministe, il faut que l'identité de chaque requête puisse être reproduite. Le nommage doit donc être déterministe. Une manière simple consiste à le baser sur l'ordre interne à l'application. Nous avons introduit dans le noyau exécutif les compteurs nécessaires à la gestion du nommage déterministe. Une identité de requête est générée, lors de son émission, sous la forme $\langle pv, pe, pl, r\rangle$ où pv est le numéro du processeur virtuel, pe celui du point d'entrée sur le processeur virtuel, pl celui du processus léger sur le point d'entrée et r celui de la requête sur le processus léger. Le compteur de requêtes d'un processus léger est incrémenté simplement lors de chaque émission d'une requête par le fil d'exécution. Par exemple les requêtes de la forme $\langle 1, 100, 0, r\rangle$ sont émises par le premier fil d'exécution créé par l'application (sur le processeur virtuel 1 et le point d'entrée 100).

D'autre part nous avons réalisé quelques aménagements nécessaires pour ajouter les modes traçage et réexécution au noyau exécutif ATHAPASCAN-0a. Il faut tout d'abord pouvoir indiquer au noyau exécutif quel mode a été choisi pour l'application. Par défaut, il s'agit du mode non-instrumenté. L'indication d'un autre mode s'effectue par l'intermédiaire d'options sur la ligne de commande. Outre le mode particulier retenu (enregistrement ou réexécution), il faut spécifier le répertoire où se trouveront les fichiers de traces.

Pour éviter de trop fréquentes entrées/sorties avec les fichiers de traces, celles-ci sont stockées dans des tampons du noyau exécutif. L'initialisation du mode traçage génère les tampons d'enregistrement et ouvre les fichiers associés. L'initialisation du mode réexécution génère les structures des bandes associées à chaque point d'entrée et les tampons de chargement. Dans les deux modes, la terminaison s'effectue après une fermeture des fichiers ouverts.

L'utilité du mécanisme d'enregistrement n'est évidente que si les traces peuvent être enregistrées même lors d'une erreur grave provoquant un plantage de l'application enregistrée. Pour garantir l'enregistrement des traces en toutes circonstances, les interruptions du système UNIX sont interceptées. Ceci permet d'enregistrer les traces jusqu'à l'instant où s'est produite

l'erreur[1]. Le traitement des interruptions consiste à écrire les tampons d'enregistrement dans les fichiers associés. L'arrêt de l'application est propagé du processeur virtuel défaillant vers tous ceux qu'il connaît jusqu'à l'arrêt de tous les processeurs virtuels.

Les paragraphes 4.2.3 et 4.2.4 décrivent les instrumentations pour le mode enregistrement et pour le mode réexécution. Nous avons implanté ces instrumentations dans le noyau exécutif ATHAPASCAN-0a. Des choix effectués pour l'un des mode dépendent souvent des choix pour l'autre mode. Le tout forme le mécanisme de réexécution déterministe réalisé pour ATHAPASCAN-0a.

4.2.3 Instrumentation pour l'enregistrement

L'enregistrement concerne tous les points d'entrée définis par l'application et les points d'entrée virtuels du noyau exécutif. Une bande est associée à chaque point d'entrée. Elle est utilisée par le chien de garde pour enregistrer les identifications des requêtes arrivantes. L'identification de chaque requête est composée de quatre éléments : le numéro du processeur virtuel, le numéro du point d'entrée sur le processeur virtuel, le numéro du processus léger sur le point d'entrée et le numéro de la requête dans le processus léger. Toutes les bandes d'un processeur virtuel sont stockées dans un fichier. L'enregistrement des bandes s'effectue de manière incrémentale lorsque les tampons des bandes sont pleins.

Pour les points d'entrée définis par le programmeur, l'ordre de traitement des requêtes est enregistré sur la bande du point d'entrée. La primitive *TestSpawn* (point d'entrée virtuel du noyau exécutif) est traitée de manière particulière. L'ordre de traitement des requêtes correspond à la relation de placement P du paragraphe 3.2.1. Il s'agit de l'ordre de réception des requêtes, puisque celles-ci sont traitées selon leur ordre d'arrivée sur chaque point d'entrée.

Sur la figure 4.1, les flèches au dessus des bandes indiquent la position du prochain enregistrement. Chaque case contient l'identité de la requête qui a été traitée par le point d'entrée. La possibilité de redéfinir les points d'entrée en cours d'exécution implique de pouvoir distinguer sur la bande d'un point d'entrée, par quelle définition a été traitée une requête enregistrée. Pour cela un marqueur est introduit sur la bande de traces lors de l'appel à la primitive *NewEntryPoint*. Les marqueurs sont représentés sur la figure par une case

1. L'objectif principal de la réexécution est de reproduire une exécution erronée jusqu'à l'erreur.

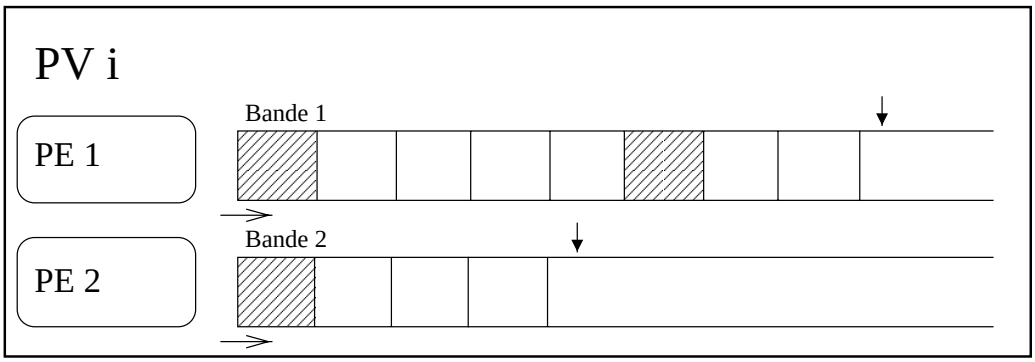


FIG. 4.1 – *Enregistrement de l'ordre de traitement des requêtes.*

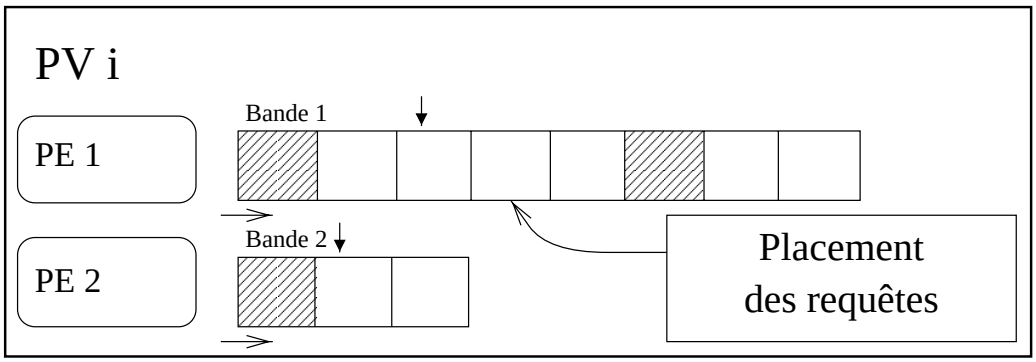
hachurée. Le point d'entrée 1 a été redéfini en cours d'exécution, après avoir traité quatre requêtes.

Pour un appel asynchrone, le temps d'attente du résultat dépend de la rapidité du processeur virtuel qui a reçu l'appel. De la charge du système dépend donc le nombre de tests négatifs (par la primitive *TestSpawn*) avant la réception du résultat. Cette primitive indéterministe est traitée selon les indications données au paragraphe 4.2.1. Seuls les numéros d'ordre des tests positifs sont enregistrés.

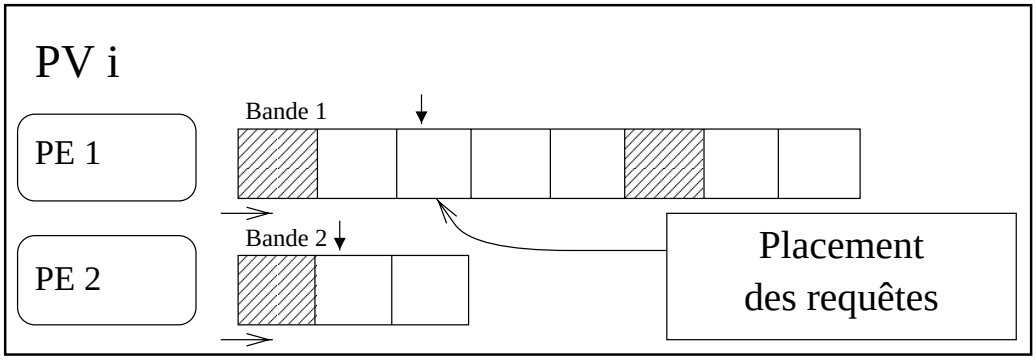
4.2.4 Instrumentation pour la réexécution

Durant les réexécutions, le chien de garde de chaque processeur virtuel administre les bandes de tous les points d'entrée de son processeur virtuel. Il reproduit l'ordre de traitement enregistré grâce à un mécanisme de retardement de prise en compte des requêtes. Les requêtes d'appel de procédure à distance arrivant sur un point d'entrée sont mises dans une file d'attente selon l'ordre donné par la bande. Un nouveau processus est créé seulement lorsqu'il y a une requête en attente associée à l'élément de la tête courante de la bande (indiquée par la flèche au dessus de celle-ci sur la figure 4.2).

Sur la figure 4.2(a), une requête est attendue en tête de bande (sous l'indicateur) pour le point d'entrée 1. Une requête arrive en avance par rapport à l'ordre enregistré. Elle est placée en attente. Dans la figure 4.2(b), la requête attendue en tête arrive. Elle est placée sur la bande puis aussitôt traitée, après l'avancement de l'indicateur de tête de bande. Lors d'un prochain basculement de fil d'exécution, la requête arrivée en avance pourra être traitée, puisque l'indicateur se trouve alors sur sa case.



(a) réception anticipée d'une requête mise en attente



(b) réception d'une requête en tête de bande

FIG. 4.2 – *Réordonnancement des requêtes lors d'une réexécution.*

Pour la réexécution de la primitive *TestSpawn*, une bande spéciale est utilisée car il n'est pas nécessaire de respecter l'ordre entre les requêtes de différents processus mais seulement l'ordre pour chacun d'eux séparément. Chaque processus doit être forcé à émettre le même nombre de tests infructueux que durant l'exécution enregistrée. Pour chaque test au cours de la réexécution, le mécanisme vérifie sur la bande la présence d'un enregistrement portant le même numéro d'ordre. S'il existe un tel enregistrement, le comportement de test est transformé en comportement d'attente car au cours de l'exécution enregistrée, le résultat était arrivé lors de ce test. S'il n'existe pas d'enregistrement de numéro correspondant à ce test, le mécanisme répond par la négative, sans même tester la présence du résultat.

Lors de la réexécution, le changement de définition d'un point d'entrée ne pourra être effectué que lorsque toutes les requêtes en attente pour l'instance courante auront été traitées. Le processus appelant la primitive *NewEntry-*

Point est suspendu jusqu'à la réalisation du changement de définition.

4.3 Exemple

La preuve du chapitre 3 montre qu'il suffit de reproduire l'ordre de traitement des requêtes pour produire une réexécution déterministe équivalente à l'exécution tracée. Toutefois cette preuve ne nous place pas à l'abri d'une erreur de conception lors de la mise en œuvre des traces et de la réexécution. Pour valider le mécanisme implanté, le déterminisme des réexécutions doit être testé de manière expérimentale. L'application utilisée pour ce test doit permettre de produire un grand nombre de processus dont l'ordre sera très variable d'une exécution à l'autre. Une telle application indéterministe doit aussi produire un résultat facilement comparable.

Dans le domaine des programmes parallèles de test, il existe un classique qui consiste à trouver tous les placements de n reines sur un échiquier $n \times n$, tels que les reines ne puissent mutuellement se prendre. Pour nous, l'intérêt du problème des reines est double. D'une part, il peut être résolu par une application indéterministe qui produit de nombreux processus. D'autre part, il permet de comparer aisément la liste de résultats produite par chaque exécution. Cette liste contient toujours les mêmes résultats mais différemment ordonnés. Pour donner une idée de l'indéterminisme de cet algorithme, considérons le problème pour 8 reines. Les 92 solutions différentes peuvent être ordonnées dans la liste de résultats selon $92! = 1.24 \times 10^{148}$ ordres possibles (plus de 150 ont effectivement été observés).

4.3.1 Algorithme indéterministe

Une des manières de concevoir un algorithme indéterministe consiste à adopter une architecture de type «ferme de processus». Les processeurs virtuels sont divisés en deux groupes distincts. L'un est chargé de la réalisation des calculs élémentaires et l'autre est chargé de la coordination. Pour simplifier la coordination, nous avons retenu une architecture avec un seul processeur virtuel dans le groupe de coordination (appelé communément «fermier») et plusieurs processeurs virtuels dans le groupe de réalisation (appelés «travailleurs»). Si le fermier est une ressource passive, qui attend les requêtes, les conflits entre travailleurs vont créer l'indéterminisme des exécutions. Cette architecture a été retenue car elle permet une répartition de la charge selon la rapidité de chaque travailleur, sans effort de la part du fer-

mier. Elle s'oppose à une architecture de même nature où le fermier est une ressource active et les travailleurs sont passifs. Lorsqu'un fermier distribue le travail de manière autoritaire, l'indéterminisme n'est pas créé par les conflits d'accès aux ressources qu'il contrôle.

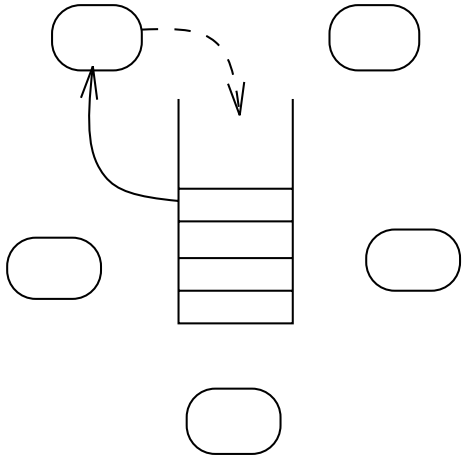


FIG. 4.3 – *Structure de l'algorithme en ferme de processus.*

Le fermier maintient une pile des placements partiels courants, qui représentent chacun une configuration à compléter. Il maintient aussi la liste des résultats qui contiendra toutes les solutions trouvées. Chaque travailleur prend de la pile un placement partiel et génère alors de nouveaux placements en ajoutant une reine dans chaque position autorisée de la colonne choisie. Les nouveaux placements sont de trois types. Dans la première catégorie tombent les impasses où aucune reine ne peut être placée et qui sont simplement effacées. Dans la seconde, se trouvent les solutions, qui sont envoyées au fermier pour publication dans la liste de résultats. Enfin la troisième catégorie contient les placements partiels à compléter et qui sont placés sur la pile. Le programme se termine lorsque la pile est vide et qu'aucun travailleur ne travaille sur un placement à compléter.

4.3.2 Structure de l'application

Le processeur virtuel du fermier publie deux points d'entrée. Un point d'entrée (SERV_PILE_CONFIG) est chargé de gérer la pile des placements partiels. Ce point d'entrée reçoit les requêtes pour prendre un placement et celles pour en déposer un. L'autre point d'entrée (SERV_AFFICHAGE) gère la liste des résultats, qui est publiée dans un fichier. Comme ils agissent sur des

structures de données persistantes, ces points d'entrée sont définis avec une limite de concurrence fixée à 1 pour éviter les incohérences dues à des accès concurrents non protégés.

Les processeurs virtuels des travailleurs publient deux autres points d'entrée. Un point d'entrée (SERV_PLACE_REINE) est chargé d'étudier la situation d'une reine à une position donnée sur un échiquier donné. Selon la situation de la reine, le placement est oublié ou bien remis au fermier pour publication d'un résultat ou pour remise dans la pile. Un autre point d'entrée (SERV_DERIVATION) acquiert un placement partiel auprès du fermier puis distribue l'étude de la validité du placement d'une reine sur une position fixée de l'échiquier. La distribution des positions s'effectue par des appels au point d'entrée SERV_PLACE_REINE sur son processeur virtuel.

L'initialisation de l'application commence par créer les différents processeurs virtuels, qui instancient chacun les points d'entrée publiés. Ensuite la configuration de l'échiquier vide est donnée au fermier pour stockage dans la pile (SERV_PILE_CONFIG). Puis les travailleurs sont démarrés (SERV_DERIVATION). La terminaison est détectée par le point d'entrée (SERV_PILE_CONFIG) lorsqu'il a reçu de chaque travailleur une requête pour prendre un placement alors que la pile est vide. Il retourne alors le signal de fin du travail à toute requête pour prendre un placement qui lui parvient après cette détection.

4.3.3 Analyse d'une trace d'exécution

Prenons comme exercice de réexécution déterministe le calcul des deux solutions au problème des quatre reines. Pour introduire de l'indéterminisme dans les exécutions, il suffit d'exploiter deux travailleurs. Ceux-ci émettent des requêtes concurrentes à destination du fermier qui les sérialise. L'ordre de réception des requêtes varie d'une exécution à l'autre. La figure 4.4 présente les bandes de traces pour une exécution avec deux travailleurs. Chaque réception d'une requête identifiée par $\langle pv, pe, pl, r\rangle$ est notée par un vecteur de la forme : $\begin{smallmatrix} pv \\ pe \\ pl \\ r \end{smallmatrix}$.

L'initialisation de l'application est assurée par le processeur virtuel 1. Le code du fermier est exécuté par le processeur virtuel 2. Les processeurs virtuels 3 et 4 exécutent le code du travailleur. Les points d'entrée obligatoires sont numérotés 3 et 4 pour *InitTask* et *TermTask* (voir paragraphe 4.1.1). Les points d'entrée SERV_PILE_CONFIG, SERV_AFFICHAGE, SERV_PLACE_REINE

et SERV_DERIVATION reçoivent respectivement les numéros 101, 102, 103 et 104. Les enregistrements hachurés sont les marqueurs de définition du point d'entrée par la primitive *NewEntryPoint*.

Après l'initialisation des processeurs virtuels (définition des points d'entrée par défaut par les requêtes $\langle pv, 0, 0, r \rangle$) et la définition des points d'entrée de l'application (requêtes $\langle pv, 3, 0, r \rangle$) sur chacun d'eux, l'échiquier vide est déposé dans la pile des configurations (requête $\langle 1, 100, 0, 4 \rangle$) par le processeur virtuel de lancement. Les dérivations sont ensuite lancées (sur les processeurs virtuels 3 et 4) et la concurrence commence entre les deux travailleurs.

PV2

PE3	pv	2	1																								
InitTask	pe	0	100																								
	pl	0	0																								
	r	1	1																								
PE4	pv	2	1																								
TermTask	pe	0	100																								
	pl	0	0																								
	r	2	7																								
PE101	pv	2	1	3	4	4	4	3	3	4	3	3	4	3	4	3	4	3	4	3	4	3	4	3	4		
SERV_PILE_CONFIG	pe	3	100	104	104	104	104	103	103	104	103	103	103	104	104	103	103	104	104	103	104	104	103	103	104	104	10
	pl	0	0	0	0	0	0	0	1	0	2	3	1	0	0	4	2	0	0	5	0	0	4	7	0	0	0
	r	1	4	1	1	2	3	1	1	4	1	1	1	6	7	1	1	8	9	1	11	11	1	1	13	13	1
PE102	pv	2	3	4																							
SERV_AFFICHAGE	pe	3	103	103																							
	pl	0	8	6																							
	r	2	1	1																							

Bandes de traces du fermier

PV3

PE3	pv	3	1								
InitTask	pe	0	100								
	pl	0	0								
	r	1	2								
PE4	pv	3	1								
TermTask	pe	0	100								
	pl	0	0								
	r	2	8								
PE103	pv	3	3	3	3	3	3	3	3	3	3
SERV_PLACE_REINE	pe	3	104	104	104	104	104	104	104	104	104
	pl	0	0	0	0	0	0	0	0	0	0
	r	1	2	3	4	5	7	9	10	12	14
PE104	pv	3	1								
SERV_DERIVATION	pe	3	100								
	pl	0	0								
	r	2	5								

PV4

PE3	pv	4	1								
InitTask	pe	0	100								
	pl	0	0								
	r	1	3								
PE4	pv	4	1								
TermTask	pe	0	100								
	pl	0	0								
	r	2	9								
PE103	pv	4	4	4	4	4	4	4	4		
SERV_PLACE_REINE	pe	3	104	104	104	104	104	104	104		
	pl	0	0	0	0	0	0	0	0		
	r	1	5	6	8	10	12	14	16		
PE104	pv	4	1								
SERV_DERIVATION	pe	3	100								
	pl	0	0								
	r	2	6								

Bandes de traces des travailleurs

FIG. 4.4 – *Exemple de traces avec deux travailleurs.*

Le premier travailleur acquiert du fermier (processeur virtuel 2, point d'entrée SERV_PILE_CONFIG) l'unique configuration disponible par la requête $\langle 3, 104, 0, 1 \rangle$. Le second travailleur tente ensuite infructueusement par trois

fois d'acquérir une configuration alors que la pile est vide. Pendant ce temps, le premier travailleur a distribué l'échiquier vide entre quatre placeurs de reine (requêtes 2 à 5 du processus de dérivation). Le second travailleur reçoit enfin une configuration (en résultat de sa requête $\langle 4, 104, 0, 4\rangle$) lorsque certaines des configurations calculées par le premier travailleur sont remises sur la pile (par les requêtes $\langle 3, 103, 0, 1\rangle$ et $\langle 3, 103, 1, 1\rangle$).

Chacun des deux travailleurs trouve une solution qu'il remet au fermier pour affichage (requêtes $\langle 3, 103, 8, 1\rangle$ et $\langle 4, 103, 6, 1\rangle$). La terminaison est détectée par le fermier lors du traitement de la requête $\langle 4, 104, 0, 17\rangle$ car la requête $\langle 3, 104, 0, 15\rangle$ n'avait pas reçu de placement à traiter. Le second travailleur reçoit immmédiatement le signal de fin. Le premier travailleur reçoit le signal lors de sa requête suivante.

Durant une réexécution déterministe, chacune de ces requêtes est traitée selon l'ordre enregistré dans les bandes. Cela signifie en particulier que le second travailleur se verra toujours refuser trois fois une configuration avant d'en recevoir une. Si au cours d'une réexécution une configuration calculée par le premier travailleur arrivait avant une des trois premières requêtes du second travailleur, son traitement serait reporté.

4.3.4 Observations

La liste de résultats est toujours l'ensemble complet des solutions mais l'ordre de publication dans la liste reflète la concurrence d'accès à la pile des placements partiels tenue par le fermier. Bien que la charge globale de travail soit *toujours* de trouver toutes les solutions au problème, la manière dont le travail est réparti entre les travailleurs influence l'ordre des solutions dans la liste de résultats. Par comparaison des listes de résultats, il peut être remarqué au premier coup d'œil si deux exécutions données ont produit leurs résultats dans le même ordre. Par l'observation de la liste de résultats de nombreuses exécutions, nous vérifions aisément que l'implantation de l'algorithme est vraiment indéterministe. Toutefois en utilisant le mécanisme de réexécution d'ATHAPASCAN, toute réexécution reproduit les mêmes solutions dans le même ordre que lors de l'exécution initiale enregistrée.

Le surcoût en temps dû à l'enregistrement de traces ne peut être mesuré avec un tel algorithme. Une application indéterministe ne convient pas pour effectuer des mesures systématiques. Au cours des expériences avec cette application d'essai, des exécutions tracées sont plus rapides que des exécutions non tracées. Cette observation peut surprendre au premier abord. En étu-

diant les traces des exécutions, il apparaît une forte variation du nombre de requêtes reçues par le fermier alors que la pile des configurations est vide. L'intrusion du traçage ralentit les travailleurs dont les requêtes insatisfaites sont alors moins nombreuses car elles sont moins fréquentes. Selon l'intrusion due au traçage et les autres perturbations de l'environnement, le nombre de requêtes infructueuses émises change le comportement de l'application. Une méthode d'analyse statistique d'un échantillon d'exécutions ne peut être appliquée sur une application dont le comportement est indéterministe. Il est attendu pour une telle méthode que le comportement de tous les éléments de l'échantillon soit le même (voir l'ouvrage de Saporta [73]). Pour une application indéterministe, les comportements des exécutions dépendent de conditions expérimentales qu'il est impossible de contrôler. Les expériences réalisées dans le chapitre suivant utilisent donc des programmes parallèles déterministes.

Dans toutes les expériences décrites ci-dessus et dans le chapitre suivant, le volume des traces enregistrées est resté raisonnablement limité. Ce volume dépend du nombre de requêtes ATHAPASCAN traitées durant l'exécution du programme, chaque requête produisant un enregistrement de 16 octets [2]. Pour le calcul des 724 solutions au problèmes des 10 reines, l'application de test a produit un peu moins de 1,2 Mo de traces en 6 minutes pour 292008 requêtes. Cela porte la fréquence des requêtes à plus de 80 requêtes traitées par seconde et par processeur virtuel.

Toutefois, les conditions expérimentales des mesures de surcoût n'étaient pas adaptées à la mesure du volume des traces enregistrées. Jusqu'à présent, ce volume s'incrit dans un intervalle de 30 à 300 octets par seconde et par processeur virtuel pour l'ensemble des programmes mesurés. Les programmes synthétiques mesurés dans le chapitre suivant produisent un maximum de 12 requêtes par seconde et par processeur virtuel. Cela permet de garantir un fonctionnement du mécanisme pour les conditions déjà observées.

Selon ces résultats, il est raisonnable d'espérer que les traces enregistrées peuvent être stockées en mémoire principale pour la majorité des programmes parallèles, ce qui maintiendra le surcoût aussi faible que dans les expériences décrites. Il peut aussi être attendu que seuls des programmes «pathologiques», qui n'ont pas encore été rencontrés, satureront l'espace mémoire et disque disponible.

2. Aucune optimisation sur la taille des compteurs n'est réalisée.

4.4 Conclusion

La réalisation du mécanisme de réexécution déterministe dans le prototype ATHAPASCAN-0a découle de l'étude théorique du chapitre 3. L'exemple d'une application très indéterministe montre comment il fonctionne. L'algorithme retenu pour calculer toutes les solutions au problème de reines est inefficace car il génère beaucoup de requêtes pour peu de calcul à effectuer par chacune. Son intérêt dans le cadre de ce chapitre réside dans sa capacité à générer des exécutions indéterministes. Pour mesurer le surcoût en temps introduit par le traçage, un tel algorithme ne convient pas. Le chapitre suivant présente la méthodologie de mesure systématique de ce surcoût.

Chapitre 5

Mesures du surcoût en temps

Tous les auteurs qui ont implanté un mécanisme de réexécution déterministe se sont intéressés à l'évaluation du surcoût en temps induit par le mécanisme d'enregistrement des traces. Cet intérêt est soutenu par le souhait de laisser le traçage comme mode normal d'exécution afin de traquer même les erreurs les plus furtives. Pour procéder à cette évaluation, ils utilisent un nombre limité (moins d'une dizaine) de programmes déjà écrits pour leur environnement de programmation.

Dans l'équipe APACHE existe un outil, nommé *ANDES* [45], qui permet de produire des programmes et d'en étudier le comportement. Cet outil génère des programmes à partir de modèles d'algorithmes paramétrables. Un même modèle peut produire plusieurs programmes aux comportements différents selon la nature et les valeurs des paramètres. Cette méthode permet de disposer plus facilement d'un banc d'essai correspondant à des critères à étudier. La méthode employée et les mesures obtenues ont fait l'objet d'une publication [24]. Ce chapitre reprend et complète cette publication.

L'objectif de nos mesures était d'évaluer systématiquement le surcoût de l'enregistrement des traces nécessaires pour réexécuter des programmes Athapascan de manière déterministe. La phase d'enregistrement des traces doit permettre d'enregistrer un comportement causalement aussi «proche» que possible des comportements non tracés. Une faible perturbation du comportement permet de conserver l'enregistrement comme mode normal d'exécution. Toute erreur furtive peut alors être capturée. La phase de réexécution est moins critique du point de vue de la causalité mais elle doit tout de même limiter le surcoût observé par un programmeur lors d'une session de mise au point interactive.

Pour ces raisons, seule la phase d'enregistrement a fait l'objet de mesures

systématiques. La phase de réexécution a été validée par quelques mesures complémentaires. Le résultat attendu de ces mesures était une confirmation des résultats préliminaires indiquant que le surcoût dû à l'enregistrement des traces reste suffisamment faible pour considérer le mode enregistrement comme mode normal d'exécution pour ATHAPASCAN. Un autre résultat potentiel de ces mesures était l'identification d'éventuels programmes «pathologiques». Les mesures montrent le faible surcoût induit par l'enregistrement des traces. Aucun programme pathologique n'a été détecté.

5.1 Méthode de mesure

L'évaluation du surcoût dû à la collecte de traces, a été réalisé en utilisant la méthode et les outils d'évaluation de performance développés pour le projet APACHE [78]. L'objectif de cette méthodologie est d'être capable de prédire les performances d'algorithmes parallèles exécutés sur une classe de machines parallèles à mémoire distribuée. La méthodologie de mesure comporte plusieurs étapes : la modélisation quantitative d'algorithmes parallèles, la génération de programmes synthétiques à partir des modèles et les mesures de performance de l'exécution des programmes synthétiques sur des machines parallèles (voir figure 5.1). Ces trois phases sont détaillées dans la suite.

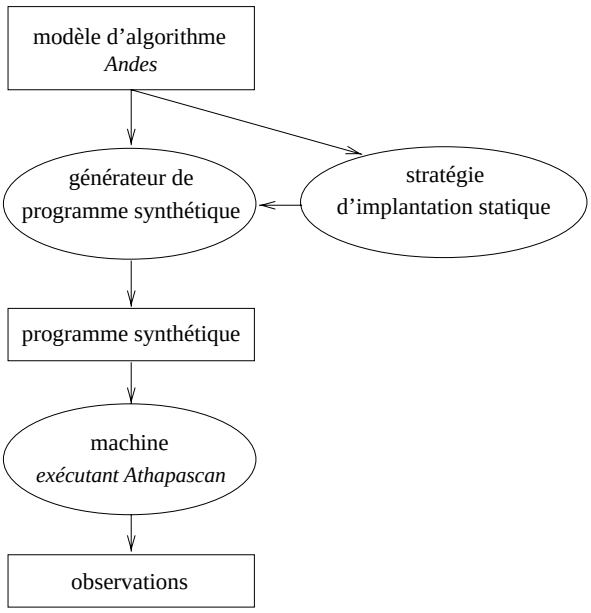


FIG. 5.1 – *Chaîne d'évaluation avec l'outil ANDES.*

5.2 Modélisation d'algorithmes

Le langage *ANDES* de modélisation d'algorithmes parallèles [45] est conçu pour le parallélisme de contrôle dont il modélise les structures de contrôle, la précédence, les coûts de calcul, les coûts liés aux données, les mouvements globaux de données, le raffinement hiérarchique du modèle et les échanges de messages. Il peut être appliqué à divers modèles de programmation qui utilisent ces notions. Chaque modèle l'utilise par la définition d'opérateurs dédiés qui marquent les nœuds d'un graphe. Le générateur de programmes synthétiques était d'abord conçu pour un modèle d'exécution proche de *CSP*. Dans le cadre de ces travaux il a été adapté pour le noyau exécutif ATHAPASCAN. Les opérateurs pour ATHAPASCAN sont présentés dans le paragraphe 5.2.2.

5.2.1 Principes généraux

Avec *ANDES*, un programme est modélisé par un graphe orienté (appelé *DG-ANDES*) dont les sommets modélisent des *nœuds de calcul* et les arcs modélisent la *précédence* (voir figure 5.2). Chaque nœud de calcul est composé de trois sortes de *logiques* : une logique *d'entrée*, une logique de *calcul* et une logique de *sortie*. Les logiques d'entrée et de sortie servent à décrire les relations de dépendance entre nœuds de calcul, essentiellement les précédences de tâches et les communications. La logique de calcul modélise une partie des calculs effectués par l'application.

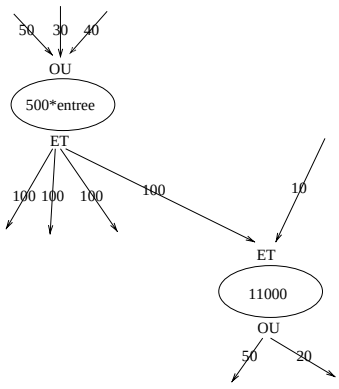


FIG. 5.2 – *Logiques d'entrée et de sortie ANDES.*

Les logiques d'entrée et de sortie peuvent être de plusieurs types. Les types les plus courants sont les logiques booléennes et les logiques associées aux

opérations globales. Par exemple, si un nœud de calcul possède une logique d'entrée *OU*, l'exécution de la logique de calcul du nœud commencera dès que la première communication en entrée sur le nœud aura lieu ; au contraire le calcul des nœuds ayant une logique d'entrée *ET* ne commencera qu'après l'exécution de toutes les communications en entrée. Une logique d'opération globale peut être utilisée pour modéliser des communications globales comme les diffusions. Une logique de sortie modélisant une diffusion est équivalente à une logique de sortie *ET* où la même donnée est associée à toutes les communications.

ANDES étant utilisé pour l'évaluation de performance quantitative, des fonctions numériques peuvent être associées aux logiques pour modéliser les besoins en ressources de la machine. Des coûts de communication sont associés aux logiques d'entrée et de sortie alors que des coûts de calcul sont associés aux logiques de calcul. Ces coûts peuvent être des constantes, des fonctions aléatoires ou des fonctions d'autres caractéristiques du graphe. Les fonctions aléatoires sont utilisées pour modéliser les coûts qui ne peuvent être prédits. Les *coûts de dépendance* sont utilisés pour modéliser des calculs dont les coûts dépendent de la taille des données en entrée. *ANDES* supporte également le développement de modèles de programme réguliers et hiérarchiques

Une logique de calcul peut modéliser du code séquentiel ou un autre calcul parallèle exprimé par un *DG-ANDES*. Les sommets peuvent modéliser un autre *DG-ANDES* permettant ainsi des représentations hiérarchiques de programmes : cette caractéristique peut être utile lors de l'analyse de la granularité du calcul et de son effet sur la performance. Plus le *DG-ANDES* est détaillé, plus le modèle est à «grain fin». Quand une logique de calcul modélise un code séquentiel, un coût est associé à cette logique. Ce coût modélise le nombre d'opérations de base du code, l'opération de base étant définie hors du modèle (opération en virgule flottante, opération entière ou autre). Les logiques d'entrée et de sortie représentent les dépendances de données entre les nœuds de calcul ainsi que les relations de précédence. Des coûts sont aussi associés aux logiques d'entrée et de sortie, représentant les tailles des données.

Il est possible de dériver plusieurs instances d'un algorithme parallèle à partir d'un modèle *ANDES* de cet algorithme en changeant la granularité des activités parallèles (tâches) ou la manière dont le travail est divisé. Un modèle *ANDES* d'algorithme représente ainsi un ensemble ou une classe d'algorithmes. Les modèles peuvent être classés selon la structure des *DG-*

ANDES qu'ils décrivent. Un modèle régulier construit une structure où les facteurs de branchement des sommets ou les sous-graphes sont identiques. Par opposition, un modèle irrégulier ne présente pas ces caractéristiques.

5.2.2 Adaptation de *ANDES*

La définition des logiques d'entrée et de sortie dépend du modèle de calcul pour lequel les programmes synthétiques doivent être produits. *ANDES* a été créé pour modéliser des programmes parallèles en C sur une machine MégaNode. Le modèle exploité était du type processus communiquants. De nouvelles logiques d'entrée et de sortie ont été définies pour modéliser les programmes ATHAPASCAN (voir figure 5.3).

L'extension d'*ANDES* au modèle ATHAPASCAN vise essentiellement à être appliquée pour la mesure du surcoût en temps d'exécution induit par l'enregistrement de traces par le mécanisme de réexécution déterministe présenté au chapitre précédent. Pour ces mesures, les programmes indéterministes ne conviennent pas (voir la remarque au paragraphe 4.3.4). L'étude de l'extension d'*ANDES* pour toutes les fonctionnalités d'ATHAPASCAN est un sujet à part entière. Il n'entre pas dans la problématique de la réexécution déterministe pour des applications exploitant des processus légers.

Seuls les opérateurs *Spawn* et *WaitSpawn* sont nécessaires pour modéliser des programmes ATHAPASCAN déterministes avec *ANDES*. Dans ce cas, l'opérateur *Call* est équivalent à un *Spawn* suivi immédiatement d'un *WaitSpawn*. Contrairement aux opérateurs d'ATHAPASCAN-0a, qui n'autorisent qu'une seule émission de requête ou qu'une seule attente de résultat, les logiques d'entrée et de sortie ont été spécifiées de manière pluraliste. Ceci permet de simplifier l'expression des modèles d'algorithmes. La gestion des boucles d'émission de requêtes ou de réception de résultats est assurée par le générateur de programmes synthétiques.

Logiques d'entrée:

REQ: entrée unique représentant une requête ATHAPASCAN.

WAIT: une entrée indiquant la continuation du fil d'exécution et une ou plusieurs entrées représentant les résultats attendus des sommets précédents.

COMP: entrée unique représentant la continuation du fil d'exécution.

Logiques de sortie:

REP: sortie unique représentant le résultat.

SPAWN: une sortie indiquant la continuation du fil d'exécution et une ou plusieurs sorties représentant les requêtes à émettre.

CONT: sortie unique indiquant la continuation du fil d'exécution.

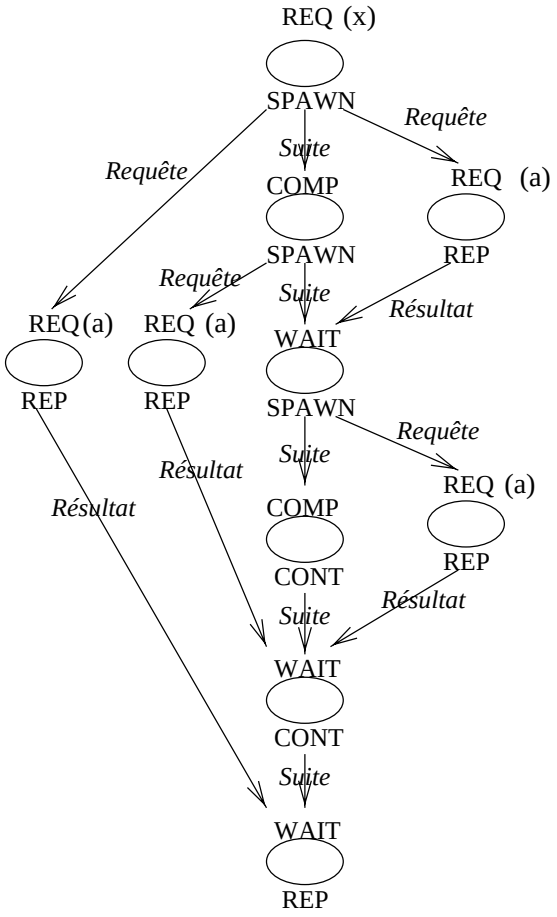


FIG. 5.3 – *Toutes les connexions licites pour* ATHAPASCAN.

Certaines compositions de sommets et d'arcs ne sont pas admises car elles n'ont pas de sémantique dans le modèle de programmation ATHAPASCAN et peuvent empêcher la génération de programmes synthétiques corrects.

Parmi les neufs combinaisons possibles de sommets, deux sont interdites. La combinaison REQ/CONT ne peut être générée car l'opérateur CONT suppose qu'il y a eu un SPAWN auparavant sur le fil d'exécution. La logique

d'entrée REQ marquant le début du fil, il n'y a pas de possibilité d'associer un SPAWN. Pour une raison symétrique, la combinaison COMP/REP ne peut être générée. L'opérateur COMP suppose qu'il y aura un WAIT ultérieurement sur le fil. La logique de sortie REP marquant la fin du fil, il n'y a pas de possibilité d'associer un WAIT.

Parmi les douze combinaisons possibles d'arcs, sept sont illicites. Une logique de sortie REP ne peut être connectée qu'à une logique d'entrée WAIT. Seul l'opérateur WAIT traite les résultats en entrée. Symétriquement, une logique d'entrée REQ ne peut être connectée qu'avec une logique de sortie SPAWN. Seul l'opérateur SPAWN génère des requêtes en sortie. La sortie *Suite* d'une logique SPAWN ne peut être connectée à une logique d'entrée REQ. Symétriquement, la sortie *Requête* d'une logique SPAWN ne peut être connectée à une logique d'entrée WAIT ou COMP. Enfin la liaison d'une logique de sortie CONT à une logique d'entrée COMP n'a pas de sens. La modélisation est simplifiée en fusionnant les logiques de calcul des deux nœuds.

5.3 Génération de programmes synthétiques

Les mesures de performances de programmes parallèles ont été retenues comme technique d'évaluation de performances des modèles *ANDES* d'algorithmes parallèles. Les programmes générés à partir des modèles *ANDES* sont en fait des programmes parallèles *synthétiques*. ces programmes ne sont pas écrits directement par un programmeur mais ils sont générés automatiquement à partir des descriptions écrites avec le langage de modélisation *ANDES*. Ils émulent les comportements décrits dans les modèles en utilisant les opérations du modèle de programmation, qui sert de support d'exécution.

Un programme parallèle synthétique est un programme réel dont la consommation de ressources, processeur, mémoire et comunication, peut aisément être contrôlée et modifiée. Même s'ils ne produisent pas un résultat exploitable, tel qu'une prévision météorologique par exemple, les programmes synthétiques ont la même consommation de ressources que les programmes modélisés. Ils sont exécutés sur un multiprocesseur réel pour obtenir des mesures de performance d'exécution aussi proches que possible des résultats des implantations réelles de l'algorithme correspondant.

5.3.1 Fonctionnement du générateur

À partir d'un modèle d'algorithme écrit avec *ANDES*, il est possible de

générer un large éventail de programmes synthétiques avec différentes structures et différents coûts. Par exemple, pour un algorithme *Diviser pour Paralléliser*, il est possible de changer le facteur de branchement des nœuds ou la hauteur de l'arbre. Une fois que la structure du flux de contrôle est fixée, il reste possible de faire varier les paramètres influant sur les coûts de calcul et de communication. Selon le modèle, les paramètres liés à la structure et aux coûts peuvent être rendus interdépendants, par exemple en calculant le facteur de branchement en fonction du volume de données reçu.

Le générateur de programmes synthétiques pour ATHAPASCAN-0a transforme le graphe *ANDES* en un ensemble de points d'entrée et des fichiers de données. Les points d'entrée sont par commodité regroupés dans un seul modèle de processeur virtuel. C'est donc le même code qui sera disponible sur tous les processeurs virtuels du programme synthétique. La structure d'un point d'entrée est donnée par un ensemble de nœuds du graphe qui sont liés par un arc de type *Suite*. Le générateur transforme une telle chaîne en un texte représentant le code d'un point d'entrée. Le premier nœud a une logique d'entrée de type *REQ(nom)*, où *nom* représente le nom qui sera généré pour le point d'entrée. Le dernier nœud de la chaîne a une logique de sortie de type *REP*. Le générateur vérifie que chaque chaîne portant le même nom a la même structure. En cas d'erreur la génération échoue et rapporte l'identité du nœud qui a servi de référence à la définition du point d'entrée et celle de celui qui est en conflit.

Selon les logiques d'entrée et de sortie rencontrées au long de la chaîne, le générateur produit le code ATHAPASCAN-0a du point d'entrée. Les plus simples sont les logiques *COMP* et *CONT*, qui ne font rien. Elles servent uniquement à poursuivre le calcul, sans émettre de requête ni recevoir de résultat. La logique de sortie *SPAWN* sert à émettre une ou plusieurs requêtes. Comme la primitive ATHAPASCAN-0a *Spawn()* n'autorise l'émission que d'une seule requête, le générateur produit une boucle qui traitera successivement toutes les émissions du nœud. Il en va de même pour les réceptions de résultats définies par la logique d'entrée *WAIT* implantée par une boucle de réceptions par la primitive *WaitSpawn()*. Les logiques de calcul, qui modélisent du code séquentiel, sont implantées par une boucle vide. Cette boucle ne calcule aucun résultat mais consomme des cycles du processeur pour faire parcourir à l'indice de contrôle un intervalle de valeurs. Cette technique n'est valide que pour les programmes qui ne sont pas optimisés à la compilation. L'optimisation supprime les instructions qui servent à rien. C'est le cas d'une telle boucle qui peut être remplacée par l'assignation à l'indice de contrôle de

la borne supérieure de l'intervalle. L'optimiseur peut même supprimer cette affectation puisque la valeur ne sera pas utilisée.

Toutefois, le code généré ne représente que le squelette du programme. Pour lui donner vie, il faut rendre compte de tous les choix définis par le modèle d'algorithme. Comme chaque fil d'exécution peut avoir des valeurs distinctes pour les volumes de communication ou pour les durées de calcul, il n'est pas possible de générer un point d'entrée pour chaque fil d'exécution. Ceci conduirait à un code énorme dont chaque point d'entrée ne serait exécuté qu'une seule fois. Au lieu de cela, chaque point d'entrée doit pouvoir s'exécuter plusieurs fois avec des valeurs différentes. Au cours de l'exécution, le squelette émule le comportement décrit par le graphe *ANDES*. Il utilise pour cela les valeurs contenues dans un fichier de données, chargées lors de l'initialisation du processeur virtuel. Ces valeurs décrivent le comportement quantitatif alors que le squelette ne décrit que l'aspect qualitatif.

Pour créer une requête de la taille voulue ou pour boucler une logique de calcul, les valeurs fixées dans le graphe sont directement utilisées. En ATHAPASCAN-0a, une émission de requête par la primitive *Spawn()* désigne toujours explicitement le processeur virtuel qui traitera la requête. Cette primitive initialise un point de synchronisation sur lequel la primitive *WaitSpawn()* se mettra en attente du résultat. La gestion des indices des points de synchronisation est assurée par le générateur de programmes synthétiques. Quant au placement des appels sur les processeurs virtuels, il est généré par des stratégies extérieures spécialisées [8]. Ces stratégies ont été développées dans le cadre du projet pour des utilisations plus générales tout en conservant un lien étroit avec l'outil *ANDES*.

5.3.2 Exemple simple

Pour illustrer la génération d'un programme synthétique, suivons les étapes pour un algorithme de type *Diviser pour Paralléliser*. Considérons un modèle régulier où le travail initial est caractérisé par une quantité (*VOLUME*) d'informations à traiter. À chaque étape du calcul, le travail est toujours divisé entre le même nombre (*LARGEUR*) d'appels de procédure. Cette division intervient jusqu'à une certaine profondeur (*PROFONDEUR*) par rapport à l'appel initial. Chaque procédure doit prendre connaissance d'une quantité d'informations dépendant du volume qui lui est communiqué. En fin d'exécution de la procédure, la quantité retournée dépend du volume reçu. La répartition, le regroupement et le calcul non divisé ont des durées relatives

au volume à traiter.

```
Algorithme
    Appeler Diviser(PROFONDEUR, VOLUME)
Fin algorithme

Diviser(profondeur, volume)
    Recevoir(volume)
    Répartir(volume)
    Si profondeur == 1
        Répéter LARGEUR
            Appeler Traiter(volume)
        Fin répéter
    Sinon
        Répéter LARGEUR
            Appeler Diviser(profondeur - 1, volume / LARGEUR)
        Fin répéter
    Fin si
    Regrouper(volume)
    Retourner(volume)
Fin diviser

Traiter(volume)
    Recevoir(volume)
    Calculer(volume)
    Retourner(volume)
Fin traiter
```

Fixons maintenant les paramètres *LARGEUR* et *PROFONDEUR* pour dessiner le graphe d'une instance d'algorithme obtenue par ce modèle. La figure 5.4 présente le graphe *DG-ANDES* pour le cas où *LARGEUR* vaut 3 et *PROFONDEUR* vaut 1.

De ce graphe, le générateur de programmes synthétiques produit les points d'entrée ATHAPASCAN-0a. Nous reproduisons le code obtenu pour le point d'entrée *Traiter*. Ce point d'entrée s'exécutera en utilisant les données produites à partir de la valeur donnée pour le paramètre *VOLUME* et des règles définies pour produire les coûts de calcul et les volumes des requêtes et des

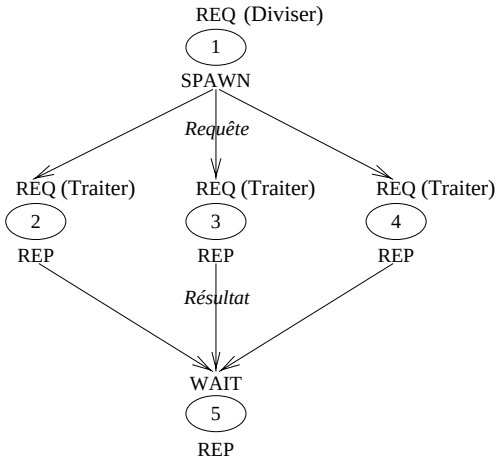


FIG. 5.4 – *Graphe ANDES pour l'exemple.*

résultats.

```
/* point d'entrée "Traiter" généré à partir du noeud 2 */
long    **tab_Traiter;  /* données d'exécution */
#define ACTION_Traiter  tab_Traiter[indreq][indact++]
/* définition du modèle de point d'entrée "Traiter" */
EP_MODEL_MACRO( Traiter, in, out)
        long            i, j;
        long            indreq, indact=0;
        BufDesc         buffreq;
        SynchroPt       joins[1];

        /* numéro de la requête à traiter */
        indreq = GetNickName();

        j = ACTION_Traiter;
        for( i = 0; i < j; i++) {
                /* boucle vide */
        }       /* fin de la répétition */

        /* emballer le résultat */
        preparer_bloc( out, ACTION_Traiter);
END_EP  /* Traiter */
```

5.4 Expériences

Le surcoût dû à l'enregistrement de traces est évalué par la mesure des temps d'exécution de programmes synthétiques générés à partir de modèles *ANDES* d'algorithmes. Les programmes synthétiques étant des programmes «réels», ils peuvent être utilisés comme n'importe quel «banc d'essai réel» pour mesurer le surcoût dû à l'enregistrement de traces. Les modèles retenus comportent des modèles d'algorithmes issus de bancs d'essai réels ou construits à partir d'algorithmes connus de la littérature.

5.4.1 Composition du banc d'essai

Comme cela a été présenté au paragraphe 4.3 pour l'application de test, certaines exécutions avec enregistrement de traces s'exécutent plus rapidement que des exécutions non tracées. Pour cette raison nous avons paradoxalement dû nous restreindre aux modèles de programme ayant un comportement déterministe. Alors que notre mécanisme est destiné à être utilisé principalement avec des applications indéterministes, la mesure valide du surcoût induit par le traçage ne peut s'opérer qu'avec un banc d'essai entièrement déterministe. Le comportement d'un programme indéterministe peut en effet être si différent pour chacune de ses exécutions que les comparaisons deviennent impossibles. Pour des programmes dont le comportement est indéterministe, il n'est pas possible d'appliquer une méthode de mesure statistique basée sur l'hypothèse que les exécutions observées ont des comportements similaires (voir l'ouvrage de Jain [40] sur les mesures de performances). Ceci nous oblige donc à mesurer des programmes déterministes en faisant l'hypothèse que les résultats obtenus sont généralisables à une classe importante de programmes parallèles.

L'ensemble des algorithmes retenus est composé des structures d'algorithme suivantes : *Diviser pour Paralléliser* (arbre équilibré), *Arbre de Recherche à la Prolog* (arbre déséquilibré), *Itération Régulière* (nombre de divisions identique à chaque étape), *Maître-Esclaves* (nombre de divisions variables d'une étape à l'autre) et le *Produit de Matrices* (selon l'algorithme numérique récursif de Strassen [75]).

Le choix d'une combinaison de coûts de calcul et de communication pour un modèle détermine un algorithme spécifique. Pour chaque modèle d'algorithme, six rapports calcul/communication ont été retenus. Ils s'échelonnent entre un usage très inefficace du système (10 fois plus de temps de commu-

nication que de temps de calcul) et un usage plus favorable (30 fois plus de temps de calcul que de temps de communication). Les valeurs intermédiaires de l'intervalle ont été espacées de manière logarithmique pour obtenir des valeurs de rapports proches de 0.1, 0.3, 1, 3, 10, 30. Les rapports calcul/communication des programmes synthétiques ont été ajustés par le réglage des valeurs des paramètres de coûts (calcul et communication) des modèles alors que les paramètres de structure, tels que le facteur de branchement par exemple, demeuraient inchangés.

Les expériences ont été réalisées en générant un programme synthétique pour chaque modèle d'algorithme et chacun des rapports retenus. Les fils d'exécutions définis dans les modèles ont été placés sur les processeurs virtuels exécutant les programmes synthétiques par un des algorithmes gloutons de la boîte à outils de placement [8] du projet APACHE. À partir de ces placements et des modèles *ANDES* d'algorithmes, des programmes ATHAPASCAN synthétiques ont été générés.

5.4.2 Méthode expérimentale

Pour mesurer le surcoût en temps de l'enregistrement, les temps d'exécution des programmes synthétiques ont été mesurés «avec» et «sans» activation du mode traçage. Pour chaque banc d'essai, un nombre suffisant de mesures a été effectué pour garantir, avec une probabilité supérieure à 95 %, que la moyenne réelle des temps d'exécution soit comprise dans un intervalle de confiance inférieur à 3 % de sa valeur et centré autour de la valeur estimée de la moyenne. Chaque surcoût ($\overline{surcoût}$) est le rapport de la différence du temps moyen d'exécution «avec» activation du mode traçage ($\bar{t}_a$)[1] et du temps moyen d'exécution «sans» activation du mode traçage ($\bar{t}_s$), divisée par le temps moyen d'exécution «sans» activation du mode traçage :

$$\overline{surcoût} = \frac{\bar{t}_a - \bar{t}_s}{\bar{t}_s} = \frac{\bar{t}_a}{\bar{t}_s} - 1$$

Si nous supposons le surcoût assez faible, ce qui est confirmé par les résultats expérimentaux (voir figures 5.5 et 5.6), l'intervalle de confiance peut être approché par une valeur fixe de 6 % (voir démonstration ci-dessous). À cause du rapport entre deux résultats connus avec une certitude de 95 % chacun, la certitude sur la précision des surcoûts est limitée à 90 %.

1. Cette notation ne doit pas être confondue avec le coefficient de Student, utilisé en statistiques.

Les mesures des temps d'exécution moyens $\bar{t}_a$ et $\bar{t}_s$ sont effectuées avec une précision fixée arbitrairement à 3 %, c'est-à-dire : $t_a \in [\bar{t}_a - c \times \bar{t}_a\ ,\ \bar{t}_a + c \times \bar{t}_a]$ et $t_s \in [\bar{t}_s - c \times \bar{t}_s\ ,\ \bar{t}_s + c \times \bar{t}_s]$ avec $c = 0,015$.

Ainsi :

$$surcoût \in \left[\frac{\bar{t}_a - c \times \bar{t}_a - (\bar{t}_s + c \times \bar{t}_s)}{\bar{t}_s + c \times \bar{t}_s}\ ,\ \frac{\bar{t}_a + c \times \bar{t}_a - (\bar{t}_s - c \times \bar{t}_s)}{\bar{t}_s - c \times \bar{t}_s} \right]$$

La borne inférieure de l'intervalle peut s'écrire :

$$\frac{\bar{t}_a}{\bar{t}_s} \times \frac{1-c}{1+c} - 1$$

comme c est petit, $\frac{1-c}{1+c}$ peut être approché par : $(1-c) \times (1-c)$ et donc $1 - 2c$ si nous négligeons le terme c^2. Ainsi la borne inférieure peut être approchée par :

$$\frac{\bar{t}_a}{\bar{t}_s} - 1 - 2c \times \frac{\bar{t}_a}{\bar{t}_s}$$

Si le surcoût est faible, $\frac{\bar{t}_a}{\bar{t}_s}$ est proche de 1, et la borne inférieure de l'intervalle peut être minorée par sa valeur approchée :

$$\frac{\bar{t}_a}{\bar{t}_s} - 1 - 2c$$

De même nous pouvons approcher la borne supérieure de l'intervalle. D'où :

$$surcoût \in \left[\frac{\bar{t}_a}{\bar{t}_s} - 1 - 2c\ ,\ \frac{\bar{t}_a}{\bar{t}_s} - 1 + 2c \right]$$

L'intervalle de confiance peut ainsi être approché par $4c$ qui vaut 0.06, c'est-à-dire une valeur fixée à 6 %, indépendante de la valeur mesurée du surcoût. Bien sûr cette valeur est une borne supérieure qui est une bonne approximation seulement si le surcoût mesuré reste faible. Ce que vérifient les mesures.

Nous pouvons aussi calculer la probabilité pour que le surcoût $\overline{surcoût}$ appartienne à l'intervalle de confiance. À partir de :

$$\overline{surcoût} = \frac{\bar{t}_a}{\bar{t}_s} - 1$$

nous savons que pour que le surcoût mesuré tombe *en dehors* de l'intervalle de confiance, il est nécessaire que l'une des moyennes mesurées $\bar{t}_a$ ou $\bar{t}_s$ n'appartienne pas à son intervalle de confiance (3 %). Ces deux mesures étant

indépendantes, la probabilité pour que le surcoût mesuré tombe *en dehors* de l'intervalle de confiance est la somme des probabilités pour que chacune des mesures tombe en dehors de son intervalle de confiance. Comme suffisament de mesures avaient été réalisées pour obtenir une certitude de 95 % pour les temps d'exécution mesurés $\bar{t}_a$ et $\bar{t}_s$, chacune de ces probabilités est de 5 % et leur somme est 10 %. Le surcoût $\overline{surcoût}$ appartient à l'intervalle de confiance avec une probabilité de 90 %.

5.4.3 Conditions expérimentales

Les programmes synthétiques ont été exécutés par le prototype du noyau exécutif ATHAPASCAN-0 sur une machine IBM SP1. Pour limiter les perturbations provenant de l'exécution de processus Unix non prévus par les expériences, le système était réservé pour les mesures sans autre utilisateur autorisé à exécuter une application simultanément. Les résultats obtenus lorsque ces conditions expérimentales n'étaient pas respectées, par l'exécution simultanée d'une autre application parallèle sur le SP1 par exemple, n'ont pu être utilisés à cause de leur comportement instable. Toutefois, il s'est avéré difficile d'éliminer toutes les perturbations, certaines provenant par exemple de l'exécution simultanée de processus système ou du traitement de démons PVM «laissés» sur la machine par d'autres utilisateurs. Ainsi certaines mesures collectées dans un environnement «modérément» perturbé ont été conservées quand l'instabilité statistique demeurait suffisamment faible pour atteindre la précision souhaitée.

Le nombre de mesures n nécessaires pour obtenir une précision donnée (3 %) avec une probabilité supérieur ou égale à 95 % est donné par la formule

$$\frac{\sigma \times 1.96}{\sqrt{n}} \leq 0.03$$

où σ est l'écart type des mesures [40]. Pour obtenir la précision souhaitée, nous avons observé que 40 mesures étaient suffisantes quelle que soit l'instance de programme et pour la plupart des conditions expérimentales, avec ou sans activation du mode traçage. Pour faciliter les expériences, le même nombre de mesures a été effectué pour tous les bancs d'essai, même si la précision souhaitée pouvait être atteinte avec moins de mesures. Ainsi l'intervalle de confiance du temps moyen d'exécution de la plupart des bancs d'essai était meilleur que 3 % et donc l'intervalle de confiance sur le surcoût inférieur à 6 %. Ceci explique les différences entre les précisions des résultats donnés dans la figure 5.6. L'expérience totale représente plus de deux mille

exécutions de programmes ATHAPASCAN, réalisées essentiellement de nuit et les jours fériés.

5.4.4 Résultats des mesures du surcoût en temps

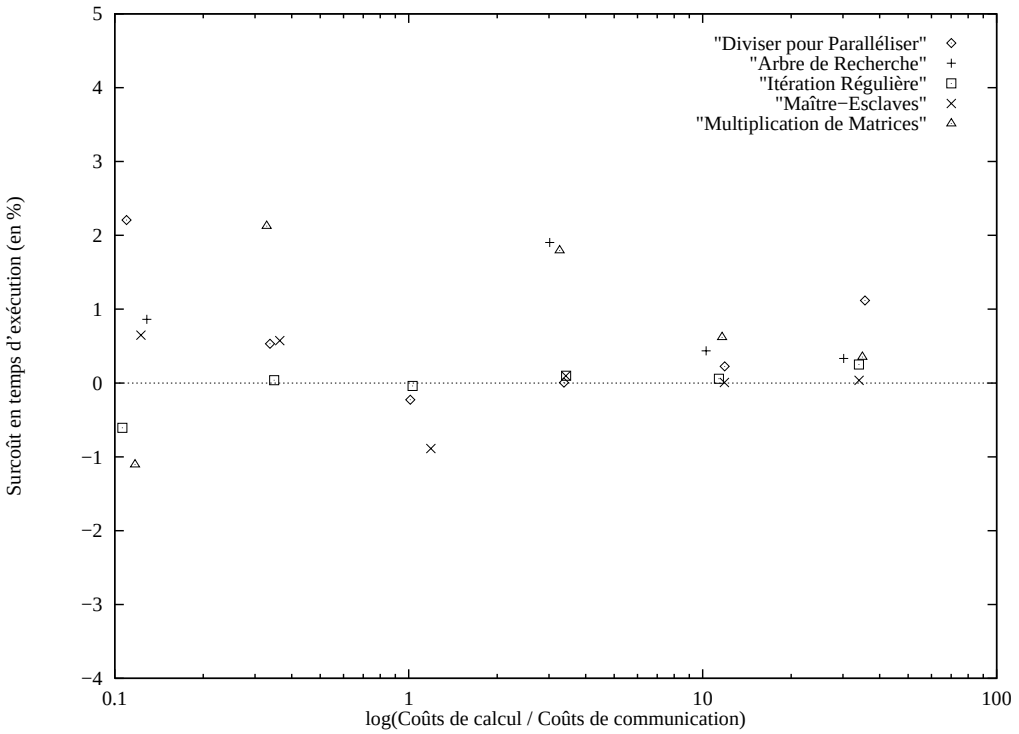


FIG. 5.5 – *Pourcentage du surcoût en temps dû au traçage.*

Les résultats des mesures sont résumés sur la figure 5.5 qui présente les surcoûts d'enregistrement moyens mesurés. Le tableau 5.5 détaille les valeurs du surcoût moyen mesuré, de la borne inférieure et de la borne supérieure de l'intervalle de confiance. Pour certaines expériences, les mesures n'ont pu être obtenues à cause des pannes fréquentes de la machine. En raison de l'évolution très rapide du contexte expérimental, il n'a pas été possible de les réaliser par la suite sans nécessiter de recommencer l'ensemble des expériences déjà réalisées.

Le principal résultat de ces mesures est que les surcoûts d'enregistrement sont inférieurs à 5 %, même pour les cas improbables où les coûts de communication représentent 10 fois les coûts de calcul. Les surcoûts moyens mesurés, dont la valeur est négative, sont dus aux conditions expérimentales. Il est

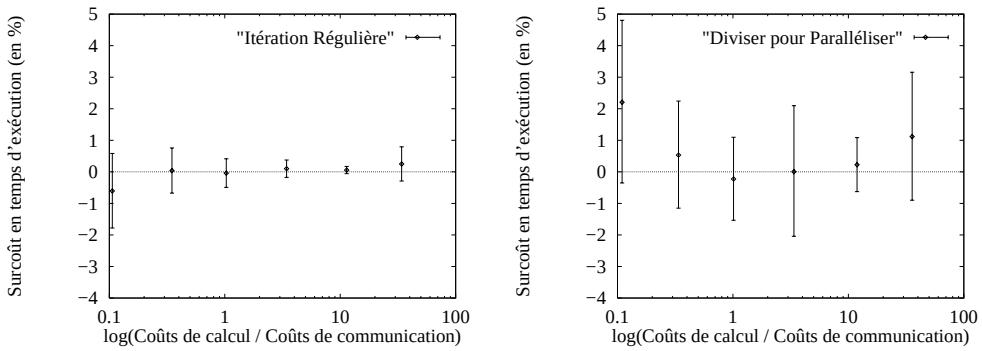


FIG. 5.6 – *Détails des surcoûts pour deux modèles.*

rappelé que la valeur réelle du surcoût appartient à l'intervalle de confiance autour de la moyenne mesurée. Aucun algorithme ne semble pathologique au regard du surcoût en temps dû à l'enregistrement. Il ne semble pas y avoir de règle simple liant les modèles d'algorithmes et la distribution du surcoût en temps dû à l'enregistrement de traces.

La figure 5.6 présente les intervalles de confiance pour le modèle le moins perturbé (*Itération Régulière*) et pour le modèle le plus perturbé (*Diviser pour Paralléliser*). Le modèle *Itération Régulière* est le moins perturbé mais ceci ne peut être généralisé aux autres schémas réguliers. En effet, le modèle *Diviser pour Paralléliser*, bien que régulier, présente les plus grandes incertitudes.

Une autre perspective sur les mesures consiste à observer le surcoût dû à l'enregistrement en fonction du rapport entre le nombre de requêtes traitées et le temps d'exécution. Cela rejoint davantage notre modèle de programmation pour lequel les opérations tracées sont les réceptions de requêtes. Les résultats sont présentés sur la figure 5.7 avec leurs intervalles de confiance. Le tableau 5.5 détaille les valeurs du surcoût moyen mesuré, de la borne inférieure et de la borne supérieure de l'intervalle de confiance. Encore une fois, il n'est pas possible de classer les algorithmes selon ce critère.

Les fréquences de traitement des requêtes pour le modèle *Multiplication de Matrices* sont concentrées autour de 10 requêtes par seconde. Pour tous les autres modèles les fréquences sont davantage distribuées. Cette concentration n'a pas d'explication claire. L'essentiel des expériences fait apparaître une fréquence comprise entre 10 et 40 requêtes traitées par seconde. Cela nous permet de donner une garantie de fonctionnement peu intrusif du mécanisme pour les applications qui traitent jusqu'à 40 requêtes par seconde. Cette

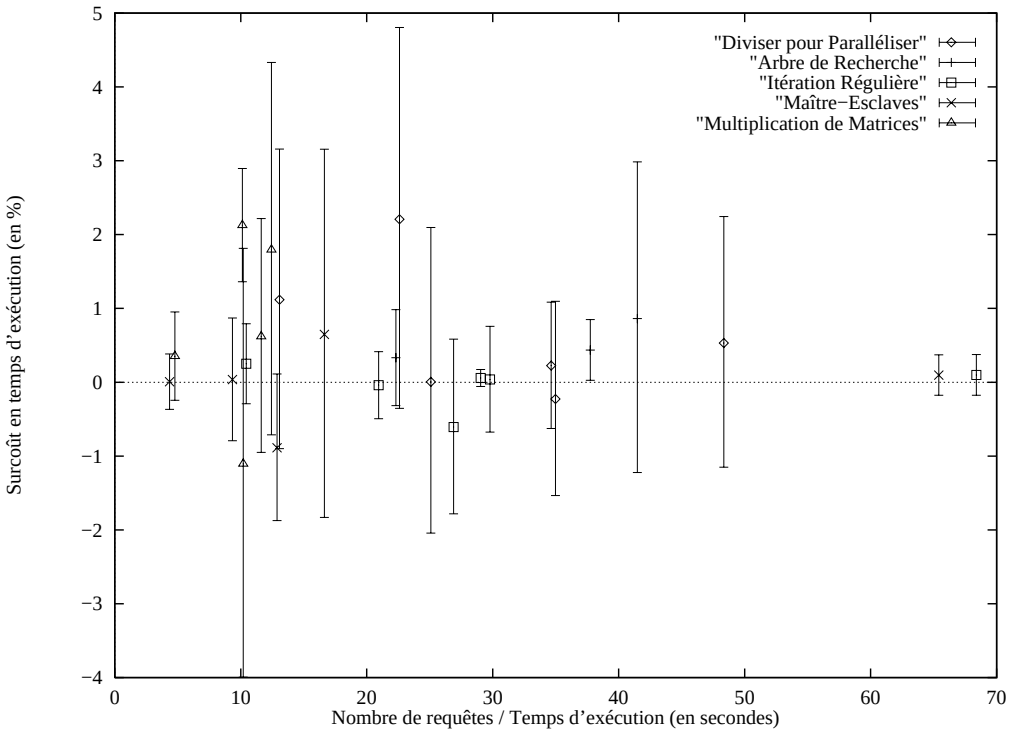


FIG. 5.7 – *Surcoût selon le nombre de requêtes traitées par seconde.*

garantie doit être valide jusqu'à 50 requêtes par seconde (en considérant les deux résultats obtenus dans la tranche entre 40 et 50 requêtes par seconde). D'autres mesures sont nécessaires pour explorer les fréquences au delà de 70 requêtes traitées par seconde.

Les mesures complémentaires effectuées sur la phase de réexécution montrent un surcoût du même ordre de grandeur que le surcoût dû à l'enregistrement. Cette observation permet de garantir aux programmeurs, lors des sessions de mise au point interactive, des temps d'exécution semblables à la durée de l'exécution enregistrée. Cette garantie s'entend hors intrusions d'un programmeur dans le déroulement d'une réexécution.

5.5 Conclusion

Les résultats obtenus montrent que le mécanisme d'enregistrement des traces pour la réexécution déterministe est très peu intrusif. Un surcoût inférieur à 5 % est garanti pour les programmes qui traitent moins de 50 requêtes par seconde. Le mode traçage peut raisonnablement être considéré comme

mode normal d'exécution pour le noyau exécutif ATHAPASCAN-0a. En raison de sa faible intrusion, ce mode d'enregistrement sert de base à Teodorescu [77] pour des méthodes de correction de l'intrusion due aux traces pour l'évaluation de performance (voir 6.1.4).

La méthodologie mise en place pour ces mesures a été appliquée dans un autre cadre [11]. Il s'agissait de comparer des stratégies de placement statique avec des stratégies de répartition dynamique de charge. Le générateur de programmes synthétiques a été adapté pour produire des programmes ATHAPASCAN-1 (couche de répartition dynamique). En raison de la constante évolution des noyaux exécutifs, de nouvelles versions des noyaux sont apparues avant la fin des mesures.

TAB. 5.1 – *Pourcentage du surcoût en temps dû au traçage.*

Modèle d'algorithme	Calcul/ Comm.	Borne inférieure	Surcoût moyen	Borne supérieure
Diviser pour Paralléliser	0.109651	-0.353205	2.20666	4.80418
	0.337548	-1.15004	0.531611	2.24336
	1.01268	-1.53488	-0.227752	1.09615
	3.37562	-2.0425	0.004401	2.09662
	11.8869	-0.626968	0.225055	1.08517
	35.6608	-0.899842	1.11739	3.15803
Arbre de Recherche	0.128617	-1.22217	0.862069	2.98495
	3.01705	1.25304	1.90237	2.55632
	10.2691	0.0255325	0.436321	0.848887
	30.1707	-0.316842	0.332814	0.983611
Itération Régulière	0.106175	-1.78193	-0.606152	0.583933
	0.348711	-0.674857	0.0389597	0.757602
	1.03067	-0.49316	-0.040541	0.414332
	3.42999	-0.175996	0.099481	0.375766
	11.3618	-0.05793	0.0572646	0.172611
	34.0241	-0.291925	0.250054	0.792991
Maître-Esclaves	0.122741	-1.83065	0.648634	3.15608
	0.36413	-0.462253	0.574418	1.62123
	1.1898	-1.87413	-0.887731	0.112899
	3.42551	-0.176667	0.0974915	0.372338
	11.8604	-0.366522	0.00715	0.382481
	34.0394	-0.791707	0.0358143	0.869428
Multiplication de Matrices	0.117252	-3.99505	-1.10609	1.81452
	0.328637	1.36146	2.12505	2.89488
	3.26038	-0.712857	1.79316	4.33064
	11.6487	-0.950904	0.61821	2.21668
	34.9454	-0.24445	0.352662	0.95272

TAB. 5.2 – *Surcoût selon le nombre de requêtes traitées par seconde.*

Modèle d'algorithme	Requêtes par seconde	Borne inférieure	Surcoût moyen	Borne supérieure
Diviser pour Paralléliser	13.0794	-0.899842	1.11739	3.15803
	22.6075	-0.353205	2.20666	4.80418
	25.0939	-2.0425	0.004401	2.09662
	34.6352	-0.626968	0.225055	1.08517
	34.9781	-1.53488	-0.227752	1.09615
	48.3468	-1.15004	0.531611	2.24336
Arbre de Recherche	22.3168	-0.316842	0.332814	0.983611
	37.7464	0.0255325	0.436321	0.848887
	41.4809	-1.22217	0.862069	2.98495
Itération Régulière	10.4411	-0.291925	0.250054	0.792991
	20.9507	-0.49316	-0.040541	0.414332
	26.902	-1.78193	-0.606152	0.583933
	29.0489	-0.05793	0.0572646	0.172611
	29.787	-0.674857	0.0389597	0.757602
	68.3906	-0.175996	0.099481	0.375766
Maître-Esclaves	4.35156	-0.366522	0.00715	0.382481
	9.3461	-0.791707	0.0358143	0.869428
	12.888	-1.87413	-0.887731	0.112899
	16.638	-1.83065	0.648634	3.15608
	65.4078	-0.176667	0.0974915	0.372338
Multiplication de Matrices	4.77779	-0.24445	0.352662	0.95272
	10.1278	1.36146	2.12505	2.89488
	10.2044	-3.99505	-1.10609	1.81452
	11.6311	-0.950904	0.61821	2.21668
	12.4368	-0.712857	1.79316	4.33064

Chapitre 6

Bilan et perspectives

Nous avons présenté des simplifications du mécanisme de réexécution déterministe dédié à ATHAPASCAN-0 par rapport à un mécanisme généraliste qui serait composé de la juxtaposition des mécanismes de réexécution déterministe pour le noyau de communication, pour le noyau de processus légers et pour les primitives de partage de mémoire. Selon le niveau d'abstraction considéré, des événements de noyaux différents peuvent se combiner en un seul. Pour ATHAPASCAN-0, c'est le cas de l'événement de réception d'un message d'appel de procédure et de l'événement de création d'un fil d'exécution. En passant d'un niveau d'abstraction à un autre, il y a simplification par la construction d'événements abstraits. L'exploitation de l'abstraction permet d'obtenir un mécanisme très peu intrusif comme le montrent les mesures. Un surcoût en temps inférieur à 5 % est mesuré pour tous les programmes qui traitent moins de 100 requêtes par seconde.

Le mécanisme de réexécution déterministe implanté dans le prototype du noyau exécutif ATHAPASCAN-0a a été utilisé par d'autres membres du projet APACHE. Il a permis d'une part de détecter une erreur furtive qui résistait à l'obstination d'un programmeur. D'autre part, il a servi à mettre en œuvre des méthodes de correction de l'intrusion pour l'évaluation de performances. Nous présentons ensuite une évaluation critique de la réexécution déterministe. La couche de portabilité ATHAPASCAN-0 a évolué vers la version ATHAPASCAN-0b du noyau exécutif. Les nouveautés introduites peuvent toujours être traitées par le modèle présenté au chapitre 3. Le paragraphe 6.3 présente une étude préliminaire pour la réalisation d'un mécanisme de réexécution déterministe pour ce noyau exécutif. Enfin d'autres perspectives, pour le projet APACHE et pour d'autres environnements, concluent ce chapitre.

6.1 Utilisations du mécanisme

Le mécanisme de réexécution déterministe pour ATHAPASCAN-0a a été utilisé d'une part pour mettre au point une application, d'autre part pour construire des méthodes de correction de l'intrusion due à l'enregistrement de traces pour l'évaluation de performances. Après l'utilisation effective, nous présenterons les limitations à l'utilisation du mécanisme. Nous décrirons ensuite son utilisation pour une expérience de tests d'ordonnancements puis son intégration dans un outil de correction des intrusions dues à l'enregistrement des traces pour l'évaluation de performances.

6.1.1 Utilisation effective

La première utilisation concrète du mécanisme de réexécution déterministe permit de corriger des erreurs dans l'implantation du prototype d'ATHAPASCAN-1 [71]. Le développeur avait déjà passé sans succès plusieurs nuits pour tenter d'isoler une erreur occasionnelle. Due à l'indéterminisme des exécutions, cette erreur furtive ne se produisait pas lorsque les processeurs virtuels étaient contrôlés par un débogueur séquentiel. Il est difficile de mettre au point un mécanisme de régulation de charge. En effet un tel mécanisme tire parti des ressources disponibles de manière opportuniste. L'indéterminisme des exécutions produites conduit à des erreurs furtives qui dépendent fortement de l'état de la machine. Après quelques exécutions tracées sans apparition d'erreur, le prototype ATHAPASCAN-1 a de nouveau exhibé le comportement erroné. À partir de cette trace, les recherches de la cause de l'erreur ont pu se dérouler comme sur une application déterministe.

6.1.2 Limitations à l'utilisation

Mis à part le développement d'ATHAPASCAN-1, les autres applications développées dans l'équipe en utilisant ATHAPASCAN-0 sont essentiellement numériques. Pour ces applications, l'indéterminisme a peu de chances *a priori* d'intervenir. Elles sont développées directement sur la couche ATHAPASCAN-0 pour valider les choix de réalisation et montrer la nécessité de cette couche de portabilité. Il semble que le besoin ne se soit pas encore fait sentir de recourir à la réexécution déterministe pour la mise au point de ces premières applications. Des applications assez régulières masquent l'indéterminisme des exécutions. Mais pour des problèmes irréguliers, la répartition de charge induit de l'indéterminisme dans les applications parallèles qui les traitent. De

telles applications sont davantage prédisposées à produire des erreurs furtives.

Une difficulté à l'utilisation de ce mécanisme est due à l'introduction tardive des sémaphores dans ATHAPASCAN-0. Les premières spécifications obligeaient les programmeurs à recourir au mécanisme de limitation de concurrence sur les points d'entrée. Différentes expérimentations ont mis en évidence les lourdeurs et les pertes de performances dues à ce mécanisme de synchronisation de niveau applicatif. Pour des raisons de performances, les programmeurs ont obtenu des mécanismes de synchronisation plus efficaces que la limitation de concurrence. Plusieurs modalités ont été réalisées dont les sémaphores, les barrières et les signaux. La conception minimaliste d'ATHAPASCAN-0a n'était plus d'actualité. Par la suite, l'implantation de la réexécution déterministe pour les sémaphores n'a pas été réalisée en raison de la bascule vers ATHAPASCAN-0b qui pose de nouveaux problèmes. La disponibilité de la plateforme ATHAPASCAN-0b permet maintenant la réalisation du mécanisme selon les principes exposés au paragraphe 6.3.2.

6.1.3 Tests d'ordonnancements

Une application originale de la réexécution déterministe consiste à la considérer comme un outil de test des ordonnancements. Comme le chapitre 2 la présente, la réexécution déterministe consiste à fixer l'ordre de certains événements lors d'une exécution guidée par des traces. Les traces exploitées lors de la réexécution comportent des indications sur l'ordre relatif de traitement des requêtes. Pour la réexécution, un ordonnancement est obtenu par l'enregistrement de l'ordre observé au cours d'une exécution. Dans le cas du test des ordonnancements, les traces ne sont pas obtenues par l'enregistrement d'une exécution. Elles sont produites par un outil de calcul d'ordonnancements.

Une étude préliminaire [11] a été ménée dans le cadre du projet APACHE avec la boîte à outil d'ordonnancement ALTO [8]. Un ordonnancement est qualifié de statique lorsqu'il est calculé avant l'exécution de l'application. Les ordonnancements statiques peuvent être transformés en traces pour guider une réexécution. Toutefois, il peut être envisagé de calculer dynamiquement un ordonnancement suivant l'évolution de la charge de calcul d'une application irrégulière qui dépend fortement des données. Les traces ne sont alors produites que peu avant d'être utilisées.

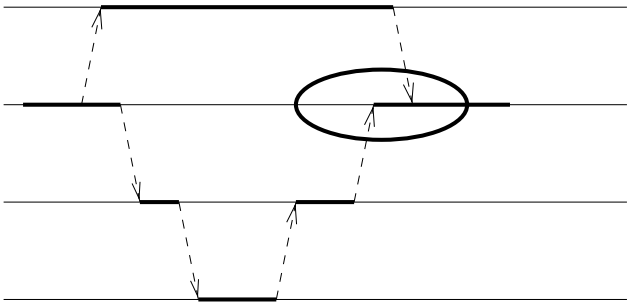
6.1.4 Évaluation de performances

Dans le cadre du développement d'outils pour l'environnement de programmation, Teodorescu [77] a basé sur la réexécution déterministe ses méthodes de correction de l'intrusion pour l'évaluation de performances. Un des problèmes les plus importants du traçage logiciel est la perturbation induite qui a deux conséquences directes : un décalage irrégulier des estampilles des événements et une altération de l'ordre des événements. Chaque estampille enregistrée est décalée de la valeur exacte d'une durée proportionnelle au nombre d'événements précédemment tracés. Ce décalage peut être corrigé précisément *post mortem* [55, 58].

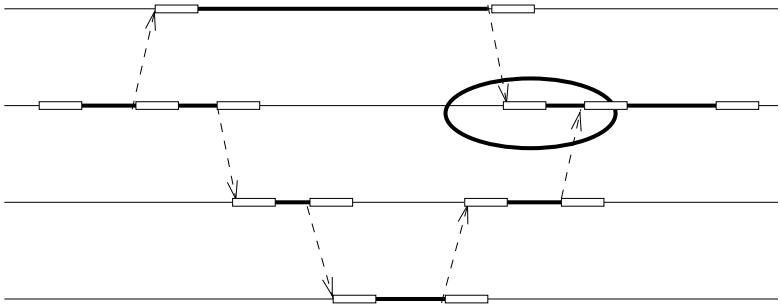
L'indéterminisme des exécutions d'un programme parallèle peut conduire la perturbation induite par le mécanisme de traçage à réduire l'espace des états du programme à un sous-ensemble particulier. Le traçage affecte également l'ordre des événements, modifiant ainsi le comportement de l'application observée. La figure 6.1 présente le cas d'un changement de comportement dû à l'accumulation irrégulière de l'intrusion sur différents processus. L'intrusion due à l'instrumentation est représentée par des rectangles blancs. L'ordre de réception des messages dans l'ovale est inversé en raison de l'instrumentation.

Pour ne pas perturber l'exécution, il faudrait avoir un mécanisme de traçage non intrusif. Entre cet idéal et un traceur pour l'évaluation de performances, un mécanisme tel que la réexécution déterministe propose un compromis. Le mécanisme d'enregistrement des traces pour la réexécution déterministe est peu intrusif. Cette propriété rend le mode traçage candidat au titre de mode d'exécution ordinaire. Le traçage minimal est actif en permanence. Les traces nécessaires à l'évaluation de performance sont prises lors d'une réexécution. Ces traces comportent alors une intrusion due à la réexécution. Une correction permet d'obtenir les temps qui auraient été observés lors de l'enregistrement initial.

L'évaluation de performances utilisant la réexécution comporte plusieurs phases : trace initiale pour la réexécution, réexécution tracée pour l'évaluation de performances et correction *post mortem* des estampilles. Durant la trace initiale, la valeur de l'horloge physique est enregistrée avec chaque événement nécessaire à la réexécution. Durant la réexécution, un traçage plus complet est effectué afin d'obtenir les dates de début et de fin d'exécution des processus légers, les dates d'émission des requêtes et de réception des réponses. En plus de la collecte de ces traces, un échantillonnage des horloges des processeurs



(a) exécution de référence



(b) exécution instrumentée

FIG. 6.1 – *Modification du comportement par le traçage.*

permet d'estimer un temps global du système parallèle.

La correction *post mortem* des estampilles utilise ce temps global et les traces des deux premières phases pour estimer les dates des événements correspondants, au cours de l'exécution initiale. La réexécution déterministe facilite cette correction des informations de performances par le déterminisme des réexécutions produites. Cela permet d'atténuer l'effet de sonde y compris sur des programmes indéterministes, par l'adaptation des méthodes dédiées aux exécutions déterministes. Quelques expériences encourageantes ont pu être faites pour lesquelles le temps d'exécution estimé après correction présente une différence de moins de 4 % du temps d'exécution lors de l'enregistrement des traces pour la réexécution.

6.2 Jugement sur la réexécution déterministe

6.2.1 Nécessité

La faible demande pour la réexécution déterministe dans notre équipe ne préjuge pas de l'absence de nécessité. Le chapitre 1 montre qu'elle est indispensable pour presque toutes les situations de mise au point. Son apport est sensible surtout pour les applications indéterministes. Les applications déterministes n'ont pas les mêmes problèmes de mise au point car leur comportement n'est pas aussi primesautier. Pour de telles applications, les outils de vérification peuvent opérer directement sur le code, voire sur une trace d'exécution. La perturbation de performances induite par le traçage n'en change presque pas le comportement. Les mêmes outils ont beaucoup de difficultés avec des applications dont le comportement est variable. L'espace des comportements possibles devient impraticablement grand. L'introduction des processus légers rend encore plus difficile les calculs de propriétés des exécutions, surtout par le grand nombre d'entités.

6.2.2 Utilité

Dans le projet APACHE, la réexécution déterministe a montré son utilité avec le prototype ATHAPASCAN-0a. La plateforme ATHAPASCAN-0b s'est enrichie de fonctionnalités qu'il reste possible de réexécuter. La définition d'un noyau exécutif doit s'accompagner d'une réflexion sur la définition d'un mécanisme de réexécution déterministe adapté. Le paragraphe 6.3 présente le résultat d'une telle réflexion. Toutes les fonctionnalités d'ATHAPASCAN-0b offertes aux programmeurs respectent le concept fondamental de *point d'entrée*.

Dans le cas général des systèmes exploitant les fils d'exécution comme support pour le parallélisme, la définition d'un mécanisme de réexécution déterministe peut être compliquée par des fonctionnalités trop pointues. Un des rôles du groupe de développement des outils de mise au point peut être de limiter les ambitions des définisseurs du langage.

6.2.3 Difficultés

Des difficultés techniques peuvent ensuite survenir pour mettre en œuvre un mécanisme de réexécution déterministe. Ces difficultés peuvent soit empê-

cher totalement l'implantation d'un tel mécanisme dans la réalisation retenue du noyau exécutif, soit mener à une implantation inefficace. L'anticipation de ces difficultés permet d'orienter la conception originelle afin qu'elle puisse être étendue par un socle d'atelier de mise au point des applications. Il y a parfois antinomie entre une réalisation efficace du noyau exécutif et une conception permettant l'intégration d'un mécanisme de réexécution déterministe.

La mémoire d'un processeur virtuel est utilisée par le système d'exploitation, le noyau de processus légers, le noyau de communications, le noyau ATHAPASCAN-0, l'éventuel traceur et le programme de l'utilisateur. Pour bien s'entendre, chacun doit s'en tenir à des opérations protégées sur des zones réservées. Chaque noyau doit offrir toutes ses fonctions sous une forme réentrante. C'est-à-dire que plusieurs appels peuvent avoir lieu avec des contextes de processus légers différents. Comme l'ordonnancement entre les processus légers ne peut être facilement choisi, il faut le considérer comme une contrainte de l'environnement qui fait progresser l'exécution du programme. Pour le programmeur, l'environnement de programmation doit offrir un outil de contrôle des codes sources de l'application pour assurer que tout accès à la mémoire est correctement protégé. En dehors de cette garantie, la réexécution déterministe peut être très difficile, voire impossible, à mettre en œuvre.

6.3 Réexécution déterministe pour Athapascan-0b

Les modèles de programmation parallèle basés sur les processus légers ne sont pas encore standardisés. Nous nous sommes intéressés à ATHAPASCAN-0b en supposant que le modèle ATHAPASCAN offre un ensemble de fonctionnalités souvent présentes dans d'autres systèmes (voir par exemple PM^2 [67], Nexus [27] ou Chant [35]). Pour la plupart des systèmes, les processus légers peuvent être directement manipulés par l'application, en terme de création (locale ou à distance), de règles d'ordonnancement et de priorités. Des primitives permettent de gérer la synchronisation entre eux, en particulier pour protéger les accès à la mémoire du processeur virtuel (partage entre processus légers locaux). Enfin les primitives de communication sont étendues pour permettre l'échange de messages entre processus légers. Toutefois un système comme PM^2 présente aussi des caractéristiques innovantes telles que la migration des processus légers.

6.3.1 Plateforme Athapascan-0b

La couche de portabilité ATHAPASCAN a évolué par la prise en compte des retours d'information des programmeurs. Entre la spécification initiale et la version finale du prototype ATHAPASCAN-0a, les sémaphores ont fait leur apparition pour le programmeur. Toutefois les performances de ce noyau exécutif pouvaient être encore améliorées. L'adéquation aux besoins des développeurs de la couche ATHAPASCAN-1 a aussi beaucoup contribué à l'évolution du prototype vers la plateforme ATHAPASCAN-0b. Outre les sémaphores, on y trouve les verrous. Ces mécanismes permettent de gérer le partage de la mémoire et la synchronisation entre les processus légers d'un même processeur virtuel.

L'absence de communication directe entre les processus légers obligeait à n'utiliser que l'appel de procédure à distance. Cela introduisait des lourdeurs pour des fonctions dont les paramètres ou les résultats étaient volumineux. Le modèle présenté au paragraphe 3.1 montre qu'il est possible de relâcher les contraintes sur les envois des paramètres et des résultats. Les aménagements des spécifications d'ATHAPASCAN-0a prévoyaient ce relâchement par l'ajout de flux de données (en anglais *data streams*). Cette introduction des flux de données ouvre également la possibilité de communications directes entre processus légers. En ATHAPASCAN-0b, l'échange de messages ne se limite pas à la structure présentée par la figure 3.3. Les processus légers peuvent librement échanger des messages avec tous ceux qu'ils connaissent.

Enfin à un niveau moins directement perceptible par le programmeur, l'introduction de la préemption et des priorités des processus légers a permis de répondre au besoin de traitements urgents. Avec l'attribution du processeur aux fils d'exécution par tranches de temps, une requête reçue n'a pas besoin d'attendre, pour être prise en compte, la fin de l'exécution du processus léger actif mais uniquement la fin de sa tranche de temps. Cette possibilité permet d'offrir un nouveau mécanisme de traitement des requêtes. Il se rapproche de celui que certains systèmes désignent sous le terme de «messages actifs» (en anglais *active messages*). En plus des appels habituels à une procédure, il est possible, avec certaines restrictions, d'appeler une procédure urgente. Une telle procédure ne doit pas utiliser de primitive bloquante qui interdirait au noyau exécutif de traiter les requêtes suivantes. Ainsi il n'est pas possible d'appeler une autre procédure ou d'effectuer des communications. Une application opportuniste peut, par ce mécanisme, interroger l'état de charge du processeur afin de décider du placement de ses futurs calculs.

Une extension en cours de développement permet les accès à la mémoire à distance. Ce mécanisme vise à implanter une forme de mémoire partagée sur une architecture à mémoire distribuée. Les requêtes d'accès à la mémoire sont traitées sur chaque processeur virtuel par un démon qui sérialise les appels. Les opérations disponibles sont la lecture, l'écriture et le verrouillage d'une zone de la mémoire. La sérialisation est totale, ce qui réalise un protocole d'accès exclusifs à chaque zone de la mémoire. La mémoire partagée à distance est donc gérée différemment de la mémoire partagée localement.

6.3.2 Stratégie d'implantation

Le traitement des primitives de synchronisation entre processus légers est proposé au paragraphe 3.3.2. Deux modes de synchronisation sont disponibles en ATHAPASCAN-0b, les sémaphores et les verrous. Dans notre modèle, chaque mode de synchronisation entre processus légers est géré par un point d'entrée «virtuel» défini par le système sur chaque processeur virtuel. Ce type de point d'entrée a la particularité de s'exécuter dans le contexte du processus léger appelant. Il n'y a pas création d'un processus léger pour exécuter la procédure associée au point d'entrée. Celle-ci s'exécute donc comme une procédure normale appelée par le processus léger. Les indications données au paragraphe 4.2.1 servent de guide pour la réalisation.

Les communications directes entre processus légers ne peuvent pas être intégrées dans le cas général au modèle d'exécution proposé au chapitre 3. Proposer une solution générale dans le cadre du modèle est difficile en raison des solutions très différentes au problème de la désignation du processus léger destinataire. Toutefois, les ports utilisés par ATHAPASCAN-0b peuvent être considérés comme des points d'entrée «virtuels» s'exécutant dans le contexte du processus léger qui gère les communications. Cette approche s'intègre parfaitement dans la solution générale développée (voir paragraphe 3.3).

Pour le traitement de la préemption et des changements de règle d'ordonnancement et de priorité, il faut définir des objectifs relativement aux besoins exprimés au chapitre 1. Il ne semble pas nécessaire de toujours présenter au programmeur l'instruction précise sur laquelle le changement de contexte a eu lieu. Comme l'ordonnancement des fils d'exécution peut ne pas être maîtrisé par le noyau ATHAPASCAN-0, cette information n'apporte aucune aide. D'autre part, même s'il est possible de considérer le processeur comme une ressource partagée, un changement de contexte ne constitue pas une synchronisation pour un accès à des données partagées. Celles-ci se trouvent soit

dans des paramètres d'appel ou de résultat, soit dans des messages, soit dans des zones de mémoire protégées.

Le démon serveur de mémoire partagée peut être vu comme une instanciation particulière d'un point d'entrée à concurrence bornée à 1. Au lieu de créer et détruire le processus léger pour traiter chaque requête, il n'est créé qu'une fois pour des raisons de performances. Il doit donc lui-même gérer la bande du point d'entrée qu'il représente. La sérialisation du traitement des requêtes lors de la réexécution est donnée par l'ordre enregistré plutôt que par l'ordre effectif de l'exécution.

6.3.3 Difficultés attendues

Les difficultés attendues sont principalement d'ordre technique. Pour des raisons d'efficacité et de portabilité, les communications sont gérées par la bibliothèque MPI. Celle-ci ne permet pas d'insérer dans les messages des informations invisibles à l'application. Pour contourner cette limitation, il est possible d'envisager l'envoi de messages de contrôle. Le rôle de ces messages serait de fournir, aux primitives instrumentées, les informations qui ne peuvent figurer dans les messages de l'application. L'envoi d'un message pour l'application serait immédiatement suivi (ou précédé) de l'envoi d'un message de contrôle pour l'instrumentation. Cette solution est difficile à mettre en place. Son coût en terme de performances de l'application instrumentée ne peut être estimé par avance. Les messages de contrôle introduits doivent être traités à part par le noyau. Ils ne perturbent pas directement l'application mais leur intrusion n'est pas nulle.

Par ailleurs, l'utilisation d'un noyau préemptif de processus légers lève des problèmes pour l'écriture des traces dans les tampons d'enregistrement. Des écritures concurrentes peuvent se produire lors d'un changement de contexte décidé par l'ordonnanceur des processus légers. Il faut donc protéger ces structures de données partagées lors de toute écriture. L'utilisation de verrous sur une unique structure commune s'avère très coûteuse s'il y a réservation et libération lors de chaque écriture. Pour la gestion des tampons d'enregistrement, une solution a été développée par Waille (de l'équipe APACHE) pour le traceur de mesure de performances. Un groupe de tampons est créé sur le processeur virtuel lors de son initialisation. Chaque processus léger, qui désire enregistrer des traces, réserve un tampon de ce groupe. Lorsque le tampon est plein, le processus léger en réserve un autre. Le tampon plein est transmis à un processus léger dédié qui en contrôle l'écriture sur le disque. Le tampon

vidé peut alors être remis dans le groupe des tampons libres. Cette solution évite la lourdeur d'un verrouillage et d'un déverrouillage systématiques lors de chaque écriture dans un tampon.

6.4 Autres perspectives

Le travail exposé dans cette thèse offre d'autres perspectives à plus long terme. Elles concernent aussi bien le projet APACHE que d'autres environnements basés sur les processus légers.

6.4.1 Pour le projet APACHE

Le modèle ATHAPASCAN-1 présente au programmeur une interface de plus haut niveau. Le langage fait totalement abstraction de la machine qui supporte l'exécution. La répartition des calculs est assurée dynamiquement, de manière presque transparente pour le programmeur. Toutefois, pour la mise au point des régulateurs de charge, il pourrait être intéressant de voir également le niveau ATHAPASCAN-0. Dans ce cas une réexécution déterministe du noyau ATHAPASCAN-0 pourrait suffire. L'étude de la réexécution déterministe pour ATHAPASCAN-1 devra s'attacher à la mise en évidence d'événements composés comme il en existe pour ATHAPASCAN-0 relativement aux couches sur lesquelles il est construit. Le volume des traces nécessaires à la réexécution peut ainsi être réduit à moindre frais.

La réexécution déterministe sur ATHAPASCAN-0 peut servir de plateforme d'expérimentation pour les outils de placement et d'ordonnancement statiques. Un important travail théorique a été réalisé sur ce thème dans l'équipe APACHE. Un début d'expérimentation a eu lieu [11] mais n'a pas été poursuivi. La méthodologie reprenait celle développée pour les mesures du surcoût en temps dû à l'enregistrement des traces pour la réexécution. L'objectif était d'aboutir à une comparaison entre ordonnancement statique et régulation dynamique.

6.4.2 Pour d'autres environnements

Dans notre approche nous traitons les points d'entrée comme des ressources partagées, selon le modèle initial de la réexécution déterministe. Une approche plus généraliste pourrait être de ne pas considérer les points d'entrée mais uniquement les processus légers. Toutefois cette approche généra-

liste peut perdre les simplifications possibles par abstraction de combinaisons d'événements. Il y a en fait une grande similitude entre un processeur virtuel et un multiprocesseur symétrique. Cette analogie est voulue par les concepteurs d'environnements de développement afin d'offrir une solution générale pour la programmation des machines parallèles. Les processus légers d'un modèle peuvent être comparés aux processus lourds de l'autre. Ainsi tous les travaux sur la réexécution déterministe, avec synchronisation des processus par mémoire partagée, par échange de messages ou par appel de procédure, peuvent être appliqués au sein d'un processeur virtuel. La coordination entre processeurs virtuels peut être traitée par un modèle existant ou adapté.

La migration de fils d'exécution est traitée dans certains systèmes comme PM² [67]. Comme conséquence de la migration, un processus peut subir des interactions sur plusieurs processeurs virtuels différents, voire sous plusieurs identités différentes. L'histoire d'un processus migré est enregistrée partiellement sur plusieurs bandes. Une première approche consiste à considérer la migration comme un appel asynchrone sans attente de résultat. Les paramètres de l'appel comportent le contexte du processus léger appelant afin que le processus créé puisse poursuivre le calcul en cours. Lors de la réexécution, la migration doit être retardée jusqu'à la réalisation de toutes les interactions qui ont été enregistrées sur le site d'origine. Sur le site de destination du processus, les interactions doivent attendre la migration effective. La couche de communication ne doit pas rejeter ou rediriger un message «en avance» sur le processus migrant.

Conclusion

D'une manière générale, la mise au point de programmes exploitant plusieurs flots d'exécution est difficile. L'indéterminisme des exécutions est un obstacle considérable à la compréhension du comportement d'une application erronée. La technique de réexécution déterministe est généralement utilisée pour se replacer dans le cadre de la correction d'une application déterministe. Cette technique est adaptée pour divers modèles de programmation. Le modèle procédural parallèle en bénéficie désormais par les travaux présentés dans cette thèse. L'objectif de l'enregistrement de traces est de permettre la réexécution de toute application erronée. Pour être mise en œuvre efficacement, cette technique nécessite de rendre possible l'enregistrement pour toutes les exécutions. Dans cette perspective, l'enregistrement doit rester aussi peu intrusif que possible. Pour cela des améliorations apportées à la technique de base consistent à ne tracer qu'une partie des dépendances entre processus.

Cette thèse définit un modèle d'exécution de programmes parallèles basés sur l'appel de procédures exécutées à distance par des processus légers. L'équivalence de deux exécutions dans le modèle procédural parallèle sert de guide pour la construction d'un mécanisme de réexécution déterministe pour le noyau exécutif Athapascan-0a. Le modèle étudié n'est pas spécifique à Athapascan et peut s'appliquer à d'autres environnements de programmation. Les modèles concernés utilisent des ressources partagées actives. Le contrôle des accès est assuré par chaque ressource. Elle est chargée d'assurer l'enregistrement et la réexécution de l'ordre des autorisations d'accès qu'elle distribue. Dans cette famille de modèles se trouvent, entre autres, les modèles d'appels de procédures à distance, les modèles *Clients-Serveurs* et les modèles à objets actifs. Le modèle exploite des événements abstraits composés d'événements de plus bas niveau. Le nombre d'événements enregistrés est ainsi réduit sans surcoût de calcul lié à cette amélioration.

La réalisation pratique du mécanisme pour Athapascan-0a a été validée sur une application très indéterministe. Celle-ci montre le fonctionnement

correct du mécanisme mais ne permet d'estimer le surcoût dû à l'enregistrement des traces. Pour évaluer le surcoût en temps introduit par le traçage, une série de mesures systématiques a été mise en œuvre. Les mesures des temps d'exécution s'effectuent pour des programmes parallèles synthétiques générés à partir de modèles d'algorithmes. Elles ont permis de montrer l'efficacité du traçage en terme de surcoût en temps d'exécution. La méthode mise en place pour ces mesures a pu être réutilisée dans le cadre du projet pour d'autres mesures comparatives. En plus des vérifications de la correction de l'implantation, ce mécanisme a été utilisé par d'autres membres de l'équipe dans une situation réelle de mise au point. Dans le cadre de l'intégration avec un outil d'évaluation de performances, la réexécution déterministe sert de mécanisme primaire de traçage peu intrusif. Les traces d'évaluation de performances sont enregistrées au cours d'une réexécution avant d'être corrigées pour effacer l'intrusion du traçage. Les premières expériences présentent des résultats encourageants.

Bien qu'elle soit largement répandue dans le monde des outils universitaires, la réexécution déterministe n'est pas encore offerte par les ateliers de mise au point industriels. Cependant elle est définie pour un grand nombre de modèles de programmation. Notre contribution originale pour le modèle procédural parallèle concourt à en élargir davantage le champ d'application. Un dernier problème technique peut en limiter la diffusion générale. Il s'agit du partage de la mémoire entre les différents noyaux sur lesquels se base une application. Même s'il existe des situations où les programmeurs ou les systèmes peuvent la mettre en échec, les services qu'elle peut rendre dans un grand nombre de cas en justifient la généralisation. C'est un confort pour tout programmeur qui respecte certaines règles d'expression dans son code. En contrepartie, elle préserve la liberté et la puissance offertes par l'écriture d'une application parallèle opportuniste.

Annexe A

Glossaire – Glossary

Certains termes sont mieux connus sous leur forme anglaise car la littérature est plus abondante dans cette langue. Le respect de la législation française oblige à l'emploi de termes qui paraissent étrangers, même aux spécialistes du domaine. Pour ne pas égarer les lecteurs, ce glossaire peut aider à établir les équivalences entre les deux langues.

A.1 Anglais – français

active message: message actif,

behavioral abstraction: abstraction comportementale,

data stream: flux de données,

entry point: point d'entrée,

heavyweight process: processus lourd, processeur virtuel,

lightweight process: processus léger, fil d'exécution,

Lightweight Remote Procedure Call: appel léger de procédure à distance,

mutex: verrou,

probe effect: effet de sonde,

record: enregistrement,

Remote Procedure Call: appel de procédure à distance,

replay: réexécution,

thread: fil d'exécution, processus léger,

Very Large Instruction Word: mot d'instruction très long,

virtual processor: processeur virtuel, processus lourd,

A.2 Français – anglais

abstraction comportementale : behavioral abstraction,

appel de procédure à distance : Remote Procedure Call,

appel léger de procédure à distance : Lightweight Remote Procedure Call,

effet de sonde : probe effect,

enregistrement : record,

fil d'exécution : thread, lightweight process,

flux de données : data stream,

message actif : active message,

mot d'instruction très long : Very Large Instruction Word,

point d'entrée : entry point,

processeur virtuel : virtual processor, heavyweight process,

processus lourd : heavyweight process, virtual processor,

processus léger : lightweight process, thread,

réexécution : replay,

verrou : mutex,

Bibliographie

[1] Anonymous. – Special issue: High Performance Fortran language specification. *ACM Fortran Forum*, vol. 13, n° 2, juin 1994.

[2] Audenaert (K.) et Levrouw (L.). – Replaying programs allowing interrupts on a multiprocessor. *In : Proceedings International Workshop on Systems Engineering for Real Time Applications (SERTA '93)*. pp. 107–112. – IEEE Press.

[3] Balter (R.), Bernadat (J.), Decouchant (D.), Duda (A.), Freyssinet (A.), Krakowiak (S.), Meysembourg (M.), Le Dot (P.), Nguyen Van (H.), Paire (E.), Riveill (M.), Roisin (C.), Rousset de Pina (X.), Scioville (R.) et Vandôme. (G.). – Architecture and Implementation of Guide, an Object-Oriented Distributed System. *Computing Systems*, vol. 4, n° 1, 1991, pp. 31–67.

[4] Bates (P.). – *The EBBA Modelling Tool, a.k.a. Event Definition Language*. – Rapport technique n° 87-35, University of Massachussetts, avril 1987.

[5] Bates (P.C.) et Wileden (J.C.). – EDL: A basis for distributed system debugging tools. *In : Proceedings of the Fifteenth Hawaii International Conference on System Sciences*, pp. 86–93.

[6] Bershad (B.N.), Anderson (T.E.), Lazowska (E.D.) et Levy (H.M.). – Lightweight remote procedure call. *ACM Trans. on Computer Systems*, vol. 8, n° 1, février 1990, pp. 37–55.

[7] Birrell (A.) et Nelson (B.). – Implementing Remote Procedure Calls. *ACM Trans. on Comp. Sys.*, vol. 2, 1984, pp. 39–59.

[8] Bouvry (P.), Chassin de Kergommeaux (J.) et Trystram (D.). – Efficient solutions for mapping parallel programs. *In : Euro-Par'95 Parallel*

Processing, éd. par Haridi (S.), Ali (K.) et Magnusson (P.). – Springer-Verlag.

[9] Budkowski (S.) et Dembinski (P.). – An introduction to Estelle: a specification language for distributed systems. *Computer Networks and ISDN Systems*, vol. 14, 1987, pp. 3–23.

[10] Calvin (C.), Colombet (L.), Desprez (F.), Jargot (B.), Michallon (P.), Tourancheau (B.) et Trystram (D.). – Towards mixed computation-communication scientific libraries. *In : Proceedings of CONPAR 94*, éd. par Buchberger et Volker. – Linz - Austria, 1994.

[11] Castañeda (M.R.), Fagot (A.), Guinand (F.), Vermeerbergen (A.) et Kitajima (J.P.). – Comparison of the Performance of Static and Dynamic Scheduling. *In : Parallel Computing : State-of-the Art Perspective (ParCo95)*, éd. par D'Hollander (E.H.), Joubert (G.R.), Peters (F.J.) et Trystram (D.), pp. 431–437. – North-Holland, 1996.

[12] Chandy (K.M.) et Lamport (L.). – Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. on Comp. Sys.*, vol. 3, n° 1, février 1985, pp. 63–75.

[13] Charron-Bost (B.). – Concerning the Size of Logical Clocks in Distributed Systems. *Information Processing Letters*, vol. 39, 1991, pp. 11–16.

[14] Christaller (M.). – *Athapascan-0a sur PVM3: définition et mode d'emploi.* – Rapport APACHE n° 11, Grenoble, IMAG, février 1994.

[15] Clémençon (C.), Endo (A.), Fritscher (J.), Müller (A.) et Wylie (B.J.N.). – *Annai Scalable Run-time Support for Interactive Debugging and Performance Analysis of Large-scale Parallel Programs.* – Rapport technique n° TR-96-04, Manno, Switzerland, Centro Svizzero di Calcolo Scientifico, avril 1996.

[16] Cooper (R.) et Marzullo (K.). – Consistent detection of global predicates. *In : ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 167–174.

[17] Curtis (R.S.) et Wittie (L.D.). – BugNet : A Debugging System for Parallel Programming Environments. *In : Proceedings of the Third International Conference on Distributed Computing Systems*, pp. 394–399. – Miami, octobre 1982.

[18] de Oliveira Stein (B.) et Chassin de Kergommeaux (J.). – Environnement de visualisation de programmes parallèles basés sur les fils d'exécution. *In : Actes de RenPar'9*, éd. par Schiper (A.) et Trystram (D.). Rencontres francophones du parallélisme. – EPF Lausanne, mai 1997.

[19] Decouchant (D.), Duda (A.), Freyssinet (A.), Paire (E.), Riveill (M.), Rousset de Pina (X.) et Vandôme (G.). – Guide: an Implementation of the Comandos Object-Oriented architecture on Unix. *In : Porc. European Unix Users Group (EUUG) Autumn Conference.* – Lisbon, octobre 1988.

[20] Ducassé (M.). – *An extendable trace analyser to support automated debugging.* – Thèse de PhD, University of Rennes I, France, juin 1992. European Doctorate.

[21] Ducassé (M.). – Automated Debugging Extensions of the Opium Trace Analyser. *In : Proceedings of the Second International Workshop on Automated and Algorithmic Debugging.* – Saint Malo, France, mai 1995.

[22] Eykholt (J.R.), Kleiman (S.R.), Barton (S.), Faulkner (R.), Shivalingiah (A.), Smith (M.) et Stein (D.). – Beyond multiprocessing - multithreading the sunos kernel. *In : Proc. of the Usenix Conference.* SunSoft Inc.

[23] Fagot (A.) et Chassin de Kergommeaux (J.). – Formal and experimental validation of a low-overhead execution replay mechanism. *In : Euro-Par'95 Parallel Processing*, éd. par Haridi (S.), Ali (K.) et Magnusson (P.). pp. 167–178. – Springer-Verlag.

[24] Fagot (A.) et Chassin de Kergommeaux (J.). – Systematic assessment of the overhead of tracing parallel programs. *In : Proceedings of the 4th Euromicro Workshop on Parallel and Distributed processing, PDP'96*, éd. par Zapata (E.L.). – Braga, janvier 1996.

[25] Fidge (C.J.). – Partial Orders for Parallel Debugging. *In : Proceedings ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 183–194.

[26] Fidge (C.J.). – Logical Time in Distributed Computing Systems. *IEEE Computer*, août 1991, pp. 28–33.

[27] Foster (I.), Kesselman (C.), Olson (R.) et Tuecke (S.). – *Nexus: An interoperability toolkit for parallel and distributed computing.* – Technical Report n° ANL/MCS-TM-189, Argonne National Laboratory, 1994.

[28] Fowler (J.) et Zwaenepoel (W.). – Causal Distributed Breakpoints. *In : Proc. 10th IEEE Int. Conf. on Distributed Computing Systems.* – Paris, mai 1990.

[29] Francioni (J.M.), Albright (L.) et Jackson (J.A.). – Debugging parallel programs using sound. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, vol. 26, n° 12, décembre 1991, pp. 68–75.

[30] Fromentin (E.). – *Détection de propriétés instables dans les exécutions réparties, application à la mise au point des programmes répartis.* – Thèse de PhD, Université de Rennes 1, novembre 1996.

[31] Gait (J.). – A Probe Effect in Concurrent Programs. *Software - Practice and Experience*, vol. 16, n° 3, mars 1986, pp. 225–233.

[32] Geib (J.M.). – Processus légers distribués et régulation de charge. *In : École Placement Dynamique et Répartition de Charge.* École Française de Parallélisme, Réseaux et Systèmes, pp. 89–102. – Presqu'île de Giens, juillet 1996.

[33] Geist (G.A.), Heath (M.T.), Peyton (B.W.) et Worley (P.H.). – *A user's guide to PICL: a portable instrumented communications library.* – Rapport technique n° ORNL/TM-11616, Oak Ridge, Tennessee, Oak Ridge National Laboratory, janvier 1992.

[34] Guidec (F.) et Mahéo (Y.). – POM : une machine virtuelle parallèle incorporant des mécanismes d'observation. *Calculateurs Parallèles*, vol. 7, n° 2, 1995, pp. 101–118.

[35] Haines (M.), Cronk (D.) et Mehrotra (P.). – On the design of chant: A talking threads package. *In : Proc. Supercomputing'94.*

[36] Heath (M.T.) et Etheridge (J.A.). – Visualizing the Performances of Parallel Programs. *IEEE Transactions on Software Engineering*, vol. 8, n° 5, mai 1991, pp. 29–39.

[37] High Performance Fortran Forum. – *High Performance Fortran Language Specification: Version 1.0.* – Rapport technique, Rice University, mai 1993.

[38] Hurfin (M.), Plouzeau (N.) et Raynal (M.). – 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.

[39] Intel Corporation. – *iPSC/2 DECON User's Guide*, octobre 1989.

[40] Jain (R.). – *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling.* – New York, John Wiley and Sons, 1991, *Wiley Professional Computing.*

[41] Jamrozik (H.). – *Aide à la Mise au Point des Applications Parallèles et Réparties à base d'Objets Persistants.* – Grenoble, Thèse de PhD, Université Joseph Fourier, mai 1993.

[42] Jard (C.), Jeron (T.), Jourdan (G.V.) et Rampon (J.X.). – A general approach to trace-checking in distributed computing sytems. *In : 14th IEEE International Conference on Distributed Computing Systems*, pp. 396–403.

[43] Jard (C.) et Jézéquel (J.M.). – A multi-processor Estelle to C compiler to experiment distributed algorithms on parallel machines. *In : Proc. of the 9th IFIP Int. Workshop on Protocol Specification, Testing and Verification.* – University of Twente, 1989.

[44] Jones (S.), Barkan (R.H.) et Wittie (L.D.). – BugNet: A Real-Time Distributed Debugging System. *In : Proc. of the 6th Int. Symposium on Reliability in Distributed Software and Database Systems*, pp. 56–65.

[45] Kitajima (J.P.) et Plateau (B.). – Modelling parallel program behaviour in *ALPES. Information and Software Technology*, vol. 36, n° 7, juillet 1994, pp. 457–464.

[46] Kohl (J.A.) et Geist (G.A.). – The PVM 3.4 Tracing facility and XPVM 1.1. *In : Proceedings of the 29th Hawaii International Conference on System Sciences.* – Hawaii, 1996.

[47] Kunz (T.) et Black (J.P.). – Abstract Debugging of Distributed Applications. *In : Working conference on programming environments for massively parallel distributed systems.* IFIP, WG10.3, pp. 353–358. – Ascona, Switzerland, avril 1994.

[48] Lamport (L.). – Time, Clocks and Ordering of Events in a Distributed System. *Communications of the ACM*, vol. 21, n° 7, juillet 1978, pp. 558–565.

[49] Larus (J.R.). – Efficient program tracing. *IEEE Computer*, vol. 26, n° 5, mai 1993.

[50] LeBlanc (T.J.) et Mellor-Crummey (J.M.). – Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, vol. C-36, n° 4, avril 1987, pp. 471–481.

[51] Leu (E.). – *La réexécution, pierre angulaire de la mise au point des programmes parallèles.* – Thèse de PhD, École Polytechnique Fédérale de Lausanne, 1992.

[52] Leu (E.), Schiper (A.) et Zramdini (A.). – Execution Replay on Distributed Memory Architectures. *In : Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pp. 106–112. – Dallas, USA, décembre 1990.

[53] Levrouw (L.J.), Audenaert (K.M.R.) et Van Campenhout (J.M.). – A New Trace and Replay System for Shared Memory Programs based on Lamport Clocks. *In : Proceedings of the Euromicro Workshop on Parallel and Dsitributed Processing.* pp. 471–478. – Malaga, Spain, janvier 1994.

[54] Maillet (É.). – Issues in Performance Tracing with Tape/PVM. *Calculateurs Parallèles : numéro thématique PVM*, vol. 8, n° 2, 1996, pp. 189–202.

[55] Maillet (É.). – *Le traçage logiciel d'applications parallèles : conception et ajustement de qualité.* – Thèse de PhD, Institut National Polytechnique de Grenoble, septembre 1996.

[56] Maillet (É.) et Tron (C.). – On efficiently implementing global time for performance evaluation on multiprocessor systems. *Journal of Parallel and Distributed Computing*, vol. 28, n° 1, 1995, pp. 84–93.

[57] Malony (A.D.). – *Performance Observability.* – Urbana, Thèse de PhD, University of Illinois, septembre 1990.

[58] Malony (A.D.), Reed (D.A.) et Wijshoff (H.A.G.). – Performance measurement intrusion and perturbation analysis. *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, n° 4, juillet 1992, pp. 433–450.

[59] Marzullo (K.) et Neiger (G.). – Detection of global state predicates. *In : 5th Workshop on Distributed Algorithms*, éd. par Springer-Verlag, pp. 254–272.

[60] Mattern (F.). – Virtual time and global states of distributed systems. *In : Proceedings of the Workshop on Parallel and Distributed Algorithms.* – Bonas, France, septembre 1988.

[61] Mattern (F.). – On the Relativistic Structure of Logical Time in Distributed Systems. *BIGRE*, vol. Thème Systèmes et Environnements, n° 78, mars 1992, pp. 3–19.

[62] Mellor-Crummey (J.M.). – *Debugging and Analysis of Large-Scale Parallel Programs.* – Rapport technique n° 312, University of Rochester, septembre 1989.

[63] Message Passing Interface Forum. – MPI: A message passing interface standard. *International Journal of Supercomputer Applications*, vol. 8, n° 3/4, 1994.

[64] Miller (B.P.) et Choi (J.D.). – Breakpoints and halting in distributed systems. *In : Proceedings of International Conference On Distributed Computing Systems.* IEEE.

[65] Miller (B.P.), Gallaghan (M.D.), Cargille (J.M.), Hollingsworth (J.K.), Irvin (R.B.), Karavanic (K.L.), Kunchithapadam (K.) et Newhall (T.). – The Paradyn parallel performance measurement tool. *IEEE Computer*, vol. 28, n° 11, novembre 1995, pp. 37–46.

[66] Mueller (F.). – A library implementation of POSIX threads under UNIX. *In : Proc. of the Winter USENIX Conference*, pp. 29–41. – San Diego, CA, janvier 1993.

[67] Namyst (R.) et Méhaut (J.F.). – PM2: Parallel Multithreaded Machine. A Computing Environment for Distributed Architectures. *In : Parallel*

Computing : State-of-the Art Perspective (ParCo95), éd. par D'Hollander (E.H.), Joubert (G.R.), Peters (F.J.) et Trystram (D.). – North-Holland, 1996.

[68] Netzer (R.H.B.) et Miller (B.P.). – 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.

[69] Plateau (B.). – *Présentation d'APACHE*. – Rapport APACHE n° 1, Grenoble, IMAG, décembre 1994.

[70] Reed (D.A.), Aydt (R.A.), Madhyastha (T.M.), Noe (R.J.), Shields (K.A.) et Schwartz (B.W.). – *An Overview of the Pablo Performance Analysis Envrionment*. – Rapport technique, Department of Computer Science, University of Illinois, 1992.

[71] Roch (J.L.), Vermeerbergen (A.) et Villard (G.). – A new load-prediction scheme based on algorithmic cost functions. *In : CONPAR 94*, éd. par Verlag (Springer). – Linz - Austria, 1994.

[72] Roos (J.F.). – *Mise au point d'applications distribuées pour environnement de développement basé sur une technologie objet*. – Thèse de PhD, Université des Sciences et Technologies de Lille, février 1994.

[73] Saporta (G.). – *Probabilités, analyse de données et statistiques*. – 27 rue Ginoux, 75737 Paris Cedex 15, Éditions Technip, février 1990. ISBN 2-7108-0565-0.

[74] Stallman (R.) et Pesch (R.H.). – *Debugging with GDB: the GNU source-level debugger*. – 675 Mass Ave, Cambridge, MA 02139, USA, Tel: (617) 876-3296, USA, Free Software Foundation, janvier 1995, 4.12, for GDB version 4.14 édition, vi + 184p.

[75] Strassen (V.). – Gaussian Elimination is not Optimal. *Numerische Mathematik*, vol. 13, n° 4, 1969, pp. 354–356.

[76] Sunderam (V.S.). – PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, vol. 2, n° 4, décembre 1990, pp. 315–339.

[77] Teodorescu (F.) et Chassin de Kergommeaux (J.). – On correcting the intrusion of tracing non deterministic programs by software. *In : Proceedings of EuroPar'97*. – Springer Verlag.

[78] Tron (C.), Arrouye (Y.), Chassin de Kergommeaux (J.), Kitajima (J.P.), Maillet (É.), Plateau (B.) et Vincent (J.M.). – Performance Evaluation of Parallel Systems: the ALPES environment. *In : Proceedings of ParCo93*. – Elsevier Science Publishers.

[79] Wylie (B.J.N.) et Endo (A.). – *Design and realization of the Annai integrated parallel programming environment performance monitor and analyzer*. – Rapport technique n° CSCS-TR-94-07, Manno, Switzerland, Centro Svizzero di Calcolo Scientifico, août 1994.

Histoire du logo APACHE

Afin de présenter le projet APACHE au cours d'une journée ouverte le 15 décembre 1994, chaque groupe de travail devait préparer un bref résumé et quelques transparents. Il nous manquait alors un élément essentiel d'unité graphique. Au cours de discussions informelles, l'idée avait été formulée de réaliser une version stylisée et exploitable par LaTeX de la couverture de nos rapports de recherche. Nous avons pris l'avis professionnel de Patrizia pour l'aspect général que devrait avoir notre logo. En attendant des résultats d'expériences, j'ai dessiné une première ébauche au cours de la nuit du 18 novembre 1994. Après l'avoir montrée à des skieurs chevronnés (Christophe, Thierry, Yannick et Yves), je l'ai corrigée dans la nuit du 23 novembre. Ainsi est né un logo provisoire pour le projet APACHE. À la date de rédaction de ces lignes, il est toujours utilisé.