# Traçage flexible d'exécutions de programmes parallèles

Cyril Guilloud

▶ **To cite this version:**

Cyril Guilloud. Traçage flexible d'exécutions de programmes parallèles. Réseaux et télécommunications [cs.NI]. Institut National Polytechnique de Grenoble - INPG, 2004. Français. NNT : . tel-00010555

**HAL Id: tel-00010555**
**https://theses.hal.science/tel-00010555**

Submitted on 12 Oct 2005

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

***INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE***

***N°attribué par la bibliothèque***
/_/_/_/_/_/_/_/_/_/_/

***THÈSE***

pour obtenir le grade de

**DOCTEUR DE L'INPG**

***Spécialité : "Informatique, Systèmes et Logiciels"***

préparée au laboratoire Informatique et Distribution
dans le cadre de ***l'École Doctorale "Mathématiques, Sciences et Technologies de l'Information, Informatique"***

présentée et soutenue publiquement

par

Cyril Guilloud

Le 5 février 2004

***Titre :***

# Traçage flexible d'exécutions de programmes parallèles

---

***Directeur de thèse :*** Brigitte Plateau
***Co-directeur :*** Jacques Chassin de Kergommeaux

---

**JURY**

| | |
|---|---|
| M. Andrzej Duda | Président |
| M. José C. Cunha | Rapporteur |
| M. Bertil Folliot | Rapporteur |
| M$^{me}$ Brigitte Plateau | Examinatrice |
| M. Jacques Chassin de Kergommeaux | Examinateur |
| M$^{me}$ Pascale Rossé-Laurent | Examinatrice |

*à mon père.*

Mes premiers remerciements vont naturellement aux membres du jury qui ont bien voulu examiner ce travail. Je suis très reconnaissant à Andrzej Duda d'avoir accepté de présider le jury, à José Cuhna et Bertil Folliot pour leurs commentaires avisés sur ce manuscrit et à Pascale Rossé pour avoir suivi ce travail au cours des trois dernières années. Je dois aussi de particuliers remerciement à Brigitte Plateau pour m'avoir permis d'effectuer cette thèse et pour son encadrement lors de mon DEA.

Jacques Chassin de Kergommeaux à été un directeur de thèse particulièrement attentif à tous les aspects techniques et rédactionnels de cette thèse. Ses conseils toujours pertinents (bien que je le les ai pas toujours suivis...), sa disponibilité et sa patience pour les nombreuses relectures du manuscrit ont permis de mener à bien cette thèse.

De nombreux aspects techniques du travail présenté ici doivent beaucoup à Jacques Briat, je l'en remercie vivement. Je tiens à saluer également : tous les membres du laboratoire ID, Philippe Waille, Jean-Marc, Thierry, Philippe Augerat, Jean-Louis, Denis et Denis, Bruno, Olivier, Guillaume, avec une mention spéciale pour Grégory dont la disponibilité et la capacité à résoudre tous les problèmes sont un atout précieux ; tous les thésards du laboratoire, PF, Georges, Bruno, Remi, Cyrille, Anne, Corine, Florence, Emmanuel(s), Mauricio, Jésus, Renaud, Gustavo et tous les autres ; Łukasz pour les longues discussions politiques et religieuses qui ont pimentées ces trois ans ; les stagiaires que j'ai encadré avec beaucoup de plaisir (Xiaoyan, Vincent, Alaa) ; les assistantes Hélène, Marion et Anne-Laure pour leur talent à vaincre les imbroglios administratifs relatifs aux diverses missions effectuées.

L'éradication d'un très grand nombre de fautes d'orthographe, de grammaire et de français est à porter au crédit des relectures attentives de Myriam, André et Aline.

Cette thèse a été un long parcours, et cette dernière année particulièrement éprouvante, c'est pourquoi je tiens à remercier très chaleureusement toutes les personnes qui m'ont aidées plus ou moins directement sur un plan plus personnel que technique à en arriver au terme : tout d'abord ma famille et nos amis pour leur présence et leur soutient et que je ne pourrait suffisamment remercier ici ; et pour finir, un grand merci à tous mes amis : tout d'abord à Francis, qui, malgré ses goûts musicaux et cinématographiques particulièrement éclectiques fut un colocataire des plus agréables et puis à tous les autres, Cathy, Grégoire, Magali, Pierre (Bontron et Neyron), Aline, Manu, Erwan, Marion, tous les Oliviers (Guyotot, Cinquin, Mansiot et Maury), Matthieu, Margueritte, Karim, Carole, Remi, Jean-Gilles, Tanguy, Pauline, Romain, Sophie, Sylvia, Jean-Christophe, Magali, Éric (et Bastien).

# Table des matières

# Table des figures

# Liste des tableaux

Chapitre

# Introduction

I

*« Au commencement, il n'y avait rien. Et Dieu dit : "Que la lumière soit !"*
*Alors, il n'y avait toujours rien, mais tout le monde le voyait. »*
Dave Thomas

## I.1 Le parallélisme

Le parallélisme en informatique est depuis longtemps un centre d'intérêt majeur, à l'intersection d'un grand nombre de disciplines de l'informatique : ordonnancement, réseaux de communication, tolérance aux pannes, évaluation de performances ou encore simulation. Le principe de base du calcul parallèle est de mettre en commun plusieurs unités de traitement pour accélérer la vitesse d'exécution de programmes. Le but recherché est non seulement de pouvoir effectuer plus vite des calculs mais aussi de pouvoir traiter des problèmes toujours plus gros afin d'assouvir les besoins de nombreuses applications, allant de la météorologie à la mécanique quantique en passant par la simulation physique.

Si l'idée du parallélisme est simple, sa mise en œuvre reste extrêmement complexe. Les contraintes matérielles liées notamment aux limitations des réseaux de communication — latence et bande passante — entraînent des problèmes considérables de conception des applications. Cela empêche l'utilisation optimale de la puissance de calcul disponible. La conception de telles applications parallèles est freinée notamment en raison de la complexité de leur architecture, du manque d'environnements de développement et de mise au point, et des rapides évolutions des outils et des concepts utilisés. Il est donc fondamental de concevoir des techniques efficaces et flexibles de développement et de mise au point d'applications parallèles, tant pour la correction que pour les performances.

## I.2 Le laboratoire Informatique et Distribution (ID-IMAG)

Le domaine scientifique abordé par le laboratoire Informatique et Distribution[1] (ID-IMAG) [ID] est celui du calcul parallèle et distribué pour les hautes performances

[1] Sous les tutelles de l'INPG, du CNRS, de l'INRIA et de l'UJF.

ainsi que de la distribution des charges et des ressources. Les travaux qui y sont effectués sont de nature théorique dans certains domaines fondamentaux, de nature conceptuelle et expérimentale, ou encore de nature transversale dans les projets.

Les aspects théoriques des travaux du laboratoire sont principalement orientés vers le développement de techniques d'ordonnancement, d'optimisation combinatoire et de modélisation de systèmes parallèles. L'ordonnancement de tâches est primordial pour l'équilibrage de charge entre les ressources disponibles. La modélisation de systèmes parallèles a pour but l'évaluation de leurs performances. Ces approches théoriques sont également mises en œuvre dans des logiciels et intergiciels permettant de faciliter la programmation d'applications parallèles et l'exploitation de plateformes pour le calcul hautes performances.

L'évolution amorcée depuis quelques années est à la démocratisation des grappes (*cluster*) de calcul basées sur des stations de travail, des grilles de grappes (*grids*) et des réseaux de calcul pair à pair (*peer to peer computing*). Toutes ces architectures offrent des capacités de calcul extrêmement intéressantes mais elles sont complexes à exploiter efficacement, principalement en raison des grandes latences lors de communications entre nœuds d'exécution. L'exploitation efficace de telles ressources de calcul est l'un des objectifs du laboratoire ID-IMAG. La voie choisie pour atteindre ce résultat est de privilégier l'*aide* à la conception d'applications parallèles portables et efficaces. Cette aide passe notamment par la création d'intergiciels pour gérer les communications ou la multiprogrammation, et la création de langages de programmation parallèles pour faciliter la conception d'applications de plus haut niveau. C'est dans ce but qu'a été créé ATHAPASCAN [Cav99, Dor99, RCDG], un langage de programmation parallèle. Il offre à l'utilisateur une interface de programmation lui permettant de se concentrer sur les aspects algorithmiques.

Afin de tirer pleinement parti de la puissance des machines parallèles, le *débogage pour les performances*, c'est-à-dire la recherche et l'élimination des problèmes entraînant des ralentissements, est une phase capitale du cycle de développement des programmes parallèles. Notre approche du débogage pour les performances se base sur l'*observation comportementale*, c'est-à-dire la représentation de l'ensemble des actions effectuées par un programme afin d'aider son concepteur à en comprendre le déroulement, à identifier les comportements susceptibles d'entraîner des problèmes de performance et à les corriger. Pour représenter graphiquement le fonctionnement de programmes parallèles sur un grand nombre de nœuds d'exécution, PAJÉ [dOS99, CdKdOS03], un outil de visualisation d'exécution de programmes parallèles, a été développé. Une caractéristique fondamentale de PAJÉ est de pouvoir être adapté facilement pour représenter tous les modèles de programmation utilisés pour les développements du laboratoire ID-IMAG.

## I.3 Contribution

Cette thèse s'inscrit dans le cadre de l'activité consacrée au débogage pour les performances au laboratoire ID-IMAG. Le principal aspect auquel nous nous sommes attachés est la flexibilité de la collecte d'informations pour l'observation comportementale. L'objectif est la réalisation d'un outil de traçage d'applications parallèles flexible dans le but d'obtenir, avec PAJÉ, un système adaptable à la plupart des situations où l'observation du déroulement d'un programme est utile pour le débogage pour les performances. Ce système doit permettre l'obtention d'informations issues des différentes couches logicielles utilisées pour la programmation d'applications parallèles.

La conception d'un système flexible de traçage se heurte à des contraintes inhérentes à l'observation et à la multiplicité des modèles de programmation. L'observation doit induire le minimum de perturbations possible afin de ne pas fausser les observations. Il faut donc utiliser les mécanismes les plus efficaces possibles et apporter une attention particulière à la gestion des données collectées. Un outil de traçage est étroitement lié aux langages et modèles de programmation utilisés pour la conception du programme observé. Afin de limiter cette dépendance au cours du cycle d'observation, nous avons choisi de nous orienter vers une séparation de format de trace et des formats utilisés pour l'exploitation des traces.

Le travail présenté dans ce document concerne donc la conception et la réalisation d'un système de *traçage flexible d'exécution de programmes parallèles*. La flexibilité est la capacité pour un système à pouvoir être utilisé dans diverses situations. Dans notre cas, cette caractéristique passe par une adaptation su système de traçage à différents programmes et modèles de programmation. Nous présentons les principales techniques d'observation, de collecte d'informations et de représentation ainsi que les moyens mis en œuvre pour concevoir un système de collecte et de manipulation de traces flexible. Ces travaux ont été présentés lors de la 9ème conférence internationale Euro-Par [CdKGdOS03]. Des travaux préliminaires sur la surveillance de grappes ont fait l'objet d'une publication[GCdKdOSA01] aux 13ème rencontres francophones du parallélisme.

Cette thèse a été initiée dans le cadre du projet LIPS (LInux Parallel Solution) du Groupement d'Intérêt Économique (GIE) DYADE[2], une collaboration entre la société Bull [Bul03] et l'INRIA Rhône-Alpes [INR]. Un des objectifs de cette collaboration est le développement de logiciels libres pour l'exploitation de grappes de PC sous LINUX.

Je voudrai également souligner que ce travail de thèse m'a permis d'encadrer

[2]Le GIE DYADE est aujourd'hui dissous mais la collaboration entre l'INRIA et Bull perdure au sein de l'action LIPS.

des stages d'étudiants avec qui la collaboration à été particulièrement agréable et appréciable. Mlle Jingyi Lu [Lu02] à participé à l'implantation du traceur et l'instrumentation d'Inuktitut. Vincent Lethuillier [Let01] et Alâa Kalaî [Kal03] ont travaillé sur la correction des dérives d'horloges sur grappe de PC. Emmanuel Grange [Gra01] a étudié la surveillance système des grappes de PC.

## I.4 Organisation de ce document

Après cette brève introduction au contexte de la réalisation de nos travaux, le **chapitre II** présente les notions de base de la mise au point des programmes parallèles : le débogage pour les performances. Nous présentons plus en détails le contexte technique dans lequel ce travail se positionne, principalement les plateformes matérielles et logicielles utilisées et développées au laboratoire ID-Imag. Les approches pour l'exploitation d'architectures parallèles et les raisons qui nous conduisent à recourir à l'observation d'exécutions de programmes parallèles sont exposées.

Le **chapitre III** décrit les techniques usuelles pour acquérir des informations au cours de l'exécution de programmes parallèles : la surveillance, le débogage interactif, les indicateurs statistiques, le profilage et l'observation comportementale. Nous expliquons les principes, les caractéristiques et les domaines d'application des différentes approches. Nous abordons également les moyens de réaliser la collecte des informations nécessaires à l'observation ainsi que les méthodes d'instrumentation.

Après avoir collecté des informations sous une forme brute, il est nécessaire d'en fournir une représentation utilisable par l'utilisateur. Nous présentons au **chapitre IV** le rôle joué par les outils de représentation et les critères de qualité des diverses techniques permettant l'interprétation des informations enregistrées. Il s'agit principalement de leurs capacités à présenter les données à l'utilisateur de manière claire et pertinente afin de déterminer l'origine des problèmes de conception des programmes entraînant des manques de performance.

Nous présentons au **chapitre V** quelques outils d'observation existants. Nous nous attachons à présenter les approches utilisées pour l'observation. Une attention particulière sera portée à Pajé, outil de visualisation générique utilisé au sein du laboratoire ID-Imag.

Le **chapitre VI** présente notre approche pour favoriser l'adaptabilité d'outils de traçage et la manière dont nous l'avons mise en œuvre dans un outil de traçage adaptable : Tumit. Nous décrivons également les techniques utilisées dans l'implantation de Tumit ainsi qu'un exemple concret d'utilisation pour le traçage de la bibliothèque Taktuk.

Afin de valider l'approche utilisée pour assurer la flexibilité du traçage et illustrer l'observation d'une bibliothèque, nous présentons au **chapitre VII** un exemple

complet d'observation de l'exécution de programmes basés sur ATHAPASCAN. Nous présentons dans ce chapitre les principaux enseignements retirés de ces expérimentations.

Le **chapitre VIII** permettra enfin de conclure à partir des travaux effectués, de présenter brièvement les développements en cours et de proposer les directions de travail qui nous semblent prometteuses.

Chapitre

# Mise au point de programmes parallèles

II

Le développement de programmes parallèles corrects et efficaces est une tâche extrêmement ardue. Les principales recherches dans ce domaine portent sur la parallélisation automatique et l'aide à la conception de programmes à parallélisme explicite. Cette aide consiste le plus souvent à offrir des outils de développement efficaces aux programmeurs afin de simplifier leur tâche et de réduire les temps de développement. Ces outils peuvent être des débogueurs [Tot], des bibliothèques de programmation [mpi, Pel01] ou encore des outils d'observation [gpr, NAW+96].

Ce chapitre présente le contexte technologique dans lequel se place notre travail tant du point de vue matériel que logiciel. Nous présentons brièvement les raisons conduisant à l'utilisation de grappes de PC et les contraintes de développement qui y sont rattachées. La phase du développement sur laquelle nous allons nous focaliser est celle de l'optimisation des performances : le *débogage pour les performances* (*performance debugging*).

## II.1 Intérêts et contraintes du parallélisme

Le parallélisme en informatique consiste à faire travailler en coopération plusieurs unités de calcul. Le but est d'offrir une plus grande capacité de travail. Cet objectif peut être atteint soit par la réduction du temps de calcul pour un problème, soit par l'augmentation de la taille des données d'un problème traitable en un temps donné.

L'utilisation combinée d'un grand nombre de machines, donc de leurs capacités de calcul (processeur) et de stockage (mémoire vive et disque dur), permet de traiter plus rapidement des problèmes de grande taille. Les problèmes visés sont, par exemple, le traitement de grandes quantités de données issues d'observations dans des domaines aussi variés que la physique ou la génomique. La programmation parallèle permet également une programmation mieux structurée. Les programmes à base de *threads* sont, par exemple, bien adaptés aux applications interactives : un *thread* traite les actions d'un utilisateur tandis qu'un autre effectue des calculs ou se charge de l'affichage. On retrouve les mêmes avantages dans la création d'applications distribuées : l'assemblage de modules simples communiquant par envoi de

messages permet de concevoir des applications réparties de grande ampleur sans concentrer les problèmes de conception sur un programme central.

Malheureusement le développement d'applications parallèles correctes et efficaces se heurte à de nombreux obstacles. Avec une diffusion croissante des systèmes parallèles (ordinateurs multiprocesseurs ou grappes de stations de travail) et l'apparition de parallélisme à l'intérieur même des processeurs (*Hyperthreading* [Hyp03]), il est indispensable de disposer d'outils permettant la conception de programmes exploitant la puissance de calcul disponible.

Le parallélisme ne se justifie que si les bénéfices, gains d'argent et de temps, générés par l'utilisation de machines parallèles sont supérieurs aux problèmes soulevés par la mise en œuvre des logiciels et des matériels utilisés. Il faut donc que la mise au point de programmes parallèles repose sur des machines compétitives et des techniques de conception aussi simples que possible.

## II.2 Les grappes de PC

Concevoir un ordinateur multiprocesseur est une tâche particulièrement ardue. Il est en effet très complexe d'assurer le bon fonctionnement de plusieurs unités de traitement collaborant tout en gardant un niveau de performance acceptable. Ces difficultés proviennent pour l'essentiel des synchronisations entre flots d'exécution et du maintien de cohérence entre les données réparties. Le développement de machines multiprocesseurs est ainsi une opération non seulement complexe, mais également coûteuse en raison des matériels spécifiques utilisés. Les *grappes* (*clusters*) de machines sont apparues afin de limiter la complexité et le coût des plateformes de calcul. L'idée de simplifier les architectures en assemblant des machines indépendantes existe depuis de nombreuses années [vax83]. Le principe est de relier par un réseau haut débit des machines standard non conçues pour fonctionner en collaboration afin d'obtenir un rapport puissance de calcul/prix très compétitif.

Bien que les ordinateurs vectoriels de grande taille conservent une supériorité en terme de puissance de calcul absolue (en terme de nombre d'opérations par seconde)[1], on constate une forte progression de l'utilisation de grappes. La principale caractéristique d'une grappe est de ne pas disposer de *mémoire partagée* (*shared memory*). Il faut donc élaborer des programmes communiquant par *envoi de messages* (*message passing*) entre les nœuds de calcul plutôt que par partage de zones de mémoire.

La puissance sans cesse croissante des stations de travail stimulée par la grande diffusion favorise également l'apparition de grappes de *PC standard* (*COTS : compu-*

[1] La machine *Earth Simulator* [Ear], actuellement la plus puissante au monde, est un ordinateur vectoriel hautement parallèle.

*ter off the shelf*) dont le rapport performance/prix est particulièrement intéressant. Il s'agit de machines du commerce (par opposition aux calculateurs dédiés ou aux stations de travail haut de gamme) reliées par un *réseau d'interconnexion* que l'on trouve classiquement dans les réseaux d'entreprise et donc bon marché.

Dans cette optique, le laboratoire ID a mis en œuvre, en collaboration avec l'équipe Hp-labs de Grenoble, le i-Cluster [RAM+01] [Wap02] (voir figure II.1), une grappe de 225 machines. Cette grappe est constituée de simples PC conçus pour la bureautique reliés par un réseau « Ethernet » classique (100 Mb/s) à des commutateurs (*switch*) du marché. Les commutateurs sont reliés entre eux par un réseau « Gigabit-Ethernet ».

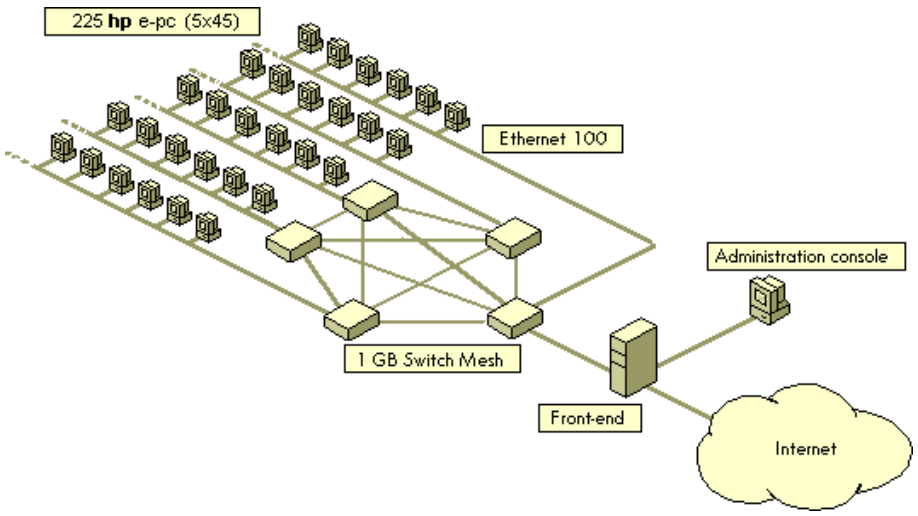



**Figure** II.1 – Le i-Cluster

Le i-Cluster mis en œuvre par le laboratoire ID et l'équipe hp-labs de Grenoble est constitué de 225 PC pentium III (256 Mo de mémoire et 15 Go de disque) interconnectés sur un réseau Ethernet 100Mb/s et reliés à un réseau gigabit Ethernet par 5 commutateurs.

L'utilisation de grappes entraîne certains choix techniques. Le fait de disposer de plusieurs *niveaux de parallélisme* a par exemple induit le développement de *modèles de programmation hybrides*. Le niveau de parallélisme correspondant à un nœud de calcul (mono ou multiprocesseur) se base sur l'utilisation de *threads* tandis que le parallélisme entre machines implique l'utilisation de bibliothèques de communication par envoi de messages comme MPI [GLDS96, GL96]. L'utilisation de plusieurs niveaux de parallélisme permet d'optimiser les performances notamment en réduisant l'inactivité des processeurs par le recouvrement des temps d'inactivité dus aux communications par des calculs [Ber97].

Le principal problème de telles grappes réside dans l'exploitation de la puissance disponible. Si la capacité de calcul brute est importante (165 Gflop/s théoriques pour les 225 PC à 733 MHz du i-Cluster), il est très compliqué d'en tirer efficacement

parti. Par exemple, avec le i-Cluster, seulement 81,6 Gflop/s ont pu être obtenus lors des tests avec le programme LINPACK [lin] servant de référence pour le classement du Top500 [Topb]. Ce résultat, qui peut être considéré comme satisfaisant, a été obtenu après de nombreux essais d'optimisation de la topologie du réseau [RAM+01]. Le manque de rendement des grappes provient essentiellement des faibles performances du réseau d'interconnexion par rapport à celles des processeurs. L'utilisation de réseaux d'interconnexion haut débit et à faible *latence* (le temps d'envoi d'un message de taille nulle) de type Myrinet [Myr] permet de réduire l'impact des communications sur les performances des programmes exécutés sur une grappe. Malgré cela, la conception de programmes adaptés à de telles architectures reste indispensable. Dans cet objectif, nous considérons que l'aide offerte aux programmeurs, par les logiciels d'observation, pour la mise au point d'applications exploitant ces grappes est primordiale. Elle doit être adaptée et prendre en compte les particularités de telles architectures. Il faut, par exemple, être capable de fournir des observations issues des différents niveaux d'abstraction utilisés dans la conception de l'application parallèle cible.

De nouvelles sources de puissance de calcul sont également en train d'émerger. Les *grilles de grappes* sont des interconnexions, par des réseaux haut débit, de grappes séparées géographiquement. Les *jachères de calcul* sont des grappes constituées par les parcs de machines de bureautique des grandes entreprises (des milliers d'ordinateurs standard). Le « *global-computing* » [Xtr] consiste à utiliser des machines reliées par le réseau Internet. Pour l'exploitation de ces architectures, dont la latence de communication est bien plus grande que celle d'un ordinateur multiprocesseurs *(SMP : Symetric Multi Processor)* ou d'une grappe, l'importance de la mise au point de programmes parallèles privilégiant les calculs par rapport aux communications est encore plus importante. Pour le « global-computing », seules des applications trivialement parallèles telles Seti@HOME [SET] peuvent pour l'instant véritablement profiter de cette puissance de calcul.

Le laboratoire ID-IMAG a pour objectif, entre autres, le développement d'applications et de bibliothèques permettant d'aider à exploiter le plus efficacement possible ces architectures de calcul.

## II.3 Utilisation efficace de grappes

Les domaines de recherche du laboratoire sont guidés par l'évolution du contexte technologique des architectures matérielles de calcul parallèle. La multiplication des grappes de stations de travail et l'émergence de grilles de grappes conduisent à orienter le développement d'outils logiciels pour faciliter l'utilisation efficace d'architectures avec des temps de latence de communication importants.

De nombreux travaux ont pour objectif la mise au point de méthodes pour l'au-

tomatisation de la parallélisation d'applications. Cet aboutissement qui serait particulièrement intéressant s'il était atteint pour des applications générales, reste hors de portée des techniques actuelles. Il n'est appliqué que dans des cas de figure précis et bien connus comme, par exemple, la parallélisation de nids de boucles.

Partant de ces constatations, les équipes du laboratoire ont décidé d'orienter leurs travaux vers la conception d'outils *facilitant la programmation parallèle*. Ces outils sont des langages de programmation d'applications parallèles de haut niveau (ATHAPASCAN [RGR03, GRCD98]), des *intergiciels* pour favoriser la portabilité et factoriser les fonctions de base de l'algorithmique parallèle (INUKTITUT [BGM03]), des programmes de lancement d'applications réparties (TAKTUK [MR01, Mar03] ou encore des outils d'administration (Ka-admin [AMS02]).

ATHAPASCAN [Dor99, RGR03] est un langage de programmation de haut niveau. Il implémente un modèle de programmation basé sur un mécanisme de création de tâches communiquant entre elles par une mémoire partagée distribuée simulée. ATHAPASCAN est conçu comme une bibliothèque C++ permettant de décrire le parallélisme des applications indépendamment de la machine cible. Il permet d'abstraire la machine parallèle sur laquelle s'exécute un programme ATHAPASCAN et de séparer l'algorithmique du programme de l'ordonnancement des tâches. L'ordonnancement des tâches sur les nœuds de calcul est effectué automatiquement [CDG+]. Pour assurer une portabilité sur différentes plateformes, des grappes de PC aux machines multiprocesseurs, ATHAPASCAN est construit au-dessus d'une couche de portabilité nommée INUKTITUT [BGM03] (anciennement ATHAPASCAN-0 [Car99, BGPP97, CBR95]). Nous reviendrons sur une description un peu plus fine de ATHAPASCAN au cours du chapitre VII.

INUKTITUT offre une interface portable de bas niveau pour l'utilisation de bibliothèques de *threads* et de communication pouvant utiliser différents protocoles réseaux (*rsh* ou CORBA par exemple). Une des caractéristiques d'INUKTITUT est de fournir des mécanismes de communication par *message actif*, c'est-à-dire qu'à l'inverse des communications MPI, il n'est pas nécessaire de prévoir *a priori* la réception sur le nœud destinataire. TAKTUK [MR01, MB02], un des composants de INUKTITUT, est un module de lancement d'applications parallèles. Il permet de lancer de manière très efficace un programme parallèle sur un grand nombre de nœuds. Pour cela, TAKTUK établit un réseau de communication pouvant s'adapter à la topologie physique du réseau et se baser sur différents protocoles réseaux (CORBA, TCP, ...). TAKTUK peut donc également être vu comme une bibliothèque de communication.

Ces deux outils, ATHAPASCAN et INUKTITUT, constituent des *couches logicielles* (voir figure II.2) sur lesquelles il est confortable de s'appuyer pour la programmation d'applications de plus haut niveau telles que SAPPE [ZFV02] ou Takakow [Tak]. En plus de la portabilité, l'utilisation de ces couches logicielles apporte une simplicité de programmation par une forte réutilisation du code existant.

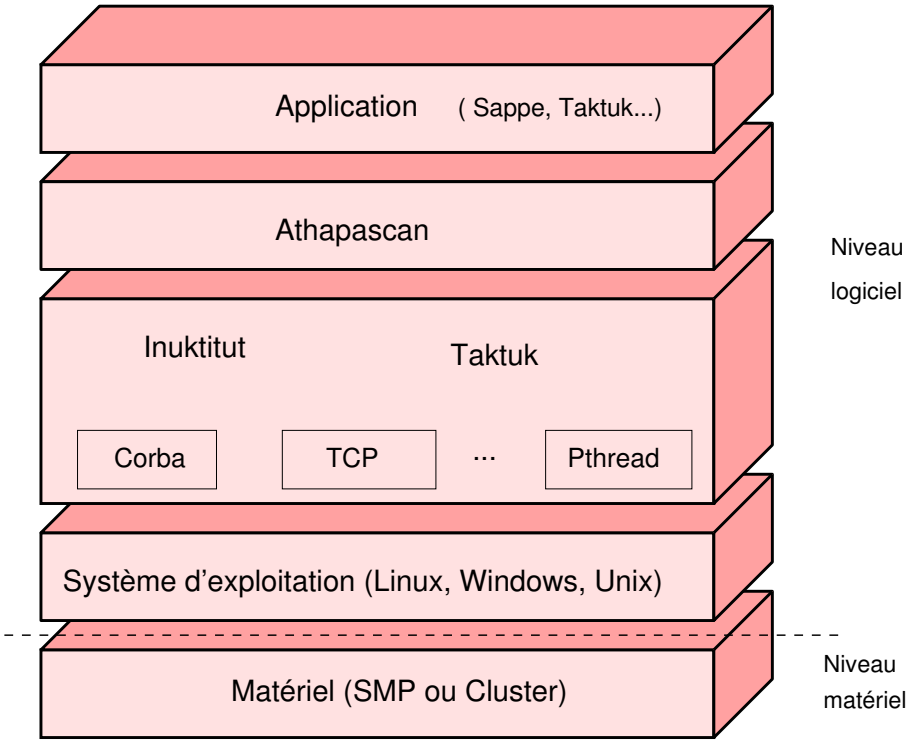



**Figure** II.2 – Les différentes couches logicielles

## II.4 Débogage des logiciels

L'utilisation de grappes de machines implique une augmentation de la latence pour les communications entre nœuds par rapport à des machines disposant de mémoire partagée. La simplification des machines, par l'utilisation de grappes, a donc pour contrepartie l'augmentation de la complexité des programmes parallèles s'exécutant sur elles.

Il est non seulement difficile de concevoir des programmes parallèles, mais il est également complexe d'assurer qu'ils tirent efficacement parti des architectures parallèles sur lesquelles ils sont exécutés. Les deux critères fondamentaux de qualité d'un programme parallèle sont la *correction* et la *performance*. La correction est la capacité du programme à fournir des résultats respectant un critère d'exactitude fixé par le programmeur. La performance est la rapidité avec laquelle ces résultats sont produits. Pour les programmes parallèles, une notion fondamentale à laquelle est liée la performance est celle de *facteur d'accélération* (*speed-up*), c'est-à-dire l'augmentation de performance en fonction du nombre de sites d'exécution. La « *scalabilité* »[2]

[2]La *Scalabilité* est un anglicisme dont la traduction serait : « aptitude au passage à l'échelle ».

(*scalability*) d'une application définit sa capacité à pouvoir exploiter efficacement un grand nombre de ressources matérielles. Une application *parfaitement scalable* (*perfectly scalable*) aura une accélération linéaire en fonction du nombre de nœuds de calcul sur lesquels elle s'exécute.

L'accélération potentielle d'une application en fonction du nombre de nœuds de calcul alloués est intimement liée aux contentions sur les données et à la saturation des interfaces de communication. Cette capacité d'accélération est extrêmement complexe à évaluer sans expérimentation car les sources potentielles de problèmes s'y opposant sont nombreuses et complexes à identifier. La phase de « mise au point pour les performances » (*performance debugging*) d'un programme parallèle est donc en partie consacrée à l'amélioration de cette caractéristique.

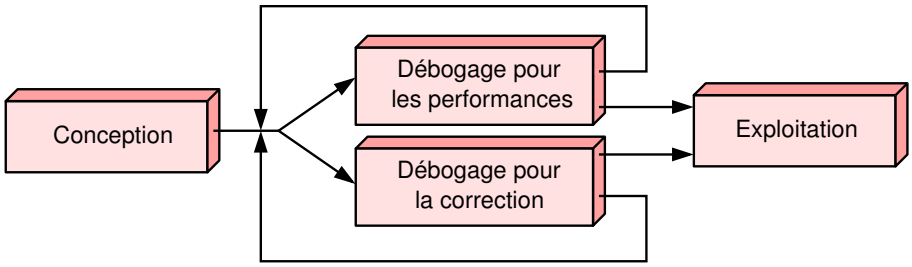



**Figure** II.3 – Cycle de mise au point d'un programme

La mise au point d'un programme parallèle consiste en une succession de phases de débogage pour la correction et de phases d'optimisation des performances du programme.

Il est à noter que ces deux formes de débogage peuvent interférer : la suppression d'une bogue de correction peut engendrer de nouvelles bogues de performance. Il faut donc porter une attention particulière à toute modification du code (par exemple en établissant un ensemble de tests de non régression).

### II.4.1 Débogage pour la correction

Le premier critère de qualité en programmation est la conception d'applications effectuant correctement le travail exigé. Bien que des résultats non exacts mais approchants soient parfois suffisants, il faut s'assurer que leur calcul respecte des contraintes précises. Si ce but est parfois difficile à atteindre dans un programme séquentiel, il le devient considérablement plus dans le cadre de la programmation parallèle. En effet, non seulement la difficulté de comprendre les interactions entre plusieurs flots d'exécution est importante, mais en plus il existe des fluctuations

entre deux exécutions d'un même programme[3].

Pour comprendre le fonctionnement de programmes séquentiels et en éliminer les erreurs, il existe des *débogueurs* [gdb]. De la même manière, des débogueurs parallèles ont été conçus [Tot]. Ils permettent d'avoir une vue très fine et précise d'un instant donné de l'exécution d'un programme et d'avancer dans le déroulement du programme instruction par instruction (ou par bloc d'instructions). On peut ainsi exécuter progressivement le programme en ayant accès aux valeurs prises par chaque variable afin d'identifier précisément l'origine des bogues.

Il est à noter qu'il existe des techniques pour vérifier la correction d'un programme sans recourir à son exécution. Il s'agit de méthodes de preuve de programmes par l'analyse du code source. Ces méthodes encore expérimentales pour les programmes séquentiels ou critiques sont marginales dans le domaine des programmes parallèles pour le calcul hautes performances.

Nous allons nous intéresser dans ce document plus particulièrement au débogage pour les performances. Toutefois les techniques utilisées peuvent également avantageusement aider à la compréhension du déroulement d'un programme parallèle et ainsi contribuer à l'éradication de bogues de correction.

## II.4.2 Débogage pour les performances

Après s'être assuré de la correction d'un programme parallèle, le problème essentiel est d'en optimiser les performances. Pour mener à bien ce débogage pour les performances, on procède classiquement de la manière suivante (voir figure II.4) : on commence par estimer les performances théoriques de l'application, puis on mesure les performances de l'application cible et finalement on compare les résultats. Si un écart flagrant ou inexpliqué est constaté, il faut identifier pourquoi et dans quelle partie du code se trouvent les problèmes de performance à corriger. Une fois la correction effectuée, il faut à nouveau évaluer les performances de l'application pour vérifier que la correction est appropriée.

### II.4.2.1 Estimation des performances

L'estimation des performances théoriques peut se faire empiriquement par l'évaluation du nombre d'actions élémentaires nécessaires à l'achèvement d'un calcul, bien que cette manière de procéder soit peu fiable et parfois trop complexe pour des problèmes fortement irréguliers. Une approche plus formelle est la *modélisation* qui consiste à concevoir un modèle mathématique du programme observé afin de prévoir les états qu'il peut atteindre et le temps mis pour y parvenir. Pour traiter un modèle, il est possible de procéder à une *simulation* ou à une *résolution analytique* [Fer98]

[3]Pour supprimer cette imprévisibilité, il est nécessaire de faire appel à des mécanismes de réexécution déterministe [RDBC+03] [Ray94]

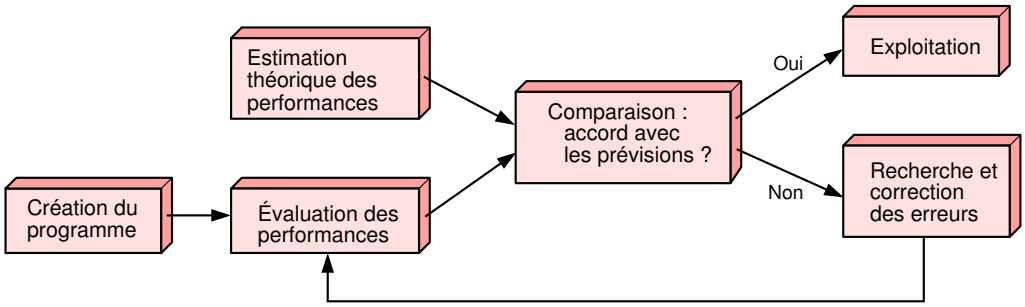



**Figure** II.4 – Cycle de débogage pour les performances

[Ben03] du modèle. La simulation va imiter une exécution, tandis que la résolution consiste à calculer mathématiquement un état du modèle. Ces techniques basées sur une formalisation du système observé ne sont cependant pas toujours facilement applicables.

#### II.4.2.2 Comparaison des performances

Un point important est de déterminer si l'application cible doit être ou non optimisée par rapport à la plateforme utilisée. Il est en effet illusoire de vouloir optimiser une application si les limitations constatées sont dues au matériel utilisé. Bien que les estimations *a priori* des performances soient possibles, en pratique, les développeurs sont souvent amenés à faire ces estimations de manière empirique. Une des solutions empiriques simple pour estimer les performances susceptibles d'être atteintes sur une plateforme par une application typique est proposée dans [Gro95]. Il s'agit de comparer les performances (en terme de nombre d'opérations par seconde) avec une application de référence. Une application adéquate pour ce type de comparaison peut être la suite de tests LINPACK . En effet ces tests sont bien connus et optimisés pour un grand nombre de plateformes. Si les performances de l'application étudiée n'arrivent pas à un certain pourcentage (75% par exemple) des performances de LINPACK sur la même plateforme, il est judicieux d'en examiner les raisons. Si l'on obtient des performances comparables, il est vraisemblable que les limitations proviennent de la machine. Bien entendu, ces considérations sont à prendre avec précaution et à relativiser en fonction du *taux de parallélisme* de l'application. Le taux de parallélisme d'un programme est le rapport entre le temps total d'exécution en parallèle et le temps total d'exécution. Plus ce nombre est proche de 1, plus le programme pourra tirer avantage d'un nombre élevé de processeurs. Ce taux de parallélisme est fortement dépendant de la structure du programme et des actions pouvant être effectuées indépendamment les unes des autres.

### II.4.2.3 Recherche et correction de problèmes de performance

Une fois le manque de performance constaté, il faut déterminer les conditions d'apparition des problèmes. Dans [Ma95], Miller et al. proposent le modèle de recherche de problèmes de performance $W^3$ ($W^3$ *search model : Why, Where, When*) pour identifier de manière pertinente les problèmes de performance. Il s'agit de trouver :

- *pourquoi* l'application n'est pas aussi rapide qu'espéré (*Why*). Cela peut être dû par exemple à des problèmes de synchronisation, à des accès concurrents sur des données ou encore à des goulots d'étranglement au niveau du réseau ;
- *où* se situe le problème (*Where*). Il s'agit d'identifier la zone du code générant un problème et quels objets, éventuellement abstraits, peuvent être impliqués (*thread*, nœud de calcul etc.) dans l'apparition du problème ;
- *quand* apparaît le problème (*When*). Le programme pouvant avoir de nombreuses phases distinctes d'exécution, il est important d'isoler précisément le moment où apparaissent les problèmes pour pouvoir les identifier et les corriger de manière ciblée.

En répondant à ces questions, il est possible de comprendre l'origine d'un problème et de proposer une correction du programme parmi des techniques de programmation usuelles pour pallier les manques de performance :

- multiprogrammation : l'utilisation de plusieurs flots d'exécution par un programme sur un même nœud de calcul permet de recouvrir les blocages liés à des communications ou à des accès disque par l'activité d'un autre *thread* ;
- migration de processus : si deux processus interagissent fortement avec d'autres nœuds, il est possible de saturer l'interface réseau du nœud. Il serait donc intéressant de remplacer l'un d'entre eux par un processus effectuant des calculs de manière autonome ;
- pré-chargement de données (*pre-fetching*) : il est parfois possible d'anticiper l'utilisation de données et de grouper des transferts pour limiter les surcoûts associés au rapatriement d'une faible quantité de données.

Pour tenter de répondre aux questions précédentes, la technique la plus simple utilisée par les programmeurs est le chronométrage de chaque phase du programme. On en déduit ainsi empiriquement les portions de code à optimiser et les comportements des algorithmes à modifier. Malheureusement, cette simple analyse ne peut convenir dans tous les cas. L'identification de problèmes de performance dans des phases d'activité efficaces et inefficaces entremêlées n'est pas possible avec une analyse aussi globale.

Pour améliorer le diagnostic sur les problèmes de performance d'une application parallèle, il est courant d'utiliser des outils d'analyse de performance [MCLD01] ou d'observation tels que des profileurs [gpr] [Mal] ou des traceurs [PAL] afin d'obtenir des informations utiles pour l'identification et la compréhension de phénomènes

indésirables.

## II.5 Observation

**Observation :** Procédé scientifique d'investigation, constatation attentive des phénomènes tels qu'ils se produisent, sans volonté de les modifier [Rob96].

Dans notre situation, les phénomènes que nous désirons observer sont les problèmes de performance potentiels.

### II.5.1 Pourquoi observer

Les besoins d'observation sont très importants au cours de la mise au point de programmes parallèles. Il est essentiel d'avoir une idée précise de la manière dont s'exécute un programme pour identifier les problèmes de performance et pouvoir modifier le comportement de ce programme en vue de corriger des erreurs ou d'obtenir de meilleures performances. Malheureusement, de par leur nature très dynamique, irrégulière et distribuée, il est difficile d'avoir une vision simple du déroulement de programmes parallèles.

Dans des cas simples, il arrive que la seule observation des résultats d'un programme permette de déterminer l'origine de certaines erreurs ou de problèmes d'efficacité. Par exemple, MPI-Povray [Ver], la version parallèle du programme de synthèse d'images par lancer de rayons Povray affiche progressivement les zones de l'image calculées par chacun des nœuds de calcul. En constatant que l'affichage d'une zone prend beaucoup plus de temps que les autres, on peut en déduire un déséquilibre de la charge de travail et tenter de réduire le temps total de rendu en améliorant la répartition des tâches de calcul.

Malheureusement ces observations simples dépendent des connaissances empiriques du programmeur et ne sont ni généralisables, ni suffisamment précises pour des applications plus complexes. Il faut donc trouver des méthodes pour observer en détail le déroulement d'un programme dans le but d'en comprendre le fonctionnement. L'observation pertinente du déroulement d'un programme doit permettre, par exemple, d'identifier les points de contention de données ou les périodes d'inactivité de processus. Ceci offre une base de travail intéressante pour modifier le comportement d'un programme en vue de l'optimisation de ses performances.

Le déroulement général d'un cycle d'observation d'un programme est le suivant (voir figure II.5) :

– collecte d'informations (variables du programme, informations du système, va-

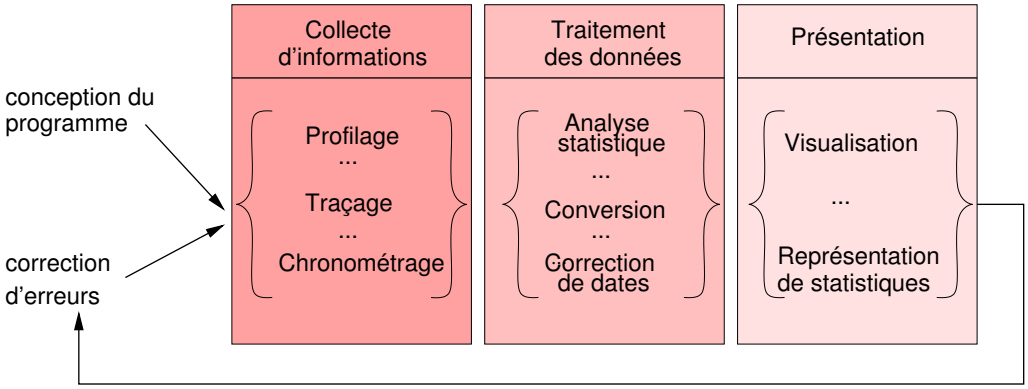



**Figure** II.5 – Cycle d'observation d'un programme

leurs des registres du processeur...) ;
- traitement des données ;
- présentation des informations sous une forme pertinente.

Chacune de ces phases peut être réalisée de différentes manières ayant chacune des finalités différentes. Cela nous amène à considérer diverses techniques d'observation :

- la surveillance ;
- le débogage interactif ;
- les indicateurs statistiques ;
- le profilage ;
- l'observation comportementale.

Ces techniques d'observation seront examinées plus en détail au chapitre III.

L'interprétation d'observations liées aux performances et au déroulement d'une application peut également être importante dans d'autres situations : guider des algorithmes d'ordonnancement ou adapter le comportement d'un programme de manière dynamique en fonction de son déroulement par exemple. Il faudra pour cela effectuer l'observation *en ligne*.

L'une des qualités fondamentales d'une observation est de minimiser *l'intrusion* c'est-à-dire les perturbations engendrées sur l'exécution du programme par les opérations d'observation. Cette intrusion peut en effet amener le programme observé à modifier son efficacité, voire à engendrer un résultat différent de celui produit sans observation.

### II.5.2 Observation en ligne ou *post mortem*

Les techniques d'observation d'exécutions de programmes parallèles peuvent être classées selon deux modes de fonctionnement : *en ligne* ou *post mortem*. L'observation

en ligne consiste à récupérer et à traiter les informations collectées sans attendre la fin de l'exécution du programme observé. Par opposition, l'observation *post mortem* effectue l'analyse des informations collectées *après* la fin d'exécution. Chacune de ces techniques présente des intérêts et des inconvénients et la réalisation de fonctions d'observation, d'une manière ou d'une autre, est sujette à discussion. Il est en effet possible d'effectuer certaines opérations de différentes manières et à des moments différents : avant, après ou au cours de l'exécution du programme à observer.

L'analyse en ligne des données générées par les procédés d'observation est utile si on désire une visualisation « temps-réel » des observation : il faut récupérer les données d'observation sans attendre la fin de l'exécution. À moins de traiter les informations localement, cela implique une surcharge non négligeable du trafic sur le réseau et perturbe d'autant plus l'exécution du programme (une parade est d'utiliser un réseau dédié à la collecte des données par exemple un réseau Ethernet en complément du réseau d'interconnexion). L'avantage du traitement en ligne des données est de permettre de modifier de manière dynamique le comportement d'un programme durant son exécution. Ceci peut être fait de deux manières : automatiquement par un système de rétroaction ou bien interactivement par l'utilisateur si le temps d'exécution et la structure du programme le permettent. Par exemple, la détection d'un excès de communications sur une interface réseau d'un nœud peut nécessiter la migration de travaux de ce nœud vers un autre pour équilibrer le trafic réseau et ainsi éviter une saturation.

L'observation *post mortem* consiste, quant à elle, à enregistrer les informations désirées (des événements par exemple) en minimisant l'impact de l'enregistrement sur le déroulement du programme. Les avantages d'une interprétation *post mortem* sont :

- La limitation des perturbations induites sur le programme observé :
  les performances des programmes dépendent également de la charge du réseau d'interconnexion. Si on récupère les données durant l'exécution du programme, on soumet celui-ci à des perturbations pouvant modifier son comportement, ce qui est gênant pour une bonne observation.
- Une plus grande souplesse dans le traitement des données :
  Il est parfois utile d'effectuer des traitements sur les données collectées (correction d'horloge, tri, filtrage, etc.). Ces traitements sont généralement plus simples à réaliser en ayant la connaissance de l'ensemble des données.

À moins de n'enregistrer qu'un nombre limité d'informations, ce qui est le cas pour les techniques de comptage et de chronométrage, il est impossible de recourir à une observation en ligne sous peine de modifier profondément le comportement de l'application observée.

## II.6 Conclusion

Nous avons présenté dans ce chapitre le contexte de cette thèse : le débogage pour les performances d'applications parallèles. Nous avons exposé aussi l'environnement logiciel (INUKTITUT, ATHAPASCAN etc.) et matériel (grappes de PC) qui a motivé notre travail et influencé certaines orientations techniques. Nous avons enfin montré les distinctions entre le débogage pour la correction de programmes et le débogage pour les performances.

Notre choix pour aider les programmeurs à effectuer le débogage pour les performances de leurs applications est de proposer des outils d'observation d'exécution de programmes parallèles. Cette approche basée sur l'examen du comportement d'un programme requiert d'être effectuée de manière *post mortem* pour garantir l'observation la plus proche possible de la réalité. Devant la diversité des outils de programmation utilisés pour aboutir à la conception d'une application (langages de programmation parallèle, bibliothèques de communication, etc.), il est important de pouvoir adapter l'observation aux différentes caractéristiques de chacun de ces outils. De plus, la diversité des plateformes rencontrées et leur évolution très rapide nécessite une grande adaptabilité de la part de tous ces outils. Notre objectif est donc de fournir des outils flexibles pour l'observation comportementale *post mortem*.

Chapitre

# III

# Techniques d'observation et de collecte de données

*«Il est certaines actions qui ont une fin et pas de commencement,*
*alors que d'autres commencent pour ne jamais s'achever.*
*Tout dépend de la position de celui qui observe.»*
Frank Herbert *Les enfants de Dune*

Nous avons présenté l'intérêt de l'observation d'exécutions de programmes parallèles pour la mise au point et l'optimisation de performances. La réalisation de ces observations peut s'effectuer de diverses manières. Une approche globale permet l'observation d'un comportement général et l'identification des problèmes les plus flagrants tandis qu'une approche comportementale détaillée permettra d'isoler plus finement un problème précis. L'observation se basant sur des données collectées lors de l'exécution d'un programme, il faut disposer de mécanismes de récolte et d'enregistrement. Ceux-ci doivent limiter la perturbation induite sur le programme observé et être aptes à traiter des programmes déployés sur un grand nombre de nœuds.

Dans ce chapitre, nous présentons les principales approches possibles pour observer l'exécution d'un programme ainsi que les techniques de collecte de données sous-jacentes. Ensuite nous abordons l'instrumentation de programmes.

## III.1 Techniques d'observation

En fonction du but recherché —analyse des ressources utilisées sur une machine, débogage pour la correction ou encore débogage pour les performances— les techniques utilisées pour obtenir des observations pertinentes sont variables. Nous présentons ici les principales approches dans le domaine du parallélisme et leur domaine d'utilisation.

### III.1.1 Surveillance (*monitoring*)

La surveillance de l'exécution d'un programme consiste à observer de manière extérieure au programme un certain nombre de paramètres dont les variations reflètent l'activité de celui-ci. La surveillance se base sur les mécanismes existants notamment

dans le système d'exploitation chargé d'exécuter le programme à observer (le pseudo système de fichiers `/proc` sous Linux par exemple).

Un avantage majeur d'effectuer l'observation de manière externe à l'application cible est de ne pas nécessiter d'instrumentation du code applicatif. Il n'est pas nécessaire de modifier le programme pour obtenir des informations sur ses interactions avec le système (utilisation CPU, consommation mémoire, accès aux interfaces réseaux, etc.) car les mécanismes nécessaires à l'observation sont implantés directement dans le système d'exploitation. Cette faculté rend l'observation très simple à réaliser.

En observant les interactions et l'état des ressources gérées par le système, on peut identifier certains problèmes de fonctionnement ou de performance comme la saturation de la mémoire ou l'existence de périodes d'inactivité. Ces techniques sont utilisées de manière intensive par les administrateurs système pour surveiller l'activité du parc de machines dont ils ont la charge. L'objectif est de détecter les anomalies de fonctionnement et d'effectuer les corrections nécessaires afin de rétablir un service satisfaisant pour les utilisateurs. Ces corrections peuvent être la suppression de processus trop consommateurs de ressources ou une meilleure répartition des charges de calcul sur plusieurs machines.

Les outils de surveillance les plus répandus sont, par exemple, `xload` [xlo] ou `top` [topa] qui affichent respectivement un graphe de la charge CPU de l'ensemble des processus au cours du temps et la liste des processus les plus consommateurs de ressources. Ces outils ont été conçus pour fournir des informations concernant une seule machine (éventuellement SMP). Ils sont le plus souvent utilisés de manière dynamique, c'est-à-dire que, sur la base de leur consultation, l'administrateur peut intervenir sur les sources des problèmes constatés et ainsi améliorer l'efficacité du système. Les outils de surveillance usuels (`xload` [xlo], `xosview` [xos], `top`, `ps` [ps], `gkrellm` [gkr]) peuvent être utilisés pour la surveillance d'une machine ou d'un nombre limité de machines (tout au plus quelques dizaines de machines). Ils ne sont cependant pas aptes à rendre compte de l'activité d'une grappe ou d'un parc de machines de grande taille.

Pour s'adapter à la surveillance de machines parallèles, des outils plus spécialisés ont été conçus : `bWatch` [Rad] `Ganglia` [Mas] `Performance Co-Pilot(PCP)` [SGI], ACE [Inc]. Ils offrent des fonctions de collecte des informations sur un grand nombre de nœuds. PCP, par exemple, utilise des démons s'exécutant sur chaque nœud pour récolter les informations localement et les envoyer à un serveur pour analyse.

Une approche intéressante est celle proposée par PHOENIX [BBF00], un outil de surveillance flexible pour la création de service de gestion de grappes. Du point de vue du fonctionnement, PHOENIX propose plusieurs modules, pour enregistrer les opérations système, instrumenter les applications, gérer les communications et faciliter la conception de composants externes. Un aspect particulièrement notable concerne la possibilité de définir, grâce à un langage de description, des conditions

sur l'activation des sondes d'enregistrement. Ce mécanisme permet d'adapter dynamiquement la granularité de la surveillance pour limiter les perturbations sur l'application observée.

#### III.1.1.1 Surveillance système et réseau

La surveillance des informations provenant du réseau d'interconnexion ou du système d'exploitation est très largement utilisée pour observer le comportement d'applications s'exécutant sur une machine ou un groupe de machines. Ces informations intéressent non seulement les administrateurs système pour connaître l'état d'un parc de machines ou d'une grappe de PC mais également les concepteurs de programmes pour examiner le comportement de leurs applications. En s'intéressant par exemple à la quantité de mémoire vive disponible, à la charge CPU et au débit des communications, on peut déterminer si les performances d'un programme sont bridées par des limitations de mémoire, du processeur ou du réseau. Si un nœud d'une grappe est particulièrement sollicité, il peut être utile de s'interroger sur la cause de ce phénomène et de repenser l'équilibrage des données ou des calculs. Si l'application cible le permet, de par sa durée d'exécution et sa structure algorithmique, on peut également mettre en place un système d'ordonnancement prenant en compte une surcharge pour équilibrer les tâches.

Il est possible de prendre en considération beaucoup de paramètres système pour surveiller une machine. Citons principalement la charge du processeur, la mémoire vive disponible ou encore la taille du fichier d'échange mémoire (*swap*) utilisé. On peut également vouloir surveiller la quantité de données transitant par la carte d'interconnexion ou le trafic sur les commutateurs du réseau. L'outil `tcpdump` [tcp] permet, par exemple, de capturer le trafic réseau.

Certains aspects de la visualisation des informations issues de la surveillance de grandes grappes de PC au niveau système ont été traités dans [GCdKdOSA01]. Nous présentons dans cet article les avantages retirés, pour la surveillance d'une grappe de PC de grande taille, de l'utilisation de l'outil de visualisation Pajé (présenté au chapitre 5).

L'observation des ressources système d'une machine permet d'obtenir des informations sur l'activité des programmes en cours d'exécution de manière globale et peu fine. L'affinage des observations comme, par exemple, l'augmentation de la fréquence d'échantillonnage de la mesure de charge du processeur peut conduire à une surcharge de travail rendant l'application inefficace et l'observation non pertinente. Un exemple flagrant est que si l'on augmente la fréquence de rafraîchissement de l'outil `top` (par appui sur la barre d'espace) on constate qu'il arrive rapidement en tête de la liste des processus en consommation CPU et sature le système.

#### III.1.1.2 Surveillance matérielle

Certains outils permettent d'utiliser les compteurs matériels (voir section III.2.3.1) pour obtenir des informations provenant directement des registres du processeur (voir la section III.2.3.1 à propos des compteurs de performance) ou des composants de la machine. Par exemple, le module `lm_sensor` [lms] permet d'obtenir des informations sur une machine telles que la vitesse de rotation des ventilateurs ou la température de certains composants.

L'outil VTune [VTu03] permet de localiser et d'identifier des points de contention dans un programme. Il procède par l'affiche d'un graphe représentant l'évolution au cours du temps de valeurs pertinentes telle que le nombre d'opérations flottantes par seconde effectuées par le processeur.

La surveillance ne peut que difficilement obtenir des informations détaillées sur un programme particulier. Pour obtenir des informations plus précises sur le comportement d'applications, il est indispensable de changer de stratégie d'observation et de se placer dans un contexte applicatif.

### III.1.2 Débogage interactif

Le but du *débogage interactif* est d'observer le comportement d'un programme pendant son exécution. Le principe est de placer des *points d'arrêt* (*breakpoint*) sur des instructions du code que l'on désire étudier. Le programme arrête alors son exécution sur les instructions marquées et il est possible de connaître les valeurs des variables et des structures de données du programme. Il est ensuite possible d'exécuter le programme instruction par instruction ou par blocs d'instructions. On peut ainsi examiner très précisément l'évolution des variables, ce qui permet de déterminer et corriger un grand nombre d'erreurs de correction du programme difficilement décelables autrement, comme des accès à des zones de mémoire interdits.

En reprenant le principe des débogueurs séquentiels, c'est-à-dire arrêter une exécution pour examiner les structures de données utilisées, il est possible de concevoir des débogueurs pour programmes parallèles. Gdb [gdb] [LO97], par exemple, permet l'observation de programmes utilisant des *threads*.

Un important problème du débogage de programmes parallèles communiquant par envoi de messages vient de la complexité à stopper leur exécution. Il est en effet difficile de garantir une vue cohérente (sans messages en attente par exemple) à un instant donné. L'outil de débogage pour programmes parallèles le plus classique est TotalView [Tot]. Il a été développé spécifiquement pour répondre aux contraintes du débogage de programmes parallèles. D'autres outils sont axés vers le débogage d'applications distribuées plus spécifiquement liées à une bibliothèque particulière

par exemple DDBG [CLAa99] pour le débogage de programmes basés sur PVM.

Bien que très utiles pour la correction des programmes, ces techniques ne sont pas utilisables pour l'optimisation de performances. En effet, l'interactivité bloque l'exécution du programme et perturbe donc son déroulement de façon importante. Une telle intrusion rend non pertinente toute mesure de performance.

### III.1.3 Génération d'indicateurs statistiques

À l'aide de certains types d'observation (chronométrage, comptage ou traçage, voir sections III.2.2, III.2.3 et III.2.4 respectivement), il est possible d'extraire des indicateurs statistiques concernant des actions effectuées au cours de l'exécution d'un programme. Il s'agit d'indicateurs de tendance de métriques de performance (minimum, maximum, moyenne, somme etc.) qui peuvent être de différente nature. Ainsi, on peut par exemple s'intéresser au temps moyen passé en attente d'un verrou par un *thread*. Cette indication permet d'avoir une idée quantitative du temps perdu lors de synchronisations d'accès à des ressources.

À partir de ces informations, le programmeur peut isoler les activités du programme qui causent les plus importantes pertes de performance. Par exemple, calculer le pourcentage de temps passé par les *threads* dans un état inactif permet de découvrir des problèmes de synchronisation ou de répartition de charge.

Ces indicateurs statistiques peuvent être difficiles à interpréter ou être trop globaux pour isoler un problème précis. Ils sont, par contre, une aide préalable précieuse afin de déterminer dans quelle direction chercher pour améliorer les performances.

### III.1.4 Profilage (*profiling*)

Le profilage de l'exécution d'un programme consiste à estimer le temps passé dans chaque sous-partie d'un programme. Cela permet de dresser un *profil d'exécution* à partir duquel il est possible de déterminer les parties de code les plus utilisées, c'est-à-dire celles où le programme passe le plus de temps. Ce sont ces zones que l'on cherchera à optimiser en priorité.

Le profilage permet d'obtenir des informations sur le comportement global d'un programme. Les informations offertes par un profileur sont très synthétiques. On peut être informé de la zone dans laquelle apparaît un problème de performance grâce au profil obtenu. Il est généralement nécessaire de disposer de plus de détails pour pouvoir l'identifier précisément. En se référant aux critères d'observation du modèle *W3* (voir section II.4.2.3 p.16), on peut dire que le profilage permet d'identifier *où* les problèmes de performance apparaissent, mais pas toujours *pourquoi*. Pour

détailler un point particulier de l'exécution d'un programme, il faut disposer de plus d'informations que les simples durées de chaque procédure.

### III.1.5 Observation comportementale

Le profilage et la génération d'indicateurs statistiques offrent des indications globales sur le déroulement d'un programme. Toujours selon les critères modèle *W3*, ces indications globales permettent de déterminer *où* situer dans le code les problèmes de performance avec précision (à la ligne de code près), mais il est par contre bien plus difficile de déterminer à quel moment a eu lieu ce problème. Les fonctions qui sont susceptibles d'impliquer des baisses de performance sont celles qui sont exécutées un grand nombre de fois au cours d'une exécution de programme. Or, les techniques de profilage ou de génération d'indicateurs statistiques ne conservent aucune information temporelle.

Pour parvenir à observer le déroulement d'un programme et comprendre les relations entre les opérations qui ont mené à un problème de performance, il est indispensable de faire appel à l'observation comportementale. L'observation comportementale consiste à reconstruire le déroulement complet d'une exécution pour permettre d'appréhender l'ensemble des actions survenues au cours de l'exécution. L'observation comportementale se base donc sur l'enregistrement d'un ensemble d'actions effectuées au cours d'une exécution. Chaque enregistrement doit être daté pour pouvoir déterminer l'ordre d'occurrence des événements survenus. Cette date peut être aussi bien logique [Lam78] que physique. Son but est de pouvoir identifier l'ordre d'occurrence des événements. Dans le cadre du débogage pour les performances, il est cependant nécessaire de recourir à une datation physique se rapportant à une horloge de référence. Cela est important pour pouvoir comparer des durées d'actions issues d'exécutions différentes ou de sites distincts afin de pouvoir estimer les gains de temps d'un programme corrigé par exemple.

## III.2 Collecte de données

De même qu'il existe diverses manières d'observer l'exécution d'un programme, il existe plusieurs techniques pour obtenir les informations nécessaires à ces méthodes d'observation. Nous classons celles-ci en deux catégories distinctes (voir figure III.1) selon la manière dont sont déclenchés les enregistrements élémentaires d'information : les techniques guidées par le temps et les techniques guidées par les événements. Les techniques guidées par le temps enregistrent des informations à partir d'une contrainte temporelle : par exemple à intervalles réguliers. C'est le

principe de l'échantillonnage (*sampling*). Pour les techniques guidées par les événements, les enregistrements des informations sont déclenchés par l'occurrence d'un événement particulier. Cet événement est généralement l'exécution d'une portion de code particulière.

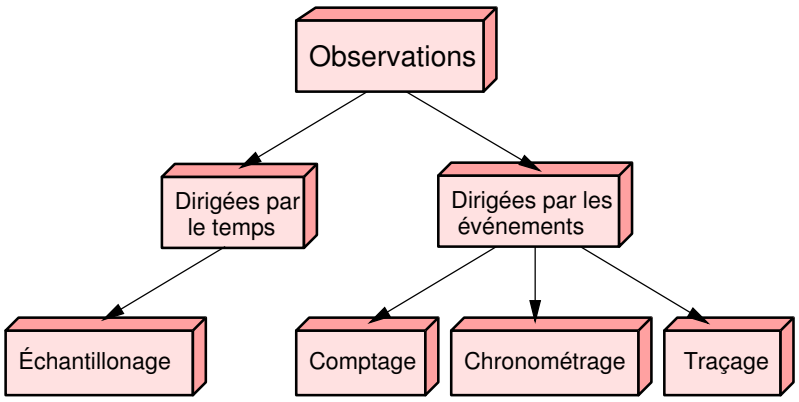



**Figure** III.1 – Classification des techniques d'enregistrement d'informations

### III.2.1 Échantillonnage

L'échantillonnage consiste à examiner périodiquement l'état d'un objet afin de rendre compte de son évolution. On appelle *pas d'échantillonnage* l'intervalle de temps entre deux mesures. Par exemple, on enregistre toutes les 10 ms le nom de la fonction courante (identifiée par un compteur du système d'exploitation) et, pour chaque fonction, on tient le compte du nombre de fois où elle est observée. On considère ensuite que le pourcentage de temps passé dans chaque fonction est proportionnel à son taux d'observation, ce qui permet de dresser un profil d'exécution.

C'est de cette manière que fonctionne le programme gprof [gpr] qui permet également de reconstruire le graphe des appels de fonction au cours d'une exécution.

La qualité de l'échantillonnage est fortement liée au pas d'échantillonnage. Si celui-ci est petit, les observations seront plus fines mais l'intrusion grande. S'il est au contraire trop grand, des actions de durée inférieure pourront ne pas être observées.

L'échantillonnage est très utilisé pour la surveillance système et réseau. La plupart des outils destinés à l'administration système se basent sur un système d'échantillonnage et de lissage afin d'engendrer leurs données d'observation. Le lissage consiste à remplacer, à chaque pas de temps, la valeur instantanée observée par la moyenne des quelques dernières observations pour éviter de brusques fluctuations peu pertinentes.

### III.2.2 Chronométrage

Le chronométrage consiste à mesurer le temps passé dans une zone de code en l'encadrant par des instructions de début et de fin de comptage du temps. Il est généralement utilisé pour mesurer le temps pris par une fonction que le programmeur juge importante et à optimiser. En chronométrant chaque fonction d'un programme, on peut dresser un profil d'exécution de celui-ci. C'est le principe d'outils tels que TAU (voir section V.1.2).

L'intrusion due au chronométrage est variable, elle dépend fortement de la fréquence d'occurrence des actions chronométrées. La quantité de données à enregistrer est réduite à quelques valeurs par action chronométrée, cela ne consomme donc que très peu de mémoire sur le site d'exécution du programme.

### III.2.3 Comptage

Le comptage comme son nom l'indique permet de connaître le nombre d'occurrences d'un événement, tel que l'exécution d'une fonction, ou le nombre d'accès à un objet. Il est utilisé pour déterminer quels fonctions ou objets sont fréquemment utilisés et donc à optimiser prioritairement.

L'intérêt du comptage est d'être peu intrusif. En effet, peu d'informations sont enregistrées et elles le sont de manière très simple (il suffit d'incrémenter une variable, il n'y a pas de copie de données). Il n'y a donc que peu d'intrusion due à l'obtention d'informations et à la gestion de celles-ci. Les techniques d'observation par comptage sont donc particulièrement bien adaptées à l'observation d'exécutions longues sur un grand nombre de nœuds.

Une facette du comptage, de plus en plus courante, est l'utilisation de compteurs de performance des processeurs afin d'obtenir des informations très détaillées et précises tout en conservant une faible intrusion.

#### III.2.3.1 Les compteurs matériels de performance

Dans les processeurs actuels (PIII [Int01], PIV [Int01], Athlon [AMD02], Itanium, Power3, MIPS, UltraSparc entre autres), il existe un certain nombre de registres servant de *compteurs matériels de performance*. Le processeur peut incrémenter la valeur de ces registres à chaque occurrence d'un événement particulier. On pourra consulter ces compteurs pour obtenir des informations sur le déroulement d'un pro-

gramme exécuté sur ce processeur. Les événements comptés concernent différents aspects du fonctionnement du processeur :

- les accès aux divers caches mémoire du processeur ;
- le nombre d'instructions (flottantes ou entières) ;
- le nombre d'instructions de branchements ;
- le nombre de cycles perdus en attente ;
- les contentions du bus de données.

Le comptage n'est pas effectué par voie logicielle grâce au système d'exploitation mais directement par le processeur. L'intrusion due au comptage est donc nulle, elle ne modifie pas le comportement du programme observé. La lecture des informations dans de tels registres est très peu coûteuse. À titre d'exemple, avec l'interface PAPI, l'initialisation de 2 compteurs (start/stop) se fait en 475 instructions et la lecture de ces compteurs en 259 instructions. Ces coûts peuvent être comparer à la durée de la lecture de la date du système d'exploitation : environ 1500 instructions avec `gettimeofday`.

Un exemple, significatif et couramment utilisé, de leur intérêt est de pouvoir accéder au nombre d'*opérations flottantes par seconde* effectuées par le processeur au cours du déroulement d'un programme. Il est bien sûr possible de déterminer théoriquement le nombre d'opérations que doit accomplir un programme pour achever un calcul, mais cela est parfois extrêmement complexe pour un programme réel. Par ailleurs, le coût de comptage de certains événements serait extrêmement élevé s'il était effectué au niveau du code utilisateur. En déléguant cette tâche au processeur, ces comptages s'effectuent au plus bas niveau possible, ce qui limite fortement le coût en mémoire et en temps de calcul.

**Accès aux compteurs** L'utilisation de ces compteurs comporte cependant quelques contraintes. Tout d'abord en raison du coût (financier) très élevé de l'implantation de registres dans un processeur. Le nombre de compteurs est généralement très inférieur au nombre d'événements comptables. Il y a, par exemple, environ 50 événements comptables sur un pentium III pour seulement 2 registres. Les processeurs plus récents ont, malgré tout, tendance à disposer de plus de registres (18 pour le PIV par exemple).

Les processeurs plus récents tels que le Pentium IV possèdent de l'ordre d'une dizaine de compteurs de performance. Il faut donc définir préalablement quels événements doivent être observés avant de pouvoir les lire. Par ailleurs, l'utilisation directe des registres n'est pas possible pour un programme utilisateur. Il faut passer par le mode privilégié du système d'exploitation. Pour cela, il faut que le système d'exploitation prévoie un accès à ces registres. C'est le cas, par exemple, avec le système AIX sur les processeurs Power3 et Power4. Le noyau Linux actuel quant à lui ne comporte pas de tels mécanismes et doit être modifié (avec perfctr [Per03] par exemple) pour offrir ces fonctions.

**Standardisation de l'accès aux compteurs** Les compteurs sont différents sur chaque processeur, à la fois par leur mode d'utilisation et par les événements qu'il est possible de compter. Pour uniformiser leur utilisation et assurer la portabilité des programmes les utilisant, des interfaces (*API*) d'accès aux registres ont été développées [Sey01] [Rab] [Bri]. PAPI [LDM+01] [DLM+01] est un exemple d'une telle interface. C'est une bibliothèque de fonctions qui a pour but de fournir une interface portable et standardisée pour l'accès aux différents compteurs matériels des divers processeurs actuels. PAPI fournit des fonctions pour initialiser, configurer et lire les compteurs de performance. L'avantage de l'utilisation d'une telle bibliothèque est de profiter d'un accès homogène aux compteurs de performance des différentes architectures supportées sans devoir réécrire le code du programme utilisateur. PAPI comporte également des fonctions de *multiplexage* permettant de compter plus d'événements que le nombre de registres dédiés.

**Observation multi-niveaux** L'utilisation de tous ces compteurs pour obtenir des informations présente donc des avantages très significatifs : ils permettent d'accéder à des informations de très bas niveau qui ne pourraient pas, ou très difficilement, être obtenues par d'autres moyens. De plus leur utilisation est très peu coûteuse, ce qui est particulièrement intéressant pour limiter l'intrusion indésirable dans le cadre du débogage pour les performances.

La présentation d'informations issues des compteurs matériels de performance en vis-à-vis des informations de trace d'un niveau applicatif (voir figure III.2) peut être bénéfique pour une bonne compréhension des problèmes de performance rencontrés. En présentant des informations du niveau applicatif et des informations de performance du niveau processeur, on pourra disposer d'une vue simultanée des activités d'un programme et des performances obtenues. Il sera ainsi possible d'identifier simultanément le moment où un problème de performance apparaît et ses causes potentielles dans le comportement du programme. La corrélation de ces informations permet de mieux comprendre le déroulement d'un programme et de localiser les moments où apparaissent des problèmes de performance.

Bien que la corrélation d'événements soit très intéressante pour la compréhension du déroulement d'un programme parallèle, elle est souvent très difficile à effectuer. Il est en effet complexe d'apparier causalement des événements de plusieurs niveaux d'abstraction si ceux-ci sont nombreux et temporellement distants. Il peut être alors nécessaire de disposer d'outils d'analyse effectuant (ou aidant à) la mise en correspondance d'événements selon différents critères.

Pour pouvoir intégrer des informations issues des compteurs de performance à un outil d'observation, il faut qu'il soit spécifiquement conçu pour cela comme VTune, ou bien qu'il soit suffisamment flexible pour pouvoir enregistrer n'importe quels type d'informations.

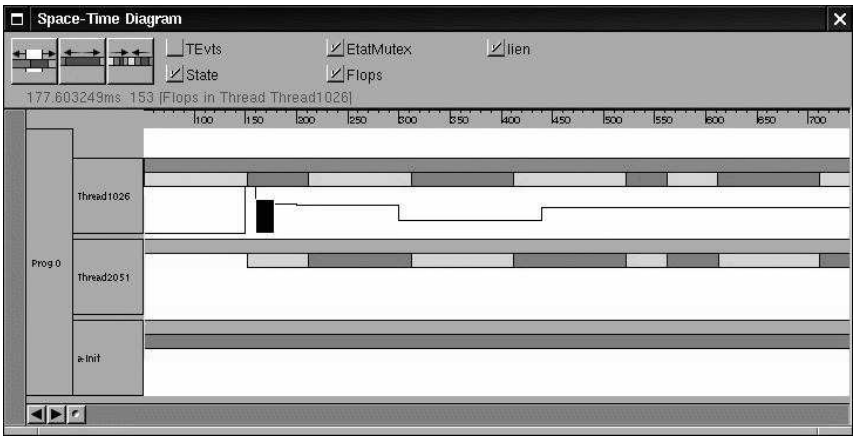



**Figure** III.2 – Visualisation d'informations des compteurs de performance

La représentation simultanée d'activités de *threads* —sous la forme de rectangles de couleur— et du nombre d'instructions flottantes par seconde (sous la forme d'une courbe face au premier *thread* (n° 1026) sur cette figure) permet d'identifier les zones de forte ou faible activité et ainsi de localiser les problèmes de performance potentiels.

**Compteur de temps** On peut noter l'existence d'un autre compteur particulier très utilisé, le registre de comptage du temps TSC [Int98] (*Time Stamp Counter*). Il est initialisé au démarrage de l'ordinateur et incrémenté de 1 à chaque tic d'horloge. Il permet donc de mesurer très précisément des durées d'exécution en nombre de cycles d'instruction. Il s'agit d'un registre spécialisé indépendant des autres registres compteurs de performance. Son utilisation est environ 50 à 100 fois moins coûteuse (selon le processeur) que l'appel à la fonction `gettimeofday` interrogeant l'horloge système. L'un des avantages de ce registre est sa lecture par un programme en mode utilisateur, ce qui facilite son utilisation. Un accès à ce compteur est très peu coûteux, de l'ordre de quelques dizaines de cycles d'horloge : 15 sur Athlon ; 34 sur PIII ; 84 sur PIV. Ceci est beaucoup moins coûteux qu'un appel à la fonction `gettimeofday` qui est de l'ordre de 1500 tics d'horloge. Il est cependant délicat de s'en servir pour déterminer une date d'occurrence d'événement car sa valeur dépend à la fois de la fréquence du processeur, qui peut varier en fonction de l'utilisation du processeur [Int], et de la date d'initialisation de la machine.

### III.2.4 Traçage événementiel

Le traçage consiste à enregistrer des informations significatives du programme observé afin de conserver une *trace d'exécution* permettant de représenter le comportement du programme.

Dans le cadre du traçage événementiel, les informations enregistrées sont consti-

tuées des descriptions d'événements qui ont eu lieu au cours du déroulement du programme. Une trace est classiquement constituée d'une suite d'*enregistrements*. Chaque enregistrement est constitué de plusieurs *champs*. Chaque champ permet de sauvegarder la valeur d'un *paramètre*.

**Définition 1** *On définit un* ***événement*** *comme le début ou la fin d'une action exécutée par un processus, qui change l'état du nœud correspondant comme, par exemple, les registres du processeur, les canaux d'entrée/sortie ou encore la mémoire.*

Selon le niveau d'abstraction choisi, un événement peut être représenté par un ou plusieurs enregistrements mais un enregistrement peut également être associé à plusieurs événements. Les événements typiques que nous désirons observer sont, par exemple, une modification de l'état d'un processus, une prise de verrou, ou encore un envoi ou une réception de message. À partir de ces événements, on peut reconstituer le déroulement d'une application ou tout au moins en obtenir une représentation. Celle-ci est plus ou moins fidèle en fonction du nombre d'événements enregistrés et de la précision avec laquelle ces enregistrements ont été effectués. Cette intrusion va perturber le programme observé et ainsi réduire la validité de l'observation. Le nombre d'événements pris en considération doit donc être un compromis entre précision et fidélité de l'observation.

Le traçage est nécessaire pour la *reconstruction*, de manière cohérente et complète, de l'enchaînement des événements ayant eu lieu au cours de l'exécution du programme. Pour cela, des paramètres doivent être enregistrés et la nature de ceux-ci dépend de l'exploitation que l'on veut faire de la trace. Généralement, les paramètres utiles sont *la date* (physique ou logique [Lam78]) pour placer temporellement l'événement, et une *identification* du lieu d'occurrence de cet événement pour le placer spatialement. Ce lieu peut aussi bien être concret, comme un nœud de calcul ou un processeur, qu'abstrait comme un *thread* ou un objet si l'on s'intéresse à l'exécution de méthodes. De plus, on doit enregistrer les paramètres dépendant directement de l'événement considéré. Par exemple, un événement d'envoi de message devra être accompagné de l'origine et du destinataire du message. Il pourra également être associé à des données émises.

La précision de la reconstruction est liée au nombre d'événements et à la précision de la datation des événements. Plus le nombre d'événements tracés est important, plus la reconstitution obtenue est fidèle. Le corollaire à un plus grand nombre d'enregistrements est une plus forte intrusion. Il est a noter qu'il est possible de générer des indices statistiques ou un profil d'exécution à partir de données générées par des techniques de traçage alors que l'inverse ne le permet pas.

Il existe différentes méthodes pour réaliser ce processus de traçage. Nous allons distinguer plusieurs sortes de traçage : matériel, hybride et logiciel.

### III.2.4.1 Traçage matériel

Des dispositifs matériels spécialisés, ajoutés aux machines utilisées, sont chargés de sauvegarder des informations nécessaires à la reconstitution du déroulement du programme (ou d'un aspect particulier du programme). Le traçage matériel a l'avantage de réduire l'intrusion quasiment à néant. Il a l'inconvénient d'être coûteux et d'un prix au moins proportionnel au nombre de machines tracées. Il est ainsi généralement plus utilisé dans de gros calculateurs que pour des grappes de stations de travail. Le développement d'un système de traçage matériel est également plus lourd et coûteux en investissements. Un inconvénient notable du traçage matériel est qu'il n'est pas forcément aisé de corréler au niveau applicatif les causes d'un phénomène observé au niveau matériel. Un système de traçage matériel est, par définition, construit physiquement et lié à une machine. Il est donc extrêmement difficile de le modifier. Cela est très pénalisant dans le cadre de la mise au point de techniques de programmation parallèle amenées à évoluer très rapidement.

### III.2.4.2 Traçage hybride

Le *traçage hybride* est un compromis entre le traçage logiciel et le traçage matériel. Il consiste à utiliser des mécanismes matériels d'enregistrement activés par des instructions au niveau logiciel. L'identification des événements se fait au niveau logiciel et leur enregistrement et leur estampillage par un système matériel. Il est ainsi plus aisé de corréler les phénomènes logiciels et les événements observés qu'avec le traçage matériel tout en bénéficiant d'une intrusion réduite par rapport au traçage logiciel grâce à l'utilisation de matériel dédié.

Le problème principal de l'utilisation d'un système de traçage hybride est celui de l'adaptation du système matériel au programme observé. Il n'existe pas, en principe, d'interface utilisable permettant la transmission d'informations, concernant l'événement courant, du système d'exploitation vers le système d'enregistrement. Pour effectuer cette transmission d'information, trois techniques principales sont utilisées [Moh92] : transmission directe *via* un système matériel conçu explicitement pour cela, transmission par une interface de sortie (port parallèle ou série par exemple) ou encore *via* le bus de communication interne. La première solution exige du matériel spécifique qu'on ne trouve pas sur les machines courantes. La deuxième solution implique un système de collecte d'informations direct par une interface de sortie de type port série ou parallèle. Elle n'est cependant concevable que si le mécanisme de sortie n'implique pas le système d'exploitation, ce qui entraînerait des surcoûts prohibitifs. La dernière méthode, quant à elle, nécessite la mise en place d'un système de collecte des informations émises sur le bus pour chaque machine (une carte de type PCI par exemple), ce qui implique un coût financier supplémentaire et la

nécessité de placer un tel dispositif dans chacune des machines utilisées, ce qui est peu compatible avec le concept de grappes de PC bon marché et impossible dans le cas de l'utilisation de jachères de calcul.

Ces inconvénients font que la plupart des systèmes de traçage utilisent des mécanismes logiciels. Il reste cependant des cas d'utilisation où des systèmes matériels très spécialisés sont nécessaires, par exemple pour la recherche de problèmes à des échelles de temps très petites (de l'ordre de quelques cycles d'horloge).

**Utilisation ponctuelle de matériel** Un exemple pratique dans le cas de grappes de stations de travail où le traçage est logiciel, mais aidé par des dispositifs matériels est celui de la collecte de traces si on désire rapatrier les informations générées au cours du traçage pour effectuer une observation en ligne. Afin de ne pas perturber les communications sur le réseau d'interconnexion haut débit (par exemple Myrinet) utilisé par les applications, on utilise un réseau annexe d'administration (de type « Ethernet »). Cela évite de perturber les applications tout en gardant un coût financier raisonnable, car les interfaces « Ethernet » sont généralement implantées par défaut sur les cartes mères utilisées.

Une autre utilisation envisageable de matériel dédié pour le traçage hybride dans des grappes de PC consiste en un mécanisme de synchronisation matériel pour disposer d'une plus grande précision dans la prise de dates afin d'obtenir un système de datation global suffisamment fiable. Bien que très utile, un tel mécanisme reste peu adapté à un grand nombre de machines en raison de son coût au moins proportionnel au nombre de machines.

#### III.2.4.3 Traçage logiciel

Dans le cas du traçage logiciel, les observations sont effectuées par des instructions logicielles . À chaque point de trace, des instructions sont chargées de récupérer les informations liées aux événements à observer. De la même manière, les données sont collectées sans matériel spécialisé, en utilisant les périphériques de stockage et de communication existants. L'instrumentation est faite soit par la modification du programme, soit par l'utilisation d'une bibliothèque de fonctions instrumentée. Il s'agit d'une version du programme comportant des instructions pour l'enregistrement des événements. Les avantages du traçage logiciel sont une grande flexibilité et un coût d'utilisation très faible. La contrepartie est une forte intrusion.

### III.2.5 Comparaisons des techniques

**Méthodes d'observation** Nous avons vu que chacune des techniques d'observation présente des avantages et des inconvénients liés à leur nature. La surveillance est

bien adaptée pour obtenir une vue de haut niveau sur l'environnement d'exécution matériel et logiciel. Par contre, elle n'est pas utilisable pour obtenir des informations précises sur une application. Au contraire, le débogage permet d'obtenir une vue extrêmement précise de l'évolution du programme au cours du temps mais, pour cela, il perturbe à l'extrême le comportement de celui-ci. Ces deux techniques sont donc peu adaptées à la recherche de problèmes pour l'optimisation des performances. La génération d'indicateurs statistiques et le profilage permettent d'obtenir des informations assez précises et reliées à une portion de code du programme observé. Ils sont donc très utilisés lors du débogage pour les performances dans le but d'identifier *où* se produisent les problèmes. Ils ne permettent cependant pas de savoir dans quelles conditions *quand* et *pourquoi* ces problèmes ont lieu. C'est pour cela que nous nous sommes orientés vers *l'observation comportementale*.

**Méthodes de collecte** L'échantillonnage, le chronométrage et le comptage sont des techniques particulièrement adaptées pour fournir des informations globales sur l'exécution d'un programme. Elles offrent également une bonne scalabilité, pour une intrusion relativement faible. Le revers de ces qualités est que ces techniques de collecte ne sont pas aptes à fournir des informations suffisantes pour effectuer une observation comportementale. L'échantillonnage peut ne pas capturer suffisamment d'informations à cause d'un pas d'échantillonnage trop grand, le comptage et le chronométrage ne fournissant que des accumulations d'informations. Nous sommes donc contraints de nous baser sur des techniques de collecte de données par *traçage événementiel*.

**Techniques de traçage** Le traçage permet de conserver les enregistrements des événements apparus lors de l'exécution d'un programme pour permettre une reconstruction globale, ce qui est impossible à partir d'un simple profil d'exécution. L'utilisation de matériel au cours du processus de traçage résulte d'un compromis entre intrusion d'une part et coût financier, flexibilité et facilité d'utilisation d'autre part. De plus, il peut être difficile d'installer des systèmes matériels dans des machines de type grappe de stations de travail qui n'ont pas été conçues pour les recevoir, voire même impossible dans le cas d'utilisation de jachères de calcul sur un parc de machines d'une entreprise. L'utilisation de traçage matériel sur des grappes de machines est donc souvent peu envisageable, encore plus pour des grappes de PC standard dont le premier atout est le rapport performance/prix. Pour le traçage logiciel, le coût financier d'implantation est constant.

## III.3 Instrumentation

Nous avons présenté différentes techniques d'observation de l'exécution d'un programme (observation comportementale, profilage, etc.) ainsi que les méthodes uti-

lisées pour collecter les informations nécessaires à leur réalisation (échantillonnage, comptage, traçage, etc.). La plupart des techniques de collecte d'informations nécessitent une *instrumentation* de l'application à observer c'est-à-dire la mise en œuvre, à l'intérieur même du programme, de mécanismes d'enregistrement des informations nécessaires à une future exploitation. Ce mécanisme doit être le moins intrusif possible afin de perturber au minimum le déroulement du programme observé et de garantir la fidélité des observations effectuées. Les endroits où sont placés les appels à un mécanisme d'enregistrement pour l'observation sont appelés *points d'observation* ou *points de trace* dans le cas du traçage. Nous allons présenter comment l'instrumentation peut être réalisée pour le cas de figure qui nous intéresse : le traçage. Il est à noter que les différentes techniques d'instrumentation sont semblables à celles employées pour les autres méthodes de collecte de données.

### III.3.1 Techniques d'instrumentation

Les principaux points d'instrumentation possibles au cours du cycle de conception d'un programme (voir [FK99] chapitre 14) sont les suivants (figure III.3) :

- instrumentation du code source ;
- instrumentation à la compilation ;
- utilisation d'une bibliothèque instrumentée ;
- instrumentation du binaire exécutable ;
- instrumentation dynamique à l'exécution.

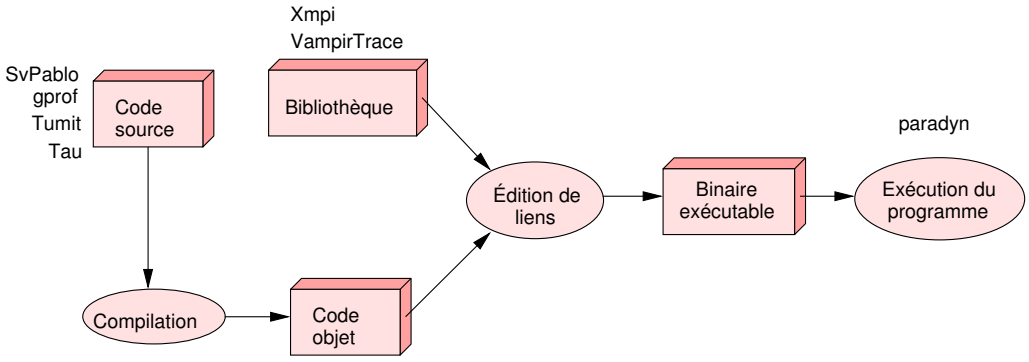



**Figure** III.3 – Les différents moments d'instrumentation au cours du cycle de compilation

### III.3.2 Instrumentation du binaire exécutable

Pour instrumenter un programme exécutable, il faut une automatisation totale du processus d'instrumentation qui se chargera de ré-écrire un programme binaire exécutable à partir de l'original. Cela permet de créer des instructions de traçage

assez efficaces. Il n'est pas envisageable pour l'utilisateur de modifier le code binaire manuellement. On doit donc disposer d'un programme pouvant lire et écrire le format du programme compilé. Ceci implique la création d'un programme complexe et peu portable, car les différences entre les fichiers binaires exécutables de diverses architectures sont importantes. Parmi les inconvénients, on peut citer la nécessité de connaître parfaitement le format de l'exécutable ce qui pose des problèmes de portabilité.

### III.3.3 Instrumentation dynamique à l'exécution

L'instrumentation dynamique consiste à placer les instructions de saut aux endroits que l'on veut instrumenter (points de trace) en allant modifier le code de l'application en mémoire. Lors de l'exécution de ces instructions, le déroulement du programme est détourné pour enregistrer les informations désirées. Après enregistrement, le cours du programme est repris normalement. Ce mécanisme de saut est nécessaire car il n'est pas possible de rajouter toutes les instructions d'enregistrement directement au niveau du point de trace. L'instrumentation dynamique permet de placer ou de supprimer des fonctions d'enregistrement pendant l'exécution du programme.

Le principal outil se basant sur l'instrumentation dynamique est PARADYN [Ma95] [ZMN99]. Buck et Hollingsworth proposent un langage [HMG+97] et une API [BH00] permettant l'instrumentation dynamique de code durant son exécution.

L'un des intérêts de l'instrumentation dynamique est la possibilité de supprimer les points d'enregistrement en cours d'exécution, par exemple si un problème d'intrusion survient afin de garder une intrusion limitée. Un avantage de l'instrumentation dynamique par rapport à l'instrumentation du binaire exécutable est que le programme n'est pas modifié. Il est donc très simple de l'exécuter en mode tracé ou non tracé sans aucune recompilation, ce qui est très utile durant la phase de mise au point.

De même que pour l'instrumentation du binaire, il n'est pas nécessaire de disposer du code source. En contrepartie, il est plus difficile d'obtenir des informations issues d'objets de haut niveau (structures ou classes complexes). Pour cela, il faut mettre en place un mécanisme de reconstruction à partir des informations de bas niveau [Ott01], ce qui peut se révéler complexe.

### III.3.4 Utilisation d'une bibliothèque instrumentée

Cela consiste à lier le code du programme, lors de l'édition de liens, à une version instrumentée d'une bibliothèque utilisée par le programmeur. Ainsi, on utilise une

bibliothèque de communication telle que MPI sous une forme instrumentée (ex : VampirTrace ou XMPI). Le code chargé de la collecte des informations est fourni par la bibliothèque. L'instrumentation est, dans ce cas, assurée par le concepteur de la bibliothèque et l'utilisateur doit se satisfaire des points de trace existants. Il peut éventuellement en masquer certains lors de l'exécution du programme mais ne peut en rajouter.

La liaison d'une bibliothèque instrumentée à un programme se fait lors de l'édition de liens. Il faut donc disposer des fichiers objets du programme concerné. Ceci est plus restrictif que de disposer du binaire, mais moins que du code source.

Il a fallu évidement instrumenter la bibliothèque utilisée. Cela est fait par le programmeur de celle-ci, ce qui limite les difficultés pour l'utilisateur final. Par contre, cela peut poser des problèmes pour obtenir des observation de différents niveaux, il ne sera pas toujours simple d'utiliser plusieurs bibliothèques instrumentées de manière non cohérente.

### III.3.5 Instrumentation du code source

L'instrumentation du code source consiste à placer, aux points de trace du code du programme à observer, des appels aux fonctions d'enregistrement. Pour cela, il faut disposer du code source du programme et être en mesure de le recompiler, ce qui peut poser problème avec les applications propriétaires.

L'avantage majeur de l'instrumentation du code source est de pouvoir accéder à toutes les informations que gère le concepteur de l'application. Il est ainsi possible d'enregistrer des structures complexes qui ne seraient pas facilement interprétables à un niveau plus bas. L'instrumentation au niveau du code source permet également d'être relativement indépendant vis-à-vis de la plateforme utilisée. Il suffit de respecter la syntaxe du langage de programmation utilisé. Le fait de placer les instructions d'enregistrement au niveau du code source permet d'optimiser au maximum les opérations de sauvegarde des données. Par l'utilisation de macro-commandes ou de fonctions « *inline* » on évite les coûteux changements de contexte. Parmi les contraintes, il faut souligner la nécessité de recompiler le programme.

L'instrumentation du code source d'un programme peut être effectuée manuellement ou automatiquement.

#### III.3.5.1 Instrumentation automatique

Contrairement à l'instrumentation manuelle, l'instrumentation automatique se charge de parcourir le code source d'un programme pour le réécrire en y plaçant des appels aux fonctions d'enregistrement. L'avantage est évidement d'avoir un processus bien plus rapide et simple pour l'utilisateur. Les inconvénients viennent de

la manière de spécifier les fonctions à tracer : soit on instrumente toutes les fonctions d'un certain type avec une énumération en intention, soit on procède à une énumération en extension. Dans le premier cas, il n'est pas possible de choisir les fonctions nécessitant ou non une instrumentation ; dans le second cas, on revient à une énumération s'approchant de l'instrumentation manuelle, ce qui en réduit l'intérêt.

#### III.3.5.2 Instrumentation manuelle

L'instrumentation manuelle offre une plus grande liberté d'action à l'utilisateur. Il lui est possible de placer des points de trace où il le désire et d'adapter les paramètres enregistrés sans être dépendant d'un outil d'instrumentation automatique plus rigide. L'instrumentation manuelle est également la plus simple à mettre en œuvre puisque c'est à l'utilisateur de placer dans son code les appels aux fonctions d'enregistrement. Il doit éventuellement spécifier les arguments des fonctions pour sauver les paramètres qu'il désire. L'inconvénient de l'instrumentation manuelle est d'être fastidieuse si le code est complexe.

L'utilisateur place dans son code une instruction du type :

```
...
code utilisateur
FONCTION_D_ENREGISTREMENT (Variable1, Variable2, Variable3);
code utilisateur
...
```

Chaque fonction d'enregistrement doit évidement être implantée pour enregistrer correctement les paramètres. Une fois compilé et lié à la bibliothèque de fonctions de traçage, l'exécution du programme engendre les fichiers de traces. En contrepartie, il peut être fastidieux d'instrumenter un code dont on n'a pas la maîtrise.

Une approche intéressante est de fournir des outils d'aide à l'instrumentation, par exemple, *via* une interface graphique affichant le code du programme. L'utilisateur sélectionne un point de trace et un type d'événement. L'insertion de l'appel de fonction est automatique, et il ne lui reste plus qu'à définir les paramètres à enregistrer.

## III.4 Fonctions d'enregistrement

Les mécanismes d'enregistrement font appel à des techniques connues pour réduire les surcoûts engendrés. Ces surcoûts sont essentiellement dus au temps d'exé-

cution du code du mécanisme d'enregistrement, et au temps passé dans des appels bloquants pour l'enregistrement sur disque.

### III.4.1 Rôle des fonctions d'enregistrement

Lors du traçage événementiel, les fonctions d'enregistrement sont chargées de récupérer les données utiles à la reconstitution du déroulement du programme observé. Pour cela, il faut enregistrer pour chaque événement sa date, son lieu d'occurrence et les informations associées. Le principe généralement utilisé est de placer ces informations dans des *tampons mémoire* avant de les enregistrer sur un support de stockage local ou distant.

### III.4.2 Efficacité de l'enregistrement

Afin de limiter l'intrusion due à l'observation, il est important de minimiser le coût des fonctions de trace et des mécanismes d'enregistrement de données. Les principales causes de perturbation viennent du blocage des processus lors de l'écriture des données récoltées et des surcoûts de changement de contexte lors des appels aux fonctions d'enregistrement.

Pour limiter les blocages inhérents aux écritures sur disque, il est classique d'utiliser des *tampons mémoire* (*buffer*) dans lesquels sont enregistrées les données avant leur écriture. On diminue ainsi les coûts d'enregistrement superflus dus à l'écriture de petites quantités de données. Dans un environnement multi-*thread*, il est nécessaire de protéger par un verrou l'accès au tampon dans lequel les données sont écrites pour éviter des accès concurrents et des écritures incohérentes. Cette manière de procéder pose de sérieux problèmes de performance car le nombre de copies de données peut être important et entraîner des blocages fréquents de *threads*, ce qui perturbe très fortement le comportement du programme. Pour éviter cela, nous utilisons un tampon mémoire par *thread*, ce qui permet d'éviter l'utilisation d'un verrou.

Pour limiter le surcoût d'appel des fonctions d'enregistrement, nous utilisons des macro-commandes. Chaque fonction d'enregistrement est définie comme une macro-commande. Celle-ci est expansée avant la compilation par le préprocesseur pour produire le code nécessaire à l'enregistrement aux emplacements des points de trace. On évite ainsi un appel de fonction et le coût des changements de contexte.

La mise en œuvre de ces mécanismes est classique et effective dans tous les programmes d'enregistrement de données. Il est cependant délicat de rendre ces mécanismes flexibles car l'utilisation de branchements conditionnels risque de réintroduire les surcoûts qu'on a cherché à éliminer.

### III.4.3 Activation / désactivation du traçage

Quand on instrumente le code source, il est parfois utile ou nécessaire de désactiver définitivement ou temporairement le traçage. Si l'utilisateur désire effectuer des mesures de performance par d'autres méthodes que le traçage, il faut supprimer les perturbations imputables au traçage. Il est également courant de ne vouloir observer qu'une partie d'un programme ; il faut pour cela pouvoir désactiver une partie des fonctions d'enregistrement.

Traditionnellement, il est possible de désactiver les systèmes de traçage à plusieurs moments :

**À la compilation par des directives de compilation** cela consiste à inclure le code à l'intérieur de directives de compilation conditionnelles :

```
#ifdef USE_TRACER
... code pour l'enregistrement ...
#endif
```

On élimine ainsi totalement les surcoûts liés au traçage si l'évaluation de la directive de compilation est négative. Il est par contre nécessaire de recompiler le programme observé.

**À l'exécution** Cela consiste à placer le code de prise de trace à l'intérieur d'un test :

```
if (UseTracer)
   { ... code pour l'enregistrement ...}
```

Il n'est alors pas nécessaire de recompiler le programme. Le choix d'enregistrer ou non peut être précisé au lancement du programme ou même durant l'exécution. Même avec les traces désactivées, il reste le coût de l'appel au test. Bien que celui-ci soit faible, il peut être de trop dans le cas de programmes cherchant une performance maximale.

La solution que nous avons adoptée est un mélange des techniques précédentes. nous combinons les méthodes par directive de compilation et de test à l'exécution. L'instrumentation d'un événement a la forme suivante :

```
#ifdef USE_TRACER          // test à la compilation
NOM_MACRO (parametres)     // macro d'enregistrement
#endif
```

La macro NOM_MACRO correspond à l'enregistrement d'un événement préalablement défini. Elle est définie de la manière suivante :

```
#define NOM_MACRO(parametres)\
if(!( TRACER_msk || NOM_NIVEAU_msk || NOM_MACRO_msk))\
  {
    enregistrement(parametres);
  }
```

La macro est expansée en un test contrôlant l'appel à la fonction d'enregistrement. Le test est effectué sur une combinaison de *masques*. Chacun des trois masques permet de désactiver les points de trace d'un événement particulier, de tout un niveau ou de la totalité des points de trace. Les niveaux permettent de grouper de manière logique un ensemble d'événements (voir section VI.2). Le coût du test existe toujours (quelques instructions), mais il est bien inférieur à celui d'un appel de fonction (changement de contexte).

Il est, de cette manière, possible de supprimer totalement les points de trace à la compilation à l'aide des directives de compilation. Il est également possible de désactiver l'enregistrement des données à l'exécution en positionnant de manière adéquate les masques correspondants. Ce système de masques permet un contrôle très fin des événements dont l'enregistrement doit être désactivé.

## III.5 Conclusion

Au regard des contraintes liées à l'observation pour le débogage de performances des intergiciels développés au laboratoire ID-Imag, nous avons choisi de nous baser sur l'observation comportementale qui, seule, permet une compréhension fine du déroulement d'un programme parallèle. Le but est d'identifier les causes de baisse de performance, les conditions dans lesquelles elles se produisent et les raisons de leur apparition. Afin de garantir un maximum de souplesse d'utilisation, nous avons basé nos travaux sur le traçage logiciel pour la collecte de données et sur l'instrumentation manuelle du code source. Nous laissons ainsi le contrôle total des points de trace aux utilisateurs. Les utilisateurs étant principalement les concepteurs d'intergiciels, ils sont les plus aptes à effectuer l'instrumentation de leur code.

Nous allons voir au chapitre suivant comment peuvent être exploitées les informations collectées au cours du traçage.

Chapitre

# Interprétation des observations

IV

*« L'interprétation n'a pas plus à être vraie que fausse ; elle a à être juste. »*
Jacques Lacan *C'est à la lecture de Freud...*

La collecte des informations relatives au déroulement d'un programme est la première phase du processus d'observation. Il faut ensuite pouvoir les exploiter efficacement. Notre objectif est l'élimination des problèmes de conception entraînant des baisses de performance. Nous cherchons à fournir des outils d'aide à leur identification puis à leur éradication.

Certains outils (Kojak [MW03, Koj03], DeepStart [RM02]) recherchent automatiquement les situations susceptibles de relever de problèmes de performance. Pour que des erreurs puissent ainsi être détectées, il faut qu'elles suivent un motif connu. Il n'est donc possible de reconnaître et d'analyser automatiquement que des types de problèmes connus et formalisés. Pour ces raisons, nous préférons nous orienter vers une *aide* à la détection de ces erreurs. Nous ne cherchons pas à automatiser la détection des erreurs, mais à guider l'utilisateur dans l'analyse des problèmes de performance par une *représentation* adéquate du comportement du programme.

Les représentations possibles des données enregistrées au cours de l'exécution sont nombreuses : textuelles, graphiques en 2 ou 3 dimensions, sonores ou en réalité virtuelle. Chacune de ces méthodes présente des avantages et des inconvénients que nous allons exposer. Nous allons auparavant expliciter le rôle des outils de représentation ainsi que les caractéristiques principales que nous cherchons à développer.

## IV.1 Rôle des outils de représentation

Par définition, dans un programme parallèle, contrairement à un programme séquentiel où les instructions s'exécutent les unes à la suite des autres, des opérations sont effectuées simultanément. Cela pose non seulement des problèmes relatifs à la programmation mais aussi à la représentation du déroulement du programme.

– Il est indispensable d'avoir une perception claire de l'*ordre* dans lequel les

actions se sont déroulées.
- Il est difficile d'interpréter collectivement un *très grand nombre* d'informations fournies sous forme textuelle. Une représentation textuelle sera réservée à des techniques d'analyse.
- Il est important de pouvoir mettre en correspondance l'occurrence d'un événement avec sa ou ses causes. Cela peut être difficile en raison du grand nombre de flots d'exécution simultanés.

C'est pour ces raisons que la plupart des logiciels d'observation offrent une représentation *graphique* du déroulement d'un programme permettant de fournir une vision compréhensible du grand nombre de données générées. Ces représentations graphiques jouent sur des variations de formes, de tailles et de couleurs d'objets géométriques pour offrir le maximum d'informations à l'utilisateur sous la forme la plus simple à comprendre possible. Le but de ces outils n'est pas de mettre en évidence des phénomènes connus, mais de faciliter la recherche de comportements pathologiques [Mil93].

Tous les modes de représentation poursuivent les deux objectifs antagonistes suivants : maximiser la quantité d'informations exposées tout en simplifiant la compréhension de l'ensemble des données. L'utilisation d'outils de représentation de données a divers objectifs. Leur rôle et leur fonctionnement sont souvent liés à la méthode de collecte de données utilisée. Une collecte d'informations effectuée par profilage fournit une représentation globale du déroulement du programme observé. De telles informations peuvent efficacement aider à identifier les zones où les programmes consomment du temps, qui sont donc a optimiser en priorité. Le traçage, quant à lui, permet la représentation complète du comportement de l'application. Il est possible de distinguer les différentes phases d'activité du programme et ainsi de mieux comprendre la manière dont se déroule l'application.

La visualisation nous intéresse principalement dans le cadre de l'optimisation de performances. Les principaux problèmes viennent des interactions entre processus dont on ne peut que difficilement prévoir avec exactitude les évolutions. Les différents modes de présentation liés à une observation détaillée cherchent à faciliter la compréhension des interactions entre les processus concurrents. Il est évidement possible d'utiliser des outils d'observation pour le débogage pour la correction. La représentation d'une exécution permet en effet de conceptualiser plus aisément la manière dont s'exécute un programme et donc de voir où peuvent apparaître des problèmes d'ordre fonctionnel (inter-blocage, attente d'un processus ou communications anormales, etc.). Connaissant cela, il est plus aisé de corriger les éventuelles erreurs de programmation.

## IV.2 Caractéristiques des techniques de représentation de données

Afin de remplir leur rôle, les outils de représentation doivent satisfaire certains critères permettant une bonne exploitation des données. Il est souhaitable que les outils offrent le maximum de ces critères, mais il est souvent difficile de les combiner. Nous présentons ici quatre caractéristiques qui nous paraissent fondamentales.

### IV.2.1 « Scalabilité »

Pour un programme de calcul hautes performances, la « scalabilité » est la capacité à tirer pleinement parti des ressources qu'on lui alloue. Pour un outil d'observation, cette propriété signifie que l'outil est apte à traiter de grosses quantités de données issues d'un grand nombre de nœuds de calcul et cela sur une longue durée d'exécution. Un outil parfaitement « scalable », aura le même comportement pour l'observation d'une exécution sur un petit nombre de nœuds que sur un grand nombre.

On peut distinguer deux aspects de la « scalabilité » d'un outil de représentation de données : celui lié à l'efficacité du logiciel, et celui lié à la manière de représenter les informations.

L'efficacité du logiciel est importante pour permettre une utilisation interactive et confortable. Si les opérations d'affichage prennent un temps prohibitif, il sera extrêmement malaisé d'utiliser l'outil.

Pour la représentation, la « scalabilité » résulte de la capacité à présenter les informations d'une manière compréhensible par l'utilisateur quel que soit le temps d'exécution et le nombre de nœuds de calcul ou de processus impliqués dans la trace. Afin de parvenir à ce résultat, les outils d'observation doivent utiliser des mécanismes pour limiter la quantité de données présentées. Il ne faut en effet pas surcharger l'utilisateur d'informations . L'affichage doit cependant rester pertinent et refléter ou donner accès aux informations importantes. Nous verrons à la section V.2.3 quelques techniques utilisées pour favoriser la « scalabilité ».

### IV.2.2 Interactivité

Les interactions possibles entre un utilisateur et un outil de présentation permettant de modifier le comportement de celui-ci et de « naviguer » au travers des données

présentées constituent un aspect important de la recherche de problèmes de performance. Pouvoir par exemple masquer une partie des informations non pertinentes permet de clarifier une représentation. Obtenir une représentation sous différentes vues des mêmes informations permet souvent de mieux comprendre les situations auxquelles l'utilisateur est confronté.

La plupart des outils disposant d'une représentation graphique offrent des mécanismes permettant d'obtenir des informations détaillées sur un objet ou une représentation différente (histogramme, diagramme circulaire ou encore diagramme de Gantt) des données.

Si les mécanismes de modification nécessitent une re-configuration ou une réinitialisation du logiciel de présentation, le confort d'utilisation est fortement réduit. Il faut que l'outil de présentation soit interactif et que son fonctionnement soit le plus intuitif possible pour que l'utilisateur ne soit pas contraint à un mode de fonctionnement rigide. L'introduction de l'interactivité dans les outils nécessite une attention particulière lors de leur implantation. Comme ces outils sont amenés à traiter de grandes quantités de données, il faut utiliser des algorithmes et des structures de données efficaces pour assurer une bonne réactivité.

### IV.2.3 Observation multi-niveaux

La plupart des applications, aussi bien séquentielles que parallèles, se basent sur un empilement de *couches d'abstraction logicielles* pour faciliter la conception et favoriser la réutilisation du code. Pour un programme parallèle, on peut typiquement énumérer les couches suivantes :

- système d'exploitation ;
- couche de gestion de processus légers ;
- bibliothèque de communication optimisée ;
- langage de programmation parallèle ;
- application.

Les performances des applications parallèles dépendent d'un grand nombre de facteurs logiciels et matériels. S'il est difficile de profiter de la puissance des architectures parallèles, cela est dû en partie à la grande complexité des interactions entre les différentes couches logicielles et le matériel. Il est donc important de disposer d'informations relatives à chacune de ces couches logicielles pour corréler des données issues de niveaux différents. Cela permet de mieux comprendre les causes d'un problème ou les conséquences d'une action [Ott01]. Pour rendre compte du fonctionnement d'une application conçue sur la base de telles couches, il faut pouvoir effectuer une observation *multi-niveaux*. Le principe est de présenter à l'utilisateur des représentations simultanées issues de différents niveaux d'abstraction.

La mise en corrélation d'informations d'une application (envoi de travail d'un maître à des esclaves) et des taux de saturation d'un routeur peut permettre de

trouver les causes des problèmes de performance dans la structure du programme (taille des travaux trop petite par rapport au coût des communications ou simultanéité des envois à tous les esclaves). Une fois ces causes déterminées, il est possible de tenter de les éliminer en établissant une politique de distribution du travail plus adaptée.

Pour fournir une visualisation des différents niveaux d'abstraction, il faut naturellement disposer d'informations les concernant. Il existe un grand nombre de systèmes pour acquérir des informations de chaque niveau possible, du registre processeur au langage de haut niveau en passant par les interfaces des bibliothèques de programmation :

- au niveau matériel : registre du processeur, informations de débit sur les interfaces réseau PAPI [Sey01], Vtune [VTu03] ;
- au niveau du système d'exploitation : `/proc`, GANGLIA [Mas], BWATCH [Rad], PCP [SGI] ;
- au niveau des bibliothèques de communication : bibliothèques instrumentées VAMPIR, XMPI ;
- au niveau applicatif : traçage logiciel. TAU [Mal], SVPABLO [DZR98]

Il est, par contre, peu fréquent de trouver des outils offrant une vue combinée ou globale de toutes ces données. Dans le cas d'un programme MPI, cela consiste par exemple à disposer d'informations issues du système d'exploitation, des communications de la bibliothèque MPI, et de l'application elle-même. À l'aide de toutes ces informations, on peut être à même de déterminer plus aisément les sources de problèmes potentiels. VAMPIR [NAW+96], par exemple, permet de mixer un traçage événementiel des opérations MPI et des informations des compteurs matériels *via* l'interface PAPI.

Des travaux ont par exemple été effectués pour le développement de l'observation multi-niveaux pour l'évaluation de performance d'applications Java distribuées (voir [Ott01]).

### IV.2.4 Flexibilité

La flexibilité d'un outil d'observation est sa capacité à pouvoir être utilisé dans différentes situations. Dans le contexte de nos travaux, nous désirons observer le comportement d'actions de diverses origines : matériel, système, intergiciels et applications. Nous devons également pouvoir utiliser différents modèles de programmation et donc adapter les outils d'observation en conséquence.

La programmation en couches logicielles entraîne un besoin d'observation multi-niveaux. Pour pouvoir collecter des informations à chaque niveau d'abstraction, il est important de pouvoir adapter le système d'observation, et en particulier de définir les éléments à observer dans le contexte du traçage. Pour l'exploitation, en aval de

la collecte de données, il faut pouvoir traiter de manière adaptée ces informations en fonction de leur provenance et de leur signification.

La flexibilité de l'observation passe par une configuration, à la fois du processus de collecte de données en fonction des informations à enregistrer et aussi des outils d'exploitation de ces données.

La mise en œuvre des mécanismes assurant une bonne flexibilité a un impact direct sur l'efficacité des mécanismes de représentation. Ils doivent prendre en compte un nombre plus important de possibilités d'affichage, ce qui limite les possibilités d'optimisation.

## IV.3 Représentation des données

Les formes de représentation de données sont nombreuses. Nous exposons ici les catégories les plus couramment utilisées dans le domaine de l'observation de programmes parallèles.

### IV.3.1 Lecture de traces sous forme textuelle

La forme de présentation la plus élémentaire pour interpréter les observations est la lecture des traces sous un format textuel. Le tableau IV.1 présente un exemple de trace sous forme de texte, générée à partir de notre outil de traçage. La représentation de traces utilisée par les outils TAU ou VAMPIR est tout à fait similaire. Il s'agit d'un simple décodage des valeurs des paramètres associés aux événements enregistrés.

**Tableau** IV.1 – Exemple de trace brute décodée

```
Level(2):Evt(0):Tid(16384):Sec(407):Usec(294931):MUTEX_INIT_0(mutex==0x80f4174)
Level(2):Evt(1):Tid(16384):Sec(407):Usec(294931):MUTEX_INIT_1(mutex==0x80f4174)
Level(3):Evt(0):Tid(16384):Sec(407):Usec(294932):CONDITION_INIT_0(condition==0x80f4190)
Level(3):Evt(1):Tid(16384):Sec(407):Usec(294933):CONDITION_INIT_1(condition==0x80f4190)
Level(2):Evt(2):Tid(16386):Sec(407):Usec(295161):MUTEX_LOCK_0(mutex==0x80f3f20)
Level(2):Evt(3):Tid(16386):Sec(407):Usec(295162):MUTEX_LOCK_1(mutex==0x80f3f20)
Level(1):Evt(5):Tid(16384):Sec(407):Usec(295195):PTH_CREATE_1( pthreadId==16386)
Level(2):Evt(0):Tid(16384):Sec(407):Usec(296074):MUTEX_INIT_0(mutex==0x80f8630)
Level(2):Evt(1):Tid(16384):Sec(407):Usec(296077):MUTEX_INIT_1(mutex==0x80f8630)
```

On constate qu'il est possible d'identifier de manière simple les caractéristiques de chaque enregistrement pris individuellement. Pour chaque enregistrement, on

distingue les paramètres suivants : numéro de niveau (`Level`) et d'événement (`Evt`), identificateur du *thread* (`Tid`), date d'occurrence en $s$ et $\mu s$ (`Sec, Usec`), nom de l'événement, ainsi que les divers paramètres associés (référence du *mutex* utilisé par exemple). Il est, par contre, extrêmement difficile de relier entre eux différents événements, pourtant liés, s'ils ne sont pas consécutifs.

Cette forme de représentation, qui semble rébarbative et complexe au premier abord, permet cependant l'utilisation d'une multitude d'outils de manipulation de texte et de langages de script familiers des programmeurs (`awk, grep, head, perl, sed, tail` [Hek97] pour ne citer que les plus courants). Cela permet de trouver aisément un événement particulier pour peu que l'on sache ce que l'on cherche. Par exemple, il est aisé de localiser tous les envois de messages à destination du nœud de calcul numéro 4.

Cette technique, encore largement utilisée pour le débogage d'applications séquentielles malgré le développement des débogueurs, est peu adaptée à une application parallèle de grande taille. Il est malaisé, avec une simple représentation textuelle, d'avoir une idée précise de l'ordre dans lequel des événements appartenant à des flots d'exécution différents ont eu lieu.

### IV.3.2 Indicateurs globaux

Les indicateurs globaux tels que les indicateurs de tendance collectés par SvPablo sont représentés en vis-à-vis du code source pour pouvoir immédiatement repérer la partie du code concernée et ainsi faciliter la correction.

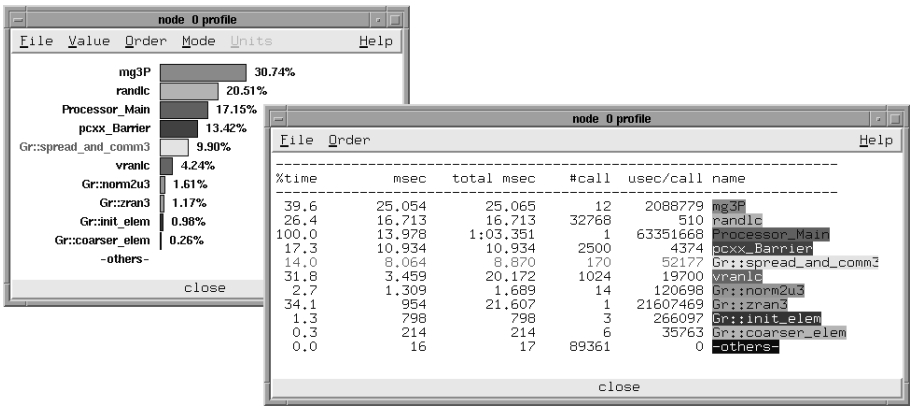



**Figure** IV.1 – Représentation, à l'aide de Racy, des résultats d'un profilage effectué à l'aide de Tau

Les informations issues de collectes par profilage peuvent être représentées sous forme de tableaux de valeurs, de diagrammes ou bien d'histogrammes. Racy (voir figure IV.1) offre une représentation de données issues de méthodes de profilage. Cette représentation est basée sur l'affichage des fonctions du programme observé et

de leurs informations de profilage. Cette représentation est donc globale et parfaitement « scalable ». Cependant une telle représentation est inadaptée pour permettre de comprendre le comportement d'un programme à un niveau très fin.

### IV.3.3 Représentation graphique 2D

Les représentations graphiques en 2D basées sur une approche événementielle sont les plus courantes car les plus intuitives pour les utilisateurs et les plus commodes à réaliser pour les concepteurs d'outils.

Dans le cadre des programmes parallèles, une présentation en 2D sous la forme d'un diagramme espace-temps (voir figure IV.2) permet d'identifier facilement l'ordre d'occurrence des événements en comparant leurs positions. Il en est de même pour l'identification des relations causales entre événements par la matérialisation de liens.

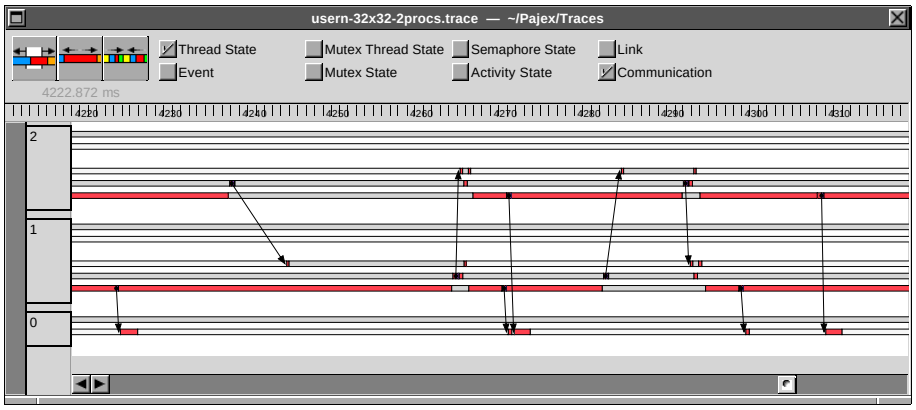



**Figure** IV.2 – Diagramme espace-temps de Pajé

De nombreux outils (Vampir [PAL], Pajé [dOS99], Xmpi [xmp], Upshot [ups]) utilisent des diagrammes espace-temps pour représenter le comportement d'un programme. Ils sont souvent complétés par d'autres formes de représentation comme des diagrammes circulaires ou des matrices de communication. Cette multiplication des possibilités offertes aux utilisateurs pour visualiser les données permet d'obtenir une vue selon laquelle les informations seront potentiellement plus simples à interpréter. Il sera, par exemple, bien plus facile de comprendre la répartition des communications entre les nœuds à l'aide d'une matrice de communication (voir figure IV.3) que d'un diagramme espace-temps.

Le principal problème auquel sont confrontés les outils de visualisation est celui de la profusion de données. Dans le cas du traçage d'un grand nombre de processus, on peut être en présence d'une quantité de données trop importante pour être représentée en intégralité. Il faut donc utiliser des moyens appropriés pour limiter le nombre des informations présentées simultanément à l'utilisateur. Ces mécanismes sont de type *agrégation* d'informations de même nature, *filtrage* d'éléments indési-

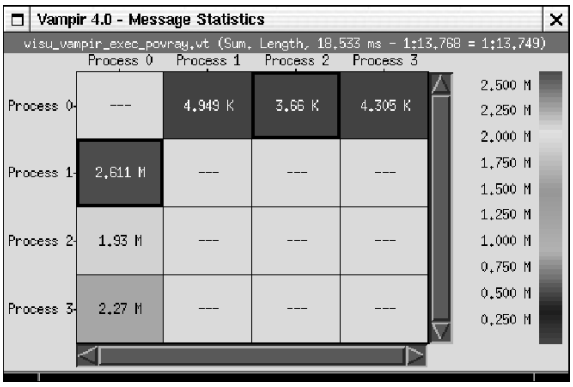



**Figure** IV.3 – Matrice de communication

Représentation des activités de communication d'une trace VAMPIR sous la forme d'une matrice.

rables ou encore réduction de la fenêtre temporelle d'observation (« *zoom* »). Nous aborderons plus en détail ces mécanismes à la section V.2 où leur mise en œuvre dans PAJÉ sera présentée.

### IV.3.4 Trois dimensions et réalité virtuelle

Des modèles de représentation graphique en 3 dimensions ont été développés. Leur but est de permettre un meilleur placement des informations afin de faciliter la compréhension des relations entre les objets représentés ou la navigation à travers les grandes quantités de données affichées.

Hackstadt et Malony, par exemple, [HM95] proposent des diagrammes de Kiviat en 3 dimensions. Il s'agit de représenter les nœuds de calcul, à un instant donné, sur un cercle Les communications se trouvent donc plus simples à représenter car elles ne chevauchent plus les nœuds extérieurs à la communication. L'évolution du temps est représentée par une suite de cercles formant un *tube* en 3 dimensions.

Afin de faciliter la navigation dans une représentation en 3 dimensions, qui se révèle peu intuitive lorsqu'elle est présentée sur un écran classique en 2 dimensions, des systèmes de réalité virtuelle [RSS+95] ont été employés. L'opération est effectuée à l'aide de dispositifs de projection vidéo sur un écran englobant l'utilisateur. Cela permet de rendre plus compréhensible une vue en trois dimensions en supprimant l'effort de reconstruction nécessaire dans le cas d'une représentation sur un écran (donc en 2D).

Malgré leurs nombreux intérêts, ces systèmes sont encore extrêmement marginaux en raison de leur coût et de la lourdeur de leur mise en œuvre.

### IV.3.5 Représentation sonore

Des essais ont été réalisés pour représenter sous forme de *sons* des événements survenant au cours de l'exécution de programmes parallèles. Ils ont pour objectif l'identification, entre autres, des phénomènes inhabituels dans le déroulement d'un programme. Madhyastha et Reed [MR95] ont ainsi proposé une bibliothèque de sonorisation de programmes de traces.

L'intérêt d'une représentation sonore est de permettre l'expression d'une grande diversité de caractéristiques par la modulation des paramètres constituant le son : *fréquence, vélocité, volume, timbre, rythme*, etc. Cette diversité offre de grandes possibilités de représentation d'informations, rapides à identifier pour un utilisateur, particulièrement si un changement intervient. Il est par exemple plus facile de distinguer un changement de sonorité que la longueur entre deux rectangles consécutifs dans un diagramme espace-temps.

Si l'utilisation exclusive de représentation sonore reste exploratoire, il est très probable que la combinaison d'informations visuelles et sonores pourra augmenter significativement l'expressivité des représentations hybrides qui en découleront. De plus, ces techniques sont bien plus légères à mettre en œuvre que celles de réalité virtuelle, notamment en raison de la très large diffusion des composants matériels de sonorisation à la quasi-totalité des ordinateurs.

## IV.4 Conclusion

Ce chapitre nous a permis de présenter le rôle joué par les outils de représentation d'informations au cours du processus d'observation. Les données issues d'exécutions de programmes parallèles sont particulièrement complexes et nécessitent, pour leur compréhension, des représentations permettant de mettre en évidence les interactions entre processus et ordonnancement des actions effectuées. Leur utilisation est fondamentale pour la simplification de l'interprétation des données collectées au cours d'une exécution.

L'efficacité de ces méthodes de représentation est principalement liée à leur caractère intuitif, leur interactivité et leur « scalabilité ». Une caractéristique nous intéressant particulièrement est leur flexibilité, c'est-à-dire leur capacité à pouvoir s'adapter à différentes situations d'observation, différents modèles de programmation et à pouvoir suivre l'évolution des techniques de programmation.

Pour ces raisons, nous avons basé nos travaux sur la représentation graphique d'exécutions de programme sous la forme d'un diagramme espace-temps à 2 dimensions. Celle-ci est la plus appropriée pour représenter le comportement d'un programme observé à travers les actions effectuées.

Chapitre

# Outils d'observation

V

*« Man must shape his tools lest they shape him.»*
Arthur Miller

Nous présentons dans ce chapitre comment les principaux outils d'observation abordent le problème du débogage pour les performances et les techniques d'observation utilisées. VAMPIR permet une observation comportementale de programmes utilisant une bibliothèque de communication MPI. TAU, quant à lui, utilise le profilage par chronométrage pour effectuer une observation globale. SVPABLO privilégie la « scalabilité » en proposant une observation globale basée sur des méthodes de comptage et de chronométrage. PARADYN se distingue des précédents outils par son mode d'instrumentation dynamique lors de l'exécution d'un programme.

Nous présentons ensuite plus en détail PAJÉ, un outil générique de visualisation d'exécutions de programmes parallèles pour lequel nous voulons concevoir un mécanisme de forniture de traces.

## V.1 Systèmes d'observation logiciels existants

### V.1.1 VAMPIR VAMPIRTRACE

VAMPIR [NAW+96] est un logiciel de visualisation d'exécutions de programmes parallèles (voir figure V.1) développé par la société Pallas GmbH [PAL]. Son objectif est la mise au point de programmes parallèles basés sur MPI. VAMPIR appartient aux outils d'observation comportementaux et offre une visualisation *post mortem*, à l'aide d'un diagramme espace-temps, de l'exécution d'un programme parallèle. Il est possible de disposer d'autres modes d'affichage pour visualiser des informations statistiques ou quantitatives telles que les volumes de communication entre nœuds de calcul ou le temps passé par un processus dans chacun des états considérés.

La bibliothèque VampirTrace est une interface instrumentée pour une bibliothèque mpi. Elle permet de générer les traces d'exécution au format Vampir.

L'utilisation de cette bibliothèque est plutôt simple, il suffit de déclarer son utilisation lors de l'édition de liens du code à observer. Lors de l'exécution du programme, un fichier de trace, qu'il faut ensuite relire avec Vampir, est généré. Ce mode de fonctionnement a l'avantage d'être simple à mettre en œuvre et de ne pas nécessiter toute la recompilation du code utilisateur.

Il est possible pour le programmeur de placer, dans son code source, des appels à des fonctions de VampirTrace. Ces appels permettent d'enregistrer des changements d'état des processus. On peut ainsi obtenir des informations sur le temps passé dans différentes parties du code utilisateur en plus des fonctions MPI.

VampirTrace retire de ce mode de fonctionnement certains avantages :
- une très bonne adaptation au traçage d'applications mpi ;
- une simplicité d'utilisation ;
- une grande robustesse.

Les principales limitations de Vampir viennent de son manque d'ouverture et d'adaptabilité. Vampir est conçu pour visualiser des exécutions de programmes mpi et il le fait bien. Mais il n'est pas possible de l'adapter facilement à différents modèles de programmation. Ceci est induit par un manque de flexibilité du couple Vampir/VampirTrace : il n'est pas possible de définir des événements utilisateurs de manière suffisamment souple.

Des essais menés avec VampirTrace ont permis d'optimiser l'exécution de la suite de tests Linpack [lin] utilisée comme test de référence pour le classement au «Top500» [Topb] de la grappe *icluster* du laboratoire ID.

Notre objectif étant de disposer d'un outil adaptable à différents modèles de programmation, il faut pouvoir facilement paramétrer l' outil pour s'adapter à une observation flexible et multi-niveaux, ce qui n'est pas le cas de Vampir et de VampirTrace.

### V.1.2 Tau

Tau [SMC+98, MSB+03, MS00, Mal] est un environnement de collecte de données pour l'analyse de performances. Il permet d'instrumenter, de mesurer, d'analyser et de représenter des informations de performance pour des applications parallèles à grande échelle. Il se veut ouvert et adaptable à un grand nombre de plateformes de programmation parallèle [SM01]. Ceci est conforté par le fait qu'il est un logiciel libre dont on peut consulter le code source et l'adapter.

Tau se base sur le chronométrage pour générer un profil d'exécution du programme observé. Il maintient au cours de l'exécution du programme instrumenté une structure de données par événement. Cela lui permet d'enregistrer pour chaque

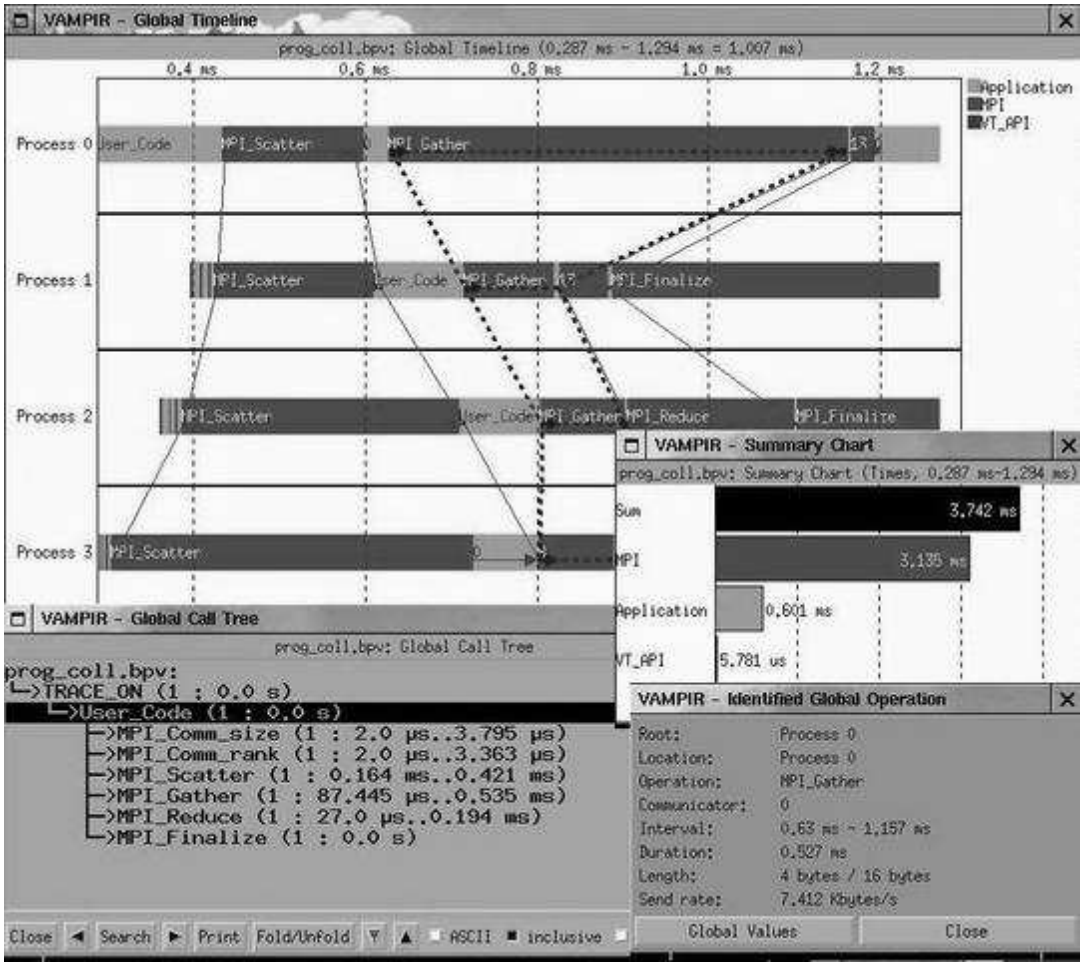



**Figure** V.1 – VAMPIR

La fenêtre principale de VAMPIR est un diagramme espace-temps qui permet de visualiser une exécution d'un programme parallèle. Elle s'accompagne de plusieurs autres vues permettant d'obtenir des statistiques sur le comportement de l'application observée. On peut distinguer sur cette figure : une hiérarchie des appels de fonctions, un récapitulatif du temps passé dans diverses zones du code (Sum, MPI, Application ou encore VT_API), et une description détaillée d'un événement affiché sur le diagramme espace-temps.

événement un certain nombre de paramètres de performance : comptage du nombre d'occurrences d'appels de fonctions ou chronométrage du temps passé dans une fonction par exemple. Parallèlement, la structure de la pile d'appel est gardée en mémoire, ce qui permet de déterminer le temps passé par le programme dans chaque partie et sous-partie du code, et ainsi de dresser un profil d'exécution du programme.

Un des points forts de cet outil se trouve dans son adaptation à la plupart des langages de programmation (C++, HPF, Java, etc.) et de plateformes utilisées en parallélisme. Il permet d'analyser des programmes basés sur des modèles de program-

mation hybrides (*MPI+Thread* ou *OpenMP+MPI*). De plus, la définition d'événements personnalisés à profiler est possible.

Étant conçu sur la base du comptage et du chronométrage, TAU peut être utilisé pour de longues exécutions sur un grand nombre de nœuds de calcul. Les informations sont synthétiques, la taille des données collectées ne sera donc pas un facteur prédominant dans les problèmes d'intrusion du processus de traçage. TAU assure ainsi une bonne « scalabilité », à la fois pour la collecte de données et pour leur représentation.

L'instrumentation d'un programme peut se faire manuellement dans le code source du programme à observer. Il est également possible d'utiliser un préprocesseur (`tau_instrumentor`) pour instrumenter automatiquement un programme C, C++ ou Fortran90. Il est possible de recourir aux bibliothèques PAPI [Sey01] et PCL [PCL] pour obtenir des informations issues des compteurs de performance des processeurs.

La visualisation des résultats peut se faire de manière textuelle (voir tableau V.1) ou par l'intermédiaire du programme *ParaProf* [BMS03] (voir figure V.2).

**Tableau** V.1 – Résultats TAU

```
NODE 0;CONTEXT 0;THREAD 1:
---------------------------------------------------------------------------------------
%Time  Exclusive    Inclusive  #Call  #Subrs  Inclusive   Std Name
            msec   total msec                usec/call   dev
---------------------------------------------------------------------------------------
100.0      0.043        5,005     1       1    5005331     0  threaded_func() int ()
100.0      5,005        5,005     1       1    5005288     0  work() int ()
  0.0      0.077        0.194     1       1        194     0  first() int ()
  0.0      0.028        0.117     1       1        117     0  second() int ()
  0.0      0.056        0.089     1       1         89     0  third() int ()
  0.0      0.033        0.033     1       0         33     0  fourth() int ()
---------------------------------------------------------------------------------------
```

Ces résultats sont une représentation textuelle simple du profil d'exécution dressé à partir des données collectées par TAU sur l'exécution d'un *thread*. Il est possible de voire que la quasi-totalité du temps a été passée dans la fonction `work()`.

TAU dispose d'un ensemble de primitives permettant le traçage d'applications et reposant sur les mêmes structures que celles de profilage (liées aux événements). On peut ainsi tracer l'exécution de certaines primitives MPI utilisées par un programme parallèle. Les traces produites par TAU peuvent être converties dans un format lisible par VAMPIR. Malheureusement, ce système de traçage est limité aux événements MPI prédéfinis. Il n'est pas encore possible de tracer les événements définis par l'utilisateur.

La configuration des observations réalisées par TAU ne concerne que l'aspect profilage, les événements tracés ne pouvant pas être définis, son utilisation est peu adaptée à l'instrumentation des intergiciels qui nous intéressent.

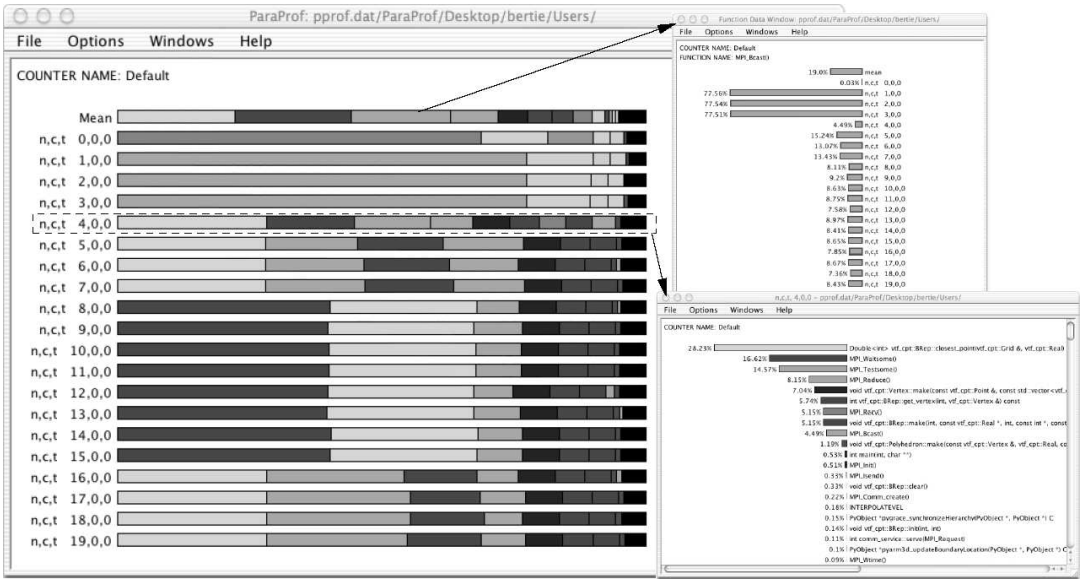



**Figure** V.2 – *ParaProf*

*ParaProf* permet de visualiser de manière simple le temps passé dans chaque zone de programme. Il affiche, pour chaque nœud d'exécution (à gauche de la figure), le temps passé dans les différentes procédures du programme (barres colorées au centre). Il permet également de détailler sous la forme d'histogrammes (figures de droite) les procédures les plus consommatrices de temps machine.

### V.1.3 SvPablo

SvPablo (*source view Pablo*) [DZR98] est un environnement graphique pour l'instrumentation de code source et la visualisation d'informations de performance. L'interface graphique permet de parcourir le code source d'un programme à observer et d'y placer des points de trace. Elle permet ensuite, après instrumentation des points de trace et exécution du programme, de visualiser les informations collectées.

Le principe d'observation utilisé par SvPablo est de conserver des statistiques concernant l'exécution de chaque événement instrumenté. Les informations enregistrées sont, par exemple, le nombre de fois où l'événement instrumenté a été exécuté ou bien le temps d'exécution cumulé. On peut également obtenir des indicateurs de tendance (minimum, maximum, moyenne, écart type) des valeurs comptées. Ces informations sont ensuite affichées en vis-à-vis des structures du code source dont elles sont issues (voir figure V.3). Cela permet de mettre en évidence les zones de code impliquées dans tel ou tel phénomène observé. Cette représentation synthétique (en parallèle avec le code source) est indépendante du nombre de nœuds d'exécution, elle est donc parfaitement « scalable ».

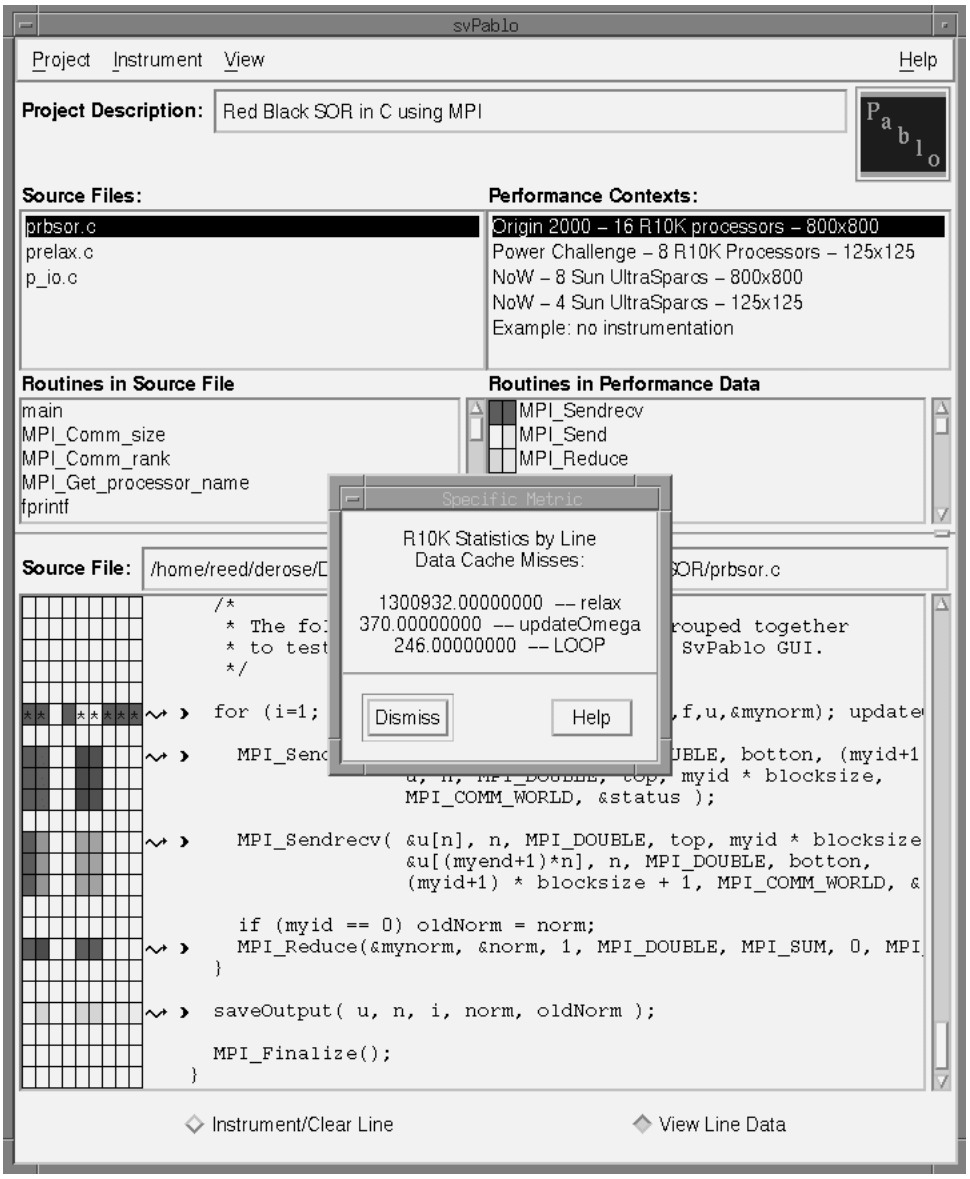



**Figure** V.3 – *SvPablo*

La fenêtre principale de SvPablo permet de placer les points d'instrumentation dans le code source à observer. Après exécution, des indicateurs de performance sont affichés sous la forme de carrés de couleur (à gauche du code).

De par son fonctionnement basé sur des statistiques, SvPablo permet une observation peu intrusive et une très bonne « scalabilité ». Peu de données doivent être enregistrées pendant l'exécution du programme à observer et l'intrusion due aux manipulations des données ou à leur transfert vers un serveur central est ainsi limitée. L'observation d'un grand nombre de nœuds de calcul n'engendre pas une augmentation importante de la taille des données récoltées, ce qui assure de garder

une bonne compréhension de celles-ci.

Les programmes C ou Fortran peuvent être instrumentés de manière interactive. Les programmes HPF (*High Performance Fortran*) peuvent être instrumentés automatiquement par le compilateur HPF de la compagnie PGI. SvPablo analyse chaque fichier source de l'application et marque les événements pouvant être instrumentés (boucles externes et fonctions). Une marque « > » s'affiche alors devant chaque événement instrumentable. L'utilisateur active ou non (apparition d'une flèche) chaque point de trace. SvPablo génère alors une nouvelle version des sources contenant les appels instrumentés aux endroits désirés. Il est possible de faire une sélection par fichier des points d'instrumentation, ou encore par fonction : on instrumente alors tous les appels à une fonction donnée.

SvPablo permet également un accès aux compteurs de performance (voir section III.2.3.1) directement sur MIPS R10000, ou en utilisant une bibliothèque spécialisée PAPI [Sey01]. Cela permet, par exemple, de mesurer des informations de bas niveau comme le nombre de défauts de cache du processeur.

SvPablo fonctionne sur une grande variété de plateformes séquentielles et parallèles. Il permet l'observation d'applications écrites en Fortran, C ou HPF sur les plates-formes Solaris, Irix, IBM SP, Alpha et Linux.

Un aspect intéressant de SvPablo est son format de fichier auto-défini SDDF (Self-Defining Data Format). Il permet une grande flexibilité d'utilisation en définissant lui-même le format des événements qui sont stockés dans une trace. Ce fonctionnement permet de séparer l'utilisation d'un langage particulier de la présentation des informations, ce qui va dans le sens d'une plus grande généricité.

SvPablo n'est cependant pas destiné à réaliser du traçage événementiel indispensable à l'observation comportementale. Son fonctionnement est, par contre, adapté dans le cas d'une observation à un niveau de granularité assez fin tout en restant apte à supporter des durées très grandes d'exécution.

### V.1.4 Xmpi

Certaines bibliothèques de programmation proposent des versions instrumentées ou un traceur adapté à leur usage.

Lam-mpi [Lam] propose une version instrumentée de la bibliothèque de communication au standard MPI et le logiciel de visualisation XMPI [xmp] correspondant. Ce système d'observation est intimement liée à la bibliothèque MPI utilisée. Il est donc possible d'obtenir toutes les informations nécessaires pour observer les actions MPI effectuées. Il est cependant bien plus complexe d'instrumenter d'autres parties de code avec ce système de traçage. De plus, la visualisation produite par XMPI ne repose pas sur un simulateur complet, ce qui limite grandement son interactivité : la méthode d'observation de XMPI est de dérouler les événements de l'application

sans possibilité de retour en arrière par exemple.

## V.2 PAJÉ

Les outils d'observation présentés sont soit axés vers un mode d'observation global, soit fortement liés à un modèle de programmation et peu adaptables. Il n'existait pas d'outil de visualisation pour l'observation comportementale lors des premiers travaux sur l'observation des intergiciels et des applications développés au laboratoire ID-IMAG. Aucun de ceux qui existaient n'offrait les principales caractéristiques recherchées : la « scalabilité » et la possibilité de représenter, à la fois, des processus communicants par envoi de messages et des *threads*. Afin de combler ce manque, l'outil de visualisation graphique PAJÉ [dOS99] [CdKdOS00a] a été réalisé par Benhur Stein.

PAJÉ a été conçu dans le but de visualiser des traces de programmes ATHAPASCAN. ATHAPASCAN était composé à l'époque de deux couches logicielles A1 et A0. La couche A1 offre une interface C++ de programmation à parallélisme explicite par tâches concurrentes. La couche inférieure A0 proposait des mécanismes de création et de gestion de *threads* et de communication. La couche A0 a aujourd'hui été remplacée par INUKTITUT. PAJÉ offrait donc la possibilité de visualiser des exécutions de programmes ATHAPASCAN grâce à un modèle de visualisation basé sur des objets représentant des processus, des *threads*, des communications et d'autres objets de synchronisation : verrous, sémaphores, etc.

Le but de PAJÉ est la recherche de problèmes de performance. Afin de remplir ce rôle correctement, il dispose des principales caractéristiques définies au chapitre précédent. Il est interactif [CdKdOSB00] et permet de « naviguer » au sein des informations affichées pour identifier les zones présentant des problèmes de performance. Il offre des mécanismes permettant une bonne « scalabilité » de la représentation. Sa généricité, c'est-à-dire la possibilité de représenter des informations définies par l'utilisateur, lui offre la possibilité de visualisation multi-niveaux et une très grande flexibilité. De plus, PAJÉ est conçu de manière *modulaire* [CdKdOS00b] pour qu'on puisse lui adjoindre de nouvelles fonctions.

Nous allons d'abord expliquer comment procède PAJÉ pour afficher la représentation d'une exécution puis présenter brièvement les mécanismes sous-jacents à l'interactivité, la « scalabilité » et la généricité.

### V.2.1 Affichage / Visualisation

PAJÉ affiche de manière graphique, sous la forme d'un diagramme espace-temps à deux dimensions (fig. V.5), les états des objets impliqués dans l'exécution du programme observé. Pour représenter ces états, PAJÉ se base sur l'affichage *d'entités*,

des composants graphiques élémentaires dont les caractéristiques peuvent varier. Ces caractéristiques sont la forme, la couleur, ou encore la taille. Les composants actuellement disponibles sont des cercles, des triangles, des rectangles, des flèches et des courbes (voir figure V.4).

Par exemple, la représentation de l'état d'un *thread* se fait par une entité de la forme d'un rectangle dont la couleur reflète son état : rouge pour bloqué, vert pour actif. La valeur d'une variable dont l'évolution est instructive pour la compréhension d'un phénomène est avantageusement représentée par une courbe. Une entité peut être représentative de plus d'un enregistrement. Par exemple, une flèche symbolisant une communication est représentative de deux enregistrements : l'émission et la réception.

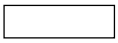
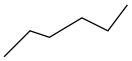
| | cercle | triangle | rectangle | flèche | courbe |
|---|---|---|---|---|---|
| **Forme** |  |  |  |  |  |
| **Catégorie** | événements instantanés | | événements non instantanés | relations | valeur |
| **Exemple** | prise de verrou | | état d'un thread | communication | charge CPU |

**Figure** V.4 – Entités graphiques utilisées pour composer une représentation graphique

Les entités peuvent être regroupées au sein de *conteneurs* . Les conteneurs peuvent eux-mêmes inclure de manière récursive d'autres conteneurs. Ils sont utilisés pour représenter des éléments de plus haut niveau que les entités. Ils peuvent être abstraits (processus) ou physiques (nœud de calcul). Ces conteneurs peuvent être manipulés par l'intermédiaire de l'interface graphique pour servir de support au filtrage ou à l'agrégation.

Une description hiérarchique permet de regrouper les entités sous une forme logique reflétant le modèle de programmation utilisé. Dans le cas d'Athapascan, une exécution est composée de plusieurs nœuds de calcul, chaque nœud fait s'exécuter plusieurs *threads* et chaque *thread* possède un état et des événements. La manière de décrire cette hiérarchie est détaillée à la section V.2.4.

La réalisation de l'affichage est possible grâce à un *simulateur*. Le rôle de celui-ci est de reconstruire le déroulement d'une exécution à partir de la trace enregistrée. Il va interpréter les événements de la trace pour reproduire les états des objets composant la représentation graphique. Son rôle est fondamental dans la mesure où nous voulons obtenir une représentation graphique dans laquelle l'utilisateur peut «naviguer» c'est-à-dire, par exemple, se déplacer le long de l'échelle de temps. Cette caractéristique n'est pas disponible dans des outils tels que XMPI [xmp] qui affiche les événements simplement en suivant leur chronologie.

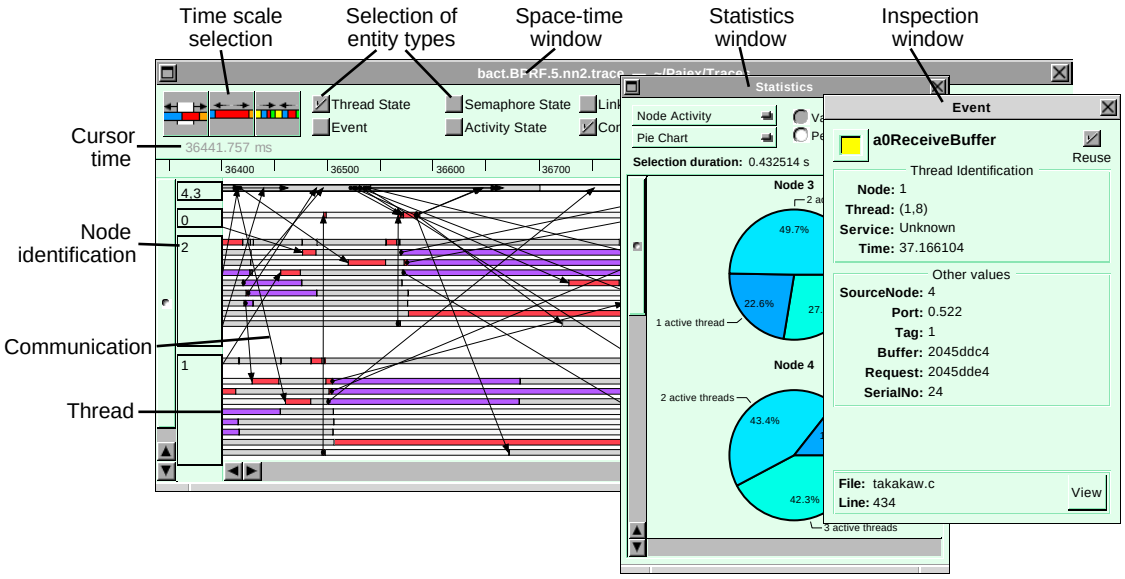



**Figure** V.5 – PAJÉ

Sur cette vue de PAJÉ, on distingue en arrière-plan la fenêtre principale du diagramme espace-temps représentant 5 nœuds de calcul dont 2 agrégés. Une fenêtre (au second plan) de statistiques liées à l'activité des *threads* et une fenêtre d'inspection donnant des informations détaillées sur une communication (au premier plan) peuvent être affichées par l'utilisateur.

## V.2.2 Interactivité

L'interactivité de PAJÉ s'exprime à travers les possibilités offertes pour obtenir des informations détaillées et modifier la représentation graphique d'une trace. Ces mécanismes sont accessibles dynamiquement, c'est-à-dire sans passer par une phase de configuration et de ré-initialisation de l'outil.

### V.2.2.1 Affichage des identifiants

Pour limiter la quantité d'information affichée, PAJÉ représente les objets par des formes géométriques simples : les *entités*. Les événements enregistrés dans la trace contiennent bien plus d'informations dont la représentation serait malaisée, parfois superflue et qui gênerait la lisibilité du diagramme. Chaque entité est incluse dans une hiérarchie d'objets et possède un identifiant et un nom. En faisant glisser le pointeur de la souris sur une entité, il est possible de faire apparaître ces informations dans la *barre d'information* (voir figure V.6) du haut du diagramme.

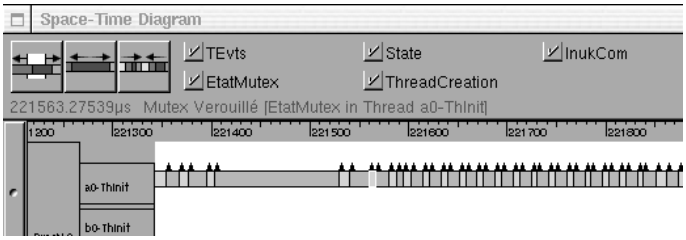



**Figure** V.6 – Détail de la barre d'information de PAJÉ

### V.2.2.2 Inspection

De la même manière, et afin de pouvoir accéder à des informations supplémentaires précises, l'utilisateur peut sélectionner une entité à l'aide du pointeur de la souris et obtenir l'affichage des informations liées à l'entité correspondante dans une fenêtre d'inspection (figureV.7). La fenêtre présente alors les dates concernant l'entité (début, fin, durée), la liste des types de la hiérarchie dans laquelle est inclus le type de l'entité détaillée ainsi que des informations supplémentaires s'il en existe.

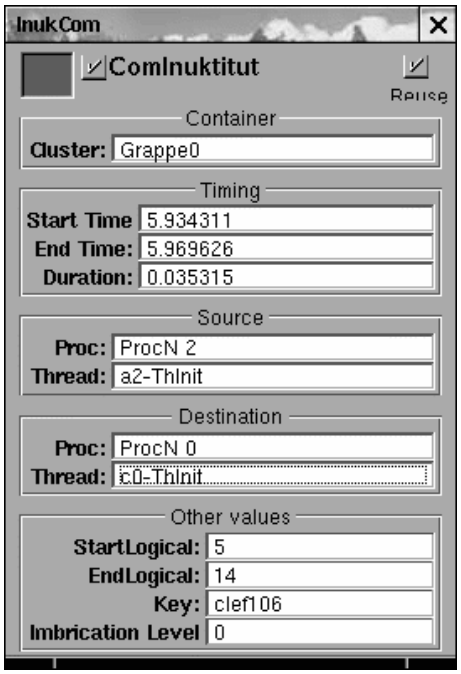



**Figure** V.7 – Fenêtre d'inspection d'une communication

La fenêtre d'inspection d'une communication affiche les détails de la source et du destinataire du message, le temps de transfert et toute autre information du ou des enregistrements des événements liés à l'objet inspecté.

L'interactivité concerne également les mécanismes utilisés pour garantir la « scalabilité » de la représentation graphique.

### V.2.3 « Scalabilité »

La « scalabilité » est la capacité à traiter un grand nombre d'objets et d'informations. Nous pouvons distinguer deux aspects importants de la « scalabilité » :

- les algorithmes et les structures de données utilisés pour assurer l'efficacité de PAJÉ sont exposés au chapitre 5 de [dOS99] ;
- la *« scalabilité » de la représentation* est obtenue par l'utilisation combinée de plusieurs mécanismes interactifs d'*inspection*, de *filtrage*, d'*agrégation* et de *«zoom»*.

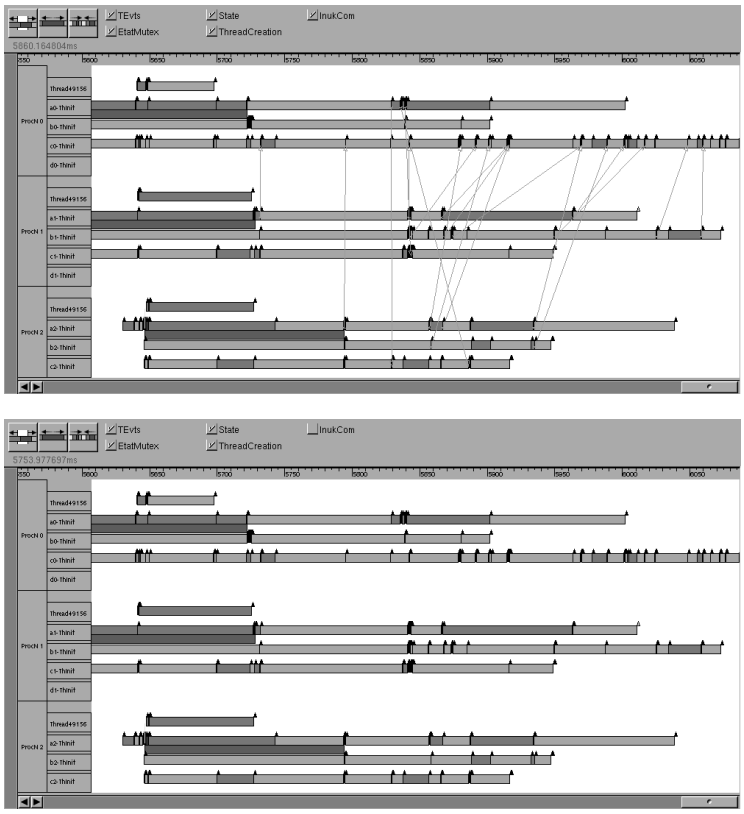



**Figure** V.8 – Filtrage de communications

Entre ces deux figures, les communications liées à INUKTITUT ont été masquées. Cela permet à un utilisateur de se concentrer sur les autres événements et les états des *threads*.

### V.2.3.1 Filtrage

Il existe de nombreuses informations que l'utilisateur peut juger non pertinentes à un instant de l'observation. Le *filtrage* permet de les supprimer temporairement. PAJÉ offre un mécanisme de filtrage interactif (voir figure V.8), c'est-à-dire qu'il est aisé de masquer ou d'afficher une catégorie d'objets.

### V.2.3.2 Agrégation

L'agrégation permet de grouper plusieurs objets en un seul pour simplifier la représentation d'une trace. L'utilisation typique est de regrouper les représentations des activités de processus pour obtenir une information synthétique d'un grand nombre de nœuds de calcul. L'utilisateur doit définir une *fonction d'agrégation*. Celle-ci est utilisée pour calculer les caractéristiques de l'état remplaçant les états agrégés. Cette fonction peut par exemple être une somme, une moyenne ou un dénombrement. Lors de l'étude de la visualisation, si un problème est observé à un niveau global (inactivité d'un groupe de processus par exemple), l'utilisateur peut dégrouper la vue pour examiner chaque processus de manière détaillée.

### V.2.3.3 « Zoom »

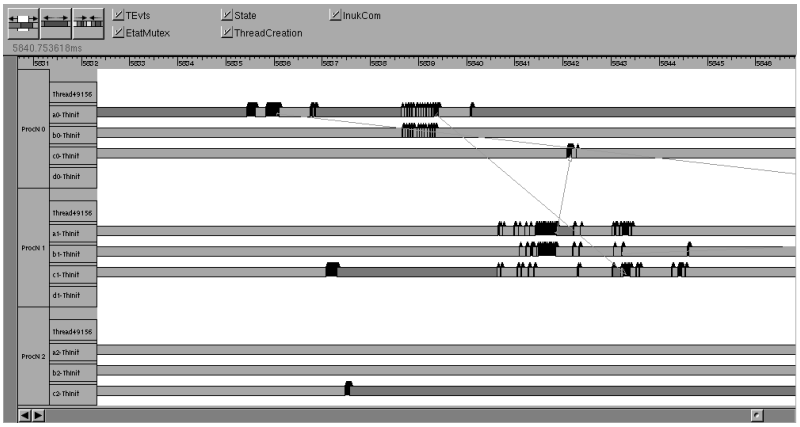



**Figure** V.9 – «Zoom»

Cette figure est une vue détaillée d'une portion de la même trace qu'à la figure V.8 située approximativement au centre des précédentes vues. On a réduit ici l'intervalle de temps observé approximativement d'un facteur 16.

Le zoom (voir figure V.9) permet de délimiter un intervalle de temps et de n'afficher que les événements contenus dans celui-ci. C'est l'opération de base pour

détailler une action précise noyée dans le comportement global d'une application. Il est important de noter que la réalisation du processus de zoom n'est pas triviale en raison des relations potentielles entre les événements inclus dans la fenêtre temporelle sélectionnée et les événements extérieurs. Il ne suffit pas de filtrer les événements extérieurs à un intervalle de temps, il faut également inclure les événements impliqués dans une action présente, même partiellement, dans l'intervalle de temps ciblé. Par exemple, si un message est reçu dans l'intervalle de temps délimité par le zoom, il est nécessaire de traiter également son émission, quand bien même celle-ci se trouverait en dehors de l'intervalle de temps.

Le rôle du simulateur est, ici aussi, prépondérant car c'est lui qui reconstruit la représentation de l'exécution à partir des traces, et qui permet de déterminer si des événements doivent apparaître ou non dans la vue courante.

### V.2.4 Généricité

Initialement PAJÉ a été conçu pour afficher des traces de programmes Athapascan en utilisant un simulateur propre à Athapascan. Au regard des différents modèles de programmation utilisés et du coût de développement d'un simulateur dédié, un *simulateur générique* a été conçu.

Pour être utilisé avec un modèle de programmation particulier, ce simulateur générique doit être *instancié*. Le principe est de définir un *modèle de visualisation* correspondant au modèle de programmation utilisé. Ce modèle va permettre de déterminer comment représenter graphiquement les événements de la trace. Il est défini comme une *hiérarchie de types d'entités*. Cette hiérarchie sera ensuite instanciée de manière à correspondre aux différentes abstractions (*threads*, nœuds, communications, etc.) apparaissant dans le programme observé. Il n'y a pas de signification associée *a priori* à chaque objet graphique. Cela implique que l'interprétation est laissée à l'utilisateur. Cet aspect de PAJÉ permet de modifier la visualisation et de l'adapter à différentes situations. Il est cependant pertinent de conserver des associations consensuelles et relativement intuitives pour l'utilisateur (utilisation d'une flèche pour représenter une communication ou une relation causale).

PAJÉ peut donc être utilisé pour représenter des données issues de différentes sources en leur associant une sémantique définie par l'utilisateur. L'apport de sémantique est fait par la représentation sous une forme graphique —convenue par l'utilisateur— des abstractions du modèle de programmation. Nous avons ainsi utilisé PAJÉ pour représenter des données venant de sources autres que les traces de programmes parallèles. Citons, par exemple, des résultats d'algorithmes d'ordonnancement ou encore des informations de surveillance pour la grappe du laboratoire.

Nous avons effectué des expérimentations [GCdKdOSA01] afin de montrer comment la généricité de PAJÉ peuvent être utilisées pour assurer la « scalabilité » de la

visualisation dans le cadre de la surveillance de grappes de processeurs.

La section suivante présente le format de trace utilisé par PAJÉ et plus particulièrement la manière de décrire la hiérarchie utilisée pour configurer le simulateur générique.

### V.2.5 Format PAJÉ

La généricité de PAJÉ au niveau graphique est basée sur l'utilisation d'entités élémentaires se combinant pour obtenir la représentation d'une exécution. Pour assurer une grande flexibilité, le format de trace doit pouvoir s'adapter à la description d'événements de natures différentes. Nous allons voir dans cette section comment est défini le modèle graphique et ce qu'apporte l'utilisation d'un format auto-défini pour la versatilité des traces que peut lire PAJÉ.

Les traces lues par PAJÉ se composent de différentes parties :
- la description du format des enregistrements utilisés ;
- la description d'un modèle graphique ;
- une instanciation du modèle graphique ;
- les événements proprement dits de la trace.

Chacune contient des informations nécessaires au processus de visualisation de la trace.

#### V.2.5.1 Description du format

Afin de s'affranchir des contraintes liées à un format de fichier strict, PAJÉ utilise un format de trace de type *auto-défini*. L'auto-définition consiste à fournir une définition des *instructions* composant la trace. Cette définition (voir exemple tableau V.2) permet une description du format des enregistrements de la trace. Elle peut être vue comme la définition des *types des événements* utilisés dans la suite de la trace. Il est donc possible d'adapter le format des enregistrements traités par PAJÉ aux contraintes d'observation de l'utilisateur.

D'un point de vue logique, cette description n'est pas reliée à une trace, mais à un ensemble de types d'événements ou à un modèle de programmation. Elle peut être séparée de la trace pour pouvoir être réutilisée. Elle est cependant pour l'instant incluse dans le fichier de trace pour des raisons de commodité (manipulation d'un unique fichier).

Dans cet exemple (voir tableau V.2), on définit une instruction nommée *PajeDefineContainerType* et numérotée «1». Celle-ci va servir à définir les types des conteneurs dont l'utilisateur pourra se servir par la suite. Cette instruction est utilisée dans la section de définition du modèle graphique (voir exemple tableau V.4 page 69).

**Tableau** V.2 – Exemple de définition d'une instruction PAJÉ

```
%EventDef       PajeDefineContainerType 1
%       NewType string
%       ContainerType   string
%       NewName string
%EndEventDef
```

Elle prend en paramètre trois chaînes de caractères (type `string`) représentant :
- un identifiant du nouveau type de conteneur créé ;
- le type du conteneur père ;
- un nom, plus explicite que l'identifiant, pour ce nouveau type de conteneur.

On définit de la même manière une série d'instructions classiques (voir tableau V.3) destinées à décrire les types, la hiérarchie et les événements utilisés pour la définition de la plupart des traces.

**Tableau** V.3 – Instructions standard de PAJÉ

| Nom | numéro |
|---|---|
| *PajeDefineContainerType* | 1 |
| *PajeDefineEventType* | 2 |
| *PajeDefineStateType* | 3 |
| *PajeDefineVariableType* | 4 |
| *PajeDefineLinkType* | 5 |
| *PajeDefineEntityValue* | 6 |

Liste des instructions utilisées pour définir le modèle graphique.

| Nom | numéro |
|---|---|
| *SetLimits* | 0 |
| *PajeCreateContainer* | 7 |
| *PajeDestroyContainer* | 8 |
| *PajeNewEvent* | 9 |
| *PajeSetState* | 10 |
| *PajePushState* | 11 |
| *PajePopState* | 12 |
| *PajeSetVariable* | 13 |
| *PajeAddVariable* | 14 |
| *PajeSubVariable* | 15 |
| *PajeStartLink* | 16 |
| *PajeEndLink* | 17 |

Liste des instructions pour l'instanciation du modèle graphique et la représentation des événements.

Les identifiants des types ou des définitions d'événements sont choisis les plus courts possible dans un but de concision pour alléger la description et pour réduire la taille du fichier de trace. Une description plus détaillée est donnée par les noms associés aux types des objets définis et elle est utilisée par PAJÉ pour l'inspection ou dans la barre d'information.

### V.2.5.2 Modèle de visualisation

Le format PAJÉ se base sur un langage de description simple pour définir une hiérarchie de types reflétant le modèle de visualisation utilisé.

La hiérarchie décrite permet de définir comment représenter les différents objets impliqués dans la trace (nœud de calcul, processeur, *thread*, etc.). Elle permet d'associer aux composants abstraits du programme à observer une représentation visuelle sous la forme d'entités. On adapte ainsi au modèle de programmation un *modèle de visualisation*.

Par exemple, voici la description d'une hiérarchie (voir tableau V.4) associée à un modèle de programmation hybride de type *thread + communications* utilisé pour instrumenter la bibliothèque Taktuk.

**Tableau** V.4 – Instructions définissant une hiérarchie de types

```
1 Prog     0                          "Programme"
1 Node     Prog                       "Node"
1 Thread   Node                       "Thread"
3 State    Thread                     "État du Thread"
6 Exec     State                      "Exécution"
6 Blocked  State                      "Bloqué"
5 Com      Prog    Thread  Thread     "TakLink"
6 TakCom   Com                        "ComTaktuk"
```

La première colonne détermine l'instruction à utiliser et la dernière définit le nom qui sera employé par PAJÉ pour désigner l'objet créé. Cette hiérarchie se compose de trois conteneurs (`Prog`, `Node` et `Thread`), d'une entité, (`State`) et d'un lien, (`Com`). L'état d'un *thread* possède 2 valeurs : `Blocked` et `Exec`. Le type de lien `Com` entre deux `Thread` au sein d'un programme est spécialisé pour définir les communications gérées par Taktuk.

### V.2.5.3 Instanciation de la hiérarchie

Ces instructions correspondent à la déclaration des objets impliqués dans la trace, c'est-à-dire, en pratique, aux événements de création des abstractions du modèle de programmation.

La première colonne représente l'instruction utilisée (7 = *PajeCreateContainer*) ; la seconde, les dates d'occurrence des événements correspondants. Les suivantes

**Tableau** V.5 – Instructions pour l'instanciation de la hiérarchie

```
7 0.00  P1    Prog    0     "Mon Programme"
7 0.00  N1    Node    P1    "Noeud 1"
7 0.00  N2    Node    P1    "Noeud 2"
7 0.21  T1-1  Thread  N1    "Thread 1"
7 0.22  T2-1  Thread  N1    "Thread 2"
7 0.23  T1-2  Thread  N2    "Thread 1"
7 0.24  T2-2  Thread  N2    "Thread 2"
```

définissent un identifiant d'objet, le type de l'objet et un nom plus explicite utilisé lors de la visualisation par PAJÉ. L'exemple présenté ici (tableau V.5) correspond à la création d'un conteneur programme nommé P1. Deux nœuds lui sont rattachés : N1 et N2. Chacun de ces nœuds gère 2 *threads*.

#### V.2.5.4 Événements de l'exécution

Les événements de la trace sont ensuite énumérés en utilisant les instructions définies précédemment. Ils représentent l'évolution des états des objets tracés. Il peut s'agir de changement d'état d'une entité, de l'occurrence d'un événement ponctuel ou d'une communication.

Voici un court exemple illustrant un envoi de message de la bibliothèque TAKTUK. La représentation graphique correspondante est donnée à la figure V.10 droite.

**Tableau** V.6 – Exemple d'instructions représentant des événements de la trace

```
10 0.98 State T1  Exec      // changement d'état du thread 1
10 1.01 State T2  Exec      // changement d'état du thread 2
10 2.34 State T1  Block     // changement d'état du thread 1
16 2.43 Com   P1  TakCom  T1  655      // envoi d'un message
17 2.63 Com   P1  TakCom  T2  655    // réception du message
10 2.91 State T2  Block     // changement d'état du thread 2
8  4.29 T2                     // destruction des conteneurs.
8  4.34 T1
8  4.34 P1
```

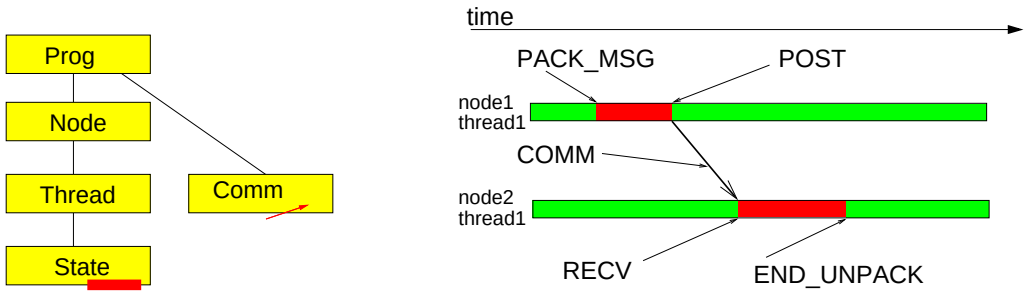



**Figure** V.10 – Hiérarchie de types et exemple de visualisation

Cet exemple très simple montre en parallèle une représentation de la hiérarchie de types décrite et utilisée dans un fichier de trace PAJÉ. On distingue les événements d'empaquetage, d'envoi, de réception et de déballage d'un message au cours d'une communication par *message actif* effectuée par la bibliothèque de communication TAKTUK.

## V.3 Conclusion

Ce chapitre a permis de présenter quelques outils d'observation existants. La plupart des outils d'observation de programmes parallèles sont fortement liés à des bibliothèques de programmation ou à des langages particuliers. Les outils multi-cibles effectuent généralement des observations globales et non comportementales.

La recherche d'un outil d'observation comportementale flexible pour des programmes basés sur différents modèles de programmation a conduit à la réalisation de PAJÉ, un outil de visualisation « scalable », interactif et générique. La flexibilité de PAJÉ se base essentiellement sur des mécanismes de configuration du format de trace utilisé[1], et sur la possibilité de configurer le modèle de visualisation à utiliser.

Si le problème de fournir une visualisation comportementale des modèles de programmation développés au laboratoire ID-IMAG semble résolu avec PAJÉ, il reste à pouvoir instrumenter simplement les applications et les intergiciels afin de récolter les traces permettant la visualisation. C'est l'objectif du travail décrit dans le chapitre suivant.

[1] Le format PAJÉ est décrit plus en détails dans le manuel de PAJÉ [dOSdK].

Chapitre

# Traçage flexible

VI

*«La souplesse et la capacité d'adaptation sont des outils indispensables pour faire face aux situations les plus difficiles.»*
anonyme

Dans le cadre de l'observation d'exécutions de programmes parallèles, nous avons besoin d'outils pour collecter des informations et les visualiser. Nous avons présenté PAJÉ, un outil de visualisation adaptable à de nombreux modèles de programmation. Pour le compléter et obtenir un environnement complet d'observation, adaptable, il manque un système de production de traces, tout aussi flexible. Un tel système doit pouvoir produire des traces compatibles avec PAJÉ, et être suffisamment souple d'utilisation pour s'adapter à différents modèles de programmation. Le problème auquel nous nous retrouvons confronté est d'arriver à concilier les caractéristiques nécessaires pour obtenir un système de traçage flexible sans pour autant compromettre son efficacité.

Dans ce chapitre, nous présentons les raisons nous amenant à proposer un système de traçage indépendant du système de visualisation, tant au niveau du fonctionnement que des formats de trace. Nous détaillons les différentes étapes nécessaires à sa mise en œuvre, de la configuration à l'exploitation des traces. Nous exposons les contraintes conduisant aux choix techniques faits pour assurer la flexibilité et l'efficacité de notre outil de traçage TUMIT[1].

## VI.1 Flexibilité du traçage

Il existe nombre de systèmes et d'environnements [Apa01] pour le débogage d'applications parallèles, tant pour la correction que pour les performances. Un défaut récurrent des outils existants est qu'ils sont souvent conçus en relation avec une

[1] TUMIT est un mot signifiant «trace» (d'un animal) en Inuit, le langage des esquimaux [Sch70].

bibliothèque ou un modèle de programmation particulier (VAMPIR avec MPI, XMPI avec LAM-MPI). Un frein majeur à l'utilisation de ces outils de mise au point vient de l'évolution très rapide des environnements de développement dans le monde du parallélisme. La durée de vie des outils spécialisés est souvent limitée par leur manque d'évolutivité. Il est courant de voir des outils abandonnés en même temps que les systèmes auxquels ils se rapportent. Cela engendre d'inutiles efforts de développement et d'apprentissage. Nous voulons donc développer des outils d'aide à la mise au point de programmes parallèles qui soient flexibles et donc réutilisables sans augmenter leur complexité d'utilisation.

Lors de la présentation de l'outil PAJÉ, nous avons détaillé les mécanismes lui conférant une grande flexibilité : la généricité du simulateur et l'utilisation d'objets graphiques élémentaires. Notre attention se porte maintenant sur le système de collecte de données nécessaire à la reconstruction de l'exécution d'un programme par un outil tel que PAJÉ. Nous avons examiné, au chapitre précédent, certains outils de collecte de données destinés à l'analyse de performances, mais leur flexibilité s'est révélée insuffisante pour notre usage. Cette caractéristique est importante pour permettre l'utilisation du système d'observation dans le cadre de différentes couches logicielles utilisées classiquement dans le domaine de la programmation d'applications parallèles.

Au lieu de proposer une bibliothèque dédiée [Vam] ou encore un programme de traçage intimement lié à une bibliothèque de programmation parallèle comme *A0-trace* [CdKDOW97], nous proposons de créer une sorte de *kit de traçage* modulaire pour aider les programmeurs à instrumenter aisément leurs programmes. Le principe est de laisser le programmeur instrumenter le code de son programme (application, intergiciel ou bibliothèque) en lui fournissant les fonctions dont il a besoin : gestion de la mémoire, enregistrement sur disque, etc.

La conviction de la nécessité d'un système de traçage indépendant d'un modèle de programmation particulier ou d'une bibliothèque spécialisée est venue des précédentes expériences effectuées au sein du laboratoire en matière de traçage. Il existait deux versions du logiciel Athapascan-0, une de développement, et une (A0-trace) instrumentée et maintenue séparément. Au fur et à mesure de l'évolution concurrente des deux programmes, il est devenu laborieux de les maintenir en parallèle de manière cohérente. En effet, un changement dans Athapascan-0 se répercutait dans A0-trace, et le moindre changement dans le mécanisme de traçage entraînait des modifications nombreuses dans A0. La conception d'un traceur lié à une bibliothèque engendre des contraintes qui le rendent fortement dépendant de l'environnement dans lequel il a été créé. Sa facilité d'utilisation ultérieure avec d'autres applications ou bibliothèques en est généralement compromise.

Afin d'arriver à offrir ces caractéristiques dans tous les cas de figure rencontrés par un programmeur, il est nécessaire de pouvoir configurer l'ensemble du processus d'observation. La flexibilité de l'outil de visualisation est assurée par la possibilité d'adapter la représentation ou l'utilisation de techniques d'analyse indépendantes

des modèles de programmation rencontrés. En ce qui concerne le traçage, l'adaptation passe par une configuration des événements de trace, et de la manière de les représenter lors de la visualisation. Les événements reflètent les objets impliqués dans l'exécution du programme à observer et, par conséquent, le modèle de programmation utilisé.

## VI.2 Définition des événements

L'adaptation du système de traçage à un modèle de programmation particulier passe par la définition des types des événements à enregistrer. Cette définition permet de décrire les noms des événements ainsi que les types et les noms des paramètres associés que l'on désire enregistrer.

L'utilisateur définit des événements sous une forme textuelle simple à la manière de prototypes de fonctions :

```
EVENT(EVENT_NAME, Type1 x, Type2 y, Type3 z)
```

Cet exemple déclare un événement de nom `EVENT_NAME` ayant trois paramètres nommés `x`, `y` et `z` de types respectivement `Type1`, `Type2` et `Type3`.

La connaissance des types des paramètres permet de connaître la taille des données à enregistrer. Ces tailles sont utilisées notamment pour assurer la bonne gestion des tampons mémoire utilisés par le traceur.

Chaque définition d'événement est faite dans le contexte d'un *niveau* — définit par un nom et un numéro — ce qui permet de regrouper des événements en ensembles logiques. Ces ensembles peuvent représenter un niveau d'abstraction logiciel pour l'observation multi-niveaux, mais aussi être un sous-ensemble d'un niveau pour regrouper les événements présentant une caractéristique commune. Par exemple, la ligne suivante définit un niveau nommé `PTH_LEVEL` et numéroté `1`.

exemple de niveau :
```
LEVEL(PTH_LEVEL, 1)
```

Ce niveau regroupe tous les événements relatifs à la gestion des *threads* par la bibliothèque Pthread (création, attente, rendez-vous, etc.). Les aspects liés aux primitives de synchronisation — Mutex et conditions — prennent place dans deux autres niveaux distincts.

Ces ensembles permettent de distinguer certains événements dans le but de faciliter les traitements ultérieurs. Par exemple, il est aisé de filtrer un ou plusieurs ensembles et ainsi d'isoler des groupes d'événements ayant des caractéristiques communes.

**Tableau** VI.1 – Exemple d'un extrait de définition d'événements
Cet exemple de définition d'événements est extrait de la bibliothèque de portabilité pour la gestion des *threads* dans INUKTITUT. On définit ici les événements liés à la gestion des *threads*.

```
#include <pth_pthreadsync.h>
// define the inuktitut tracer
TRACER(InukTracer)
using namespace PTH;
LEVEL(PTH_LEVEL,1)
EVENT(PTH_CREATE_0, ThreadId pthreadId)
EVENT(PTH_CREATE_1, ThreadId pthreadId)
EVENT(PTH_SLEEP_0, ThreadId pthreadId, long milli, long nano)
EVENT(PTH_SLEEP_1, ThreadId pthreadId, long milli, long nano)
EVENT(PTH_YIELD_0, ThreadId pthreadId)
EVENT(PTH_YIELD_1, ThreadId pthreadId)
```

Dans cet extrait (voir tableau VI.1), on peut distinguer :
- le nom des fichiers d'en-tête pour disposer des types de variables utilisés : (`pth_pthreadsync.h`) ;
- le nom du traceur à utiliser (`InukTracer`) ;
- des instructions propres au langage utilisé (ici une déclaration d'utilisation d'un espace de noms C++ [Str01] « `using namespace PTH ;` » pour avoir accès aux types des variables utilisées par la bibliothèque INUKTITUT de gestion de *threads*) ;
- un ensemble d'instructions de définition d'événements sous la forme suivante :
  - la définition du niveau nommé `PTH_LEVEL` et numéroté `1` ;
  - les définitions des événements du niveau `PTH_LEVEL`.

À chaque événement est automatiquement attribué un numéro, ce qui permet de définir un couple (N° de Niveau, N° d'événement) permettant d'identifier chaque événement. Cette description est fournie par l'utilisateur pour définir les événements impliqués dans le processus de traçage. Nous allons ensuite nous baser sur cette définition pour construire automatiquement (en utilisant le processeur de macro M4 [M4]) des fonctions de traçage et de relecture.

## VI.3 Les formats de trace

Un des points critiques du traçage est l'utilisation d'un format adapté à l'enregistrement des événements constituant la trace. Le choix d'un format résulte d'un

compromis entre différentes contraintes :

- minimisation de la taille de la trace ;
- simplicité d'exploitation de la trace ;
- adéquation aux techniques d'enregistrement et d'interprétation.

### VI.3.1 Taille des traces

Le traçage de l'exécution d'un programme parallèle peut engendrer une très grande quantité d'observations, ce qui conduit à des fichiers de trace particulièrement importants. Pour donner un ordre de grandeur, sur un processeur cadencé à 1 Ghz, en enregistrant un événement d'une dizaine d'octets toutes les 10 000 instructions, on arrive à la production d'une trace d'un ordre de grandeur du méga-octet par seconde. Pour une application durant quelques minutes sur une dizaine de machines, on obtient très rapidement des tailles de fichiers de trace se mesurant en giga-octets. Il est donc important de limiter la taille des traces. Il faut cependant prendre garde à ne pas enregistrer trop d'événements car on arrive rapidement dans une situation où l'intrusion change radicalement le comportement du programme observé.

Ces tailles de fichiers ont une conséquence directe sur l'occupation des systèmes de stockage du nœud de calcul pour l'enregistrement des traces. Afin de ne pas saturer la mémoire vive du nœud de calcul, les données sont enregistrées sur un disque local ou transférées vers un système de stockage global. Dans tous les cas de figure, la taille des données va influer directement sur l'exécution du programme observé en raison de l'intrusion du système d'enregistrement bloquant le système d'opération pour effectuer les entrées/sorties.

De nombreux travaux portent sur la réduction de la taille des fichiers générés au cours du traçage [FCL02, JJZ01]. Cet aspect est extrêmement important en raison des tailles gigantesques que peuvent atteindre ces fichiers. La limitation de taille se base sur des techniques de compression, c'est-à-dire sur un codage plus dense des informations. La compression peut se faire en utilisant des techniques très classiques et générales. Le simple fait d'enregistrer sous un format binaire permet de réduire d'un facteur 2 une trace par rapport au même format exprimé textuellement et cela réduit également l'intrusion due au codage sous forme ASCII des valeurs des paramètres des événements. Des techniques de compression classiques (`gzip`, `compress`, etc.) permettent de gagner encore en densité, mais leur emploi réclame une utilisation supplémentaire de temps de calcul et donc augmente l'intrusion. Johnson et al.[JJZ01] proposent des techniques se basant sur l'existence d'un grand nombre de références de même nature souvent corrélées. On peut ainsi obtenir une réduction d'un ordre de grandeur des tailles de fichiers de traces. Il est également possible de se baser sur des techniques spécialisées utilisant la reconnaissance de motifs dans des traces souvent répétitives [FCL02]. Celles-ci sont utilisées dans le

cadre d'enregistrement d'information pour des analyses statistiques de performance mais elles sont cependant peu adaptées à l'observation comportementale en raison des pertes d'informations.

### VI.3.2 Exploitation des traces

Pour l'exploitation des informations enregistrées lors d'une exécution, il faut que celles-ci soient compréhensibles pour le logiciel destiné à les exploiter. Communément, cette caractéristique est garantie par la définition d'un format strict que doivent respecter, en écriture, le programme de production de traces et en lecture, le logiciel d'exploitation. Il résulte de cette conception une très forte dépendance du format de trace envers le modèle de programmation et l'outil de traitement des traces. Afin de réduire cette dépendance, diverses solutions peuvent être mises en œuvre :

- utiliser des formats fixes différents et convertir les traces ;
- définir un format idéal générique.

Nous allons voir que ces techniques ne sont pas satisfaisantes pour garantir une forte adaptabilité du système d'observation.

#### VI.3.2.1 Utilisation d'un format de trace standard

Cette approche, la plus évidente, consiste à définir un format de trace suffisamment évolué pour convenir à la majorité des outils d'exploitation de traces. Les outils de collecte de données suivraient également ce format et seraient tous compatibles. Malheureusement, cette solution se heurte à de nombreuses contraintes techniques et pratiques. D'un point de vue technique, la multiplicité des utilisations possibles des données d'observation de programmes rend la définition d'un format unique très complexe [Moh92]. Il existe un grand nombre de formats de trace différents car il est bien plus difficile de définir un format unique et adapté à toutes les situations que de redéfinir pour chaque nouvelle utilisation un format adapté à l'utilisation visée. Cette situation est analogue à celle observée dans le monde de la production de documents où chaque outil utilise un format différent pour ses propres besoins. L'utilisation d'un format généraliste tel que XML [XML] n'est apparue que récemment et elle n'est pas encore universelle. Contrairement à XML, les fichiers de traces ont une extrême exigence de minimisation de taille et ne peuvent êtres formatés à l'aide de marqueurs.

B. Mohr [Moh92] considère que la standardisation d'un format de trace basé sur la représentation d'événements n'est pas une approche pertinente, à moins d'inclure dans la trace des informations de définition du format, et il propose de standardiser plutôt les *méthodes d'accès* aux informations contenues dans la trace. Ce mode de

fonctionnement peut être comparé à l'utilisation de XML ou des formats de fichiers pour la vidéo (quicktime ou avi [Avi] par exemple). Tous deux reposent sur des standards pour l'accès aux données brutes, mais nécessitent des informations supplémentaires (DTD pour XML, *codec* (bibliothèque de fonctions) de décompression pour les vidéos) pour pouvoir être exploités.

Dans le domaine des formats de trace de programmes, des efforts pour l'adaptabilité ont été faits dans le sens de l'unification par la création de formats auto-définis. Ceux-là, tels que le format SDDF de PABLO [Ayd03] ou le format PAJÉ [dOSdK] s'appuient sur une description de la sémantique de la trace.

Il ne s'agit donc pas simplement d'un format, unique pour tout type d'application, mais également d'un langage de définition permettant de *définir* le format utilisé. Il est alors nécessaire, pour analyser la trace, de disposer non seulement des événements survenus au cours de l'exécution, mais également de la description de ceux-ci. Cette manière de procéder améliore la flexibilité de l'utilisation d'un format : on peut définir et utiliser des événements adaptés à chaque modèle de programmation ou type de langage. Un même format de fichier peut ainsi être adapté à des cas d'utilisation très différents. L'utilisation d'un format auto-défini entraîne une complexification des outils de visualisation qui doivent adapter leur représentation graphique aux formats définis.

### VI.3.2.2 Conversions entre formats de trace

Si on utilise des formats différents ayant chacun leurs spécificités, il faut, pour gérer le problème de la diversité, concevoir des programmes de conversion. Si les différences entre deux formats sont purement syntaxiques, il est relativement aisé de créer un outil de conversion pour traduire des traces d'un format à l'autre. Il est bien plus complexe de combler les écarts liées à des différences entre modèles de programmation ou entre niveaux d'abstraction des informations tracées. Un format de trace conçu pour des enregistrements de traces de programmes MPI ne pourra pas aisément gérer des traces de programme comportant des processus légers. Un autre format de trace prévu pour gérer des traces d'applications ne sera pas forcément apte à supporter des informations de type système. De plus, créer et maintenir $n \times n$ convertisseurs nécessaires pour $n$ formats de trace est une solution nécessitant un travail considérable et peu satisfaisant. Une solution intermédiaire (adoptée par certains systèmes de traduction automatique des langues naturelles) est de définir un « format pivot » d'un plus haut niveau d'abstraction et de concevoir seulement $2 \times n$ convertisseurs pour passer de n'importe quel format à n'importe quel autre. Le problème qui apparaît alors est celui de trouver un format pivot d'une expressivité suffisante pour pouvoir être utilisé pour toutes les conversions possibles. Cela revient à disposer d'un format suffisamment riche pour exprimer toutes les notions utilisées par tous les modèles de programmation, et on se retrouve dans le contexte d'un format de trace universel.

Au cours d'expérimentations préliminaires, nous avons ainsi mis au point quelques scripts de conversion de traces pour visualiser des traces produites par VampirTrace et TAU à l'aide de PAJÉ. S'il est aisé de réaliser un simple traducteur de traces pour passer d'un format à un autre de niveau sémantique équivalent, la conception peut se révéler plus problématique pour la conversion entre des formats ayant des sémantiques distinctes. Quant à l'automatisation de la conversion des traces d'un format quelconque vers un autre, elle est extrêmement difficile à réaliser. Il est en effet nécessaire pour cela de disposer de la sémantique des deux formats pour convertir chaque enregistrement. Cette information, implicitement connue par le programmeur de scripts de conversion, est difficile à obtenir et à formaliser pour la création automatique de convertisseurs.

### VI.3.2.3 Généricité des formats de trace

Il est intéressant de noter que la description d'une action à l'aide d'un format générique tel que celui de PAJÉ entraîne une augmentation du nombre d'événements. Dans un format spécialisé, on ne manipule que des événements explicitement définis. Il est donc facile de réduire le nombre d'enregistrements par la création d'événements complexes englobant plusieurs événements élémentaires, puis de spécialiser et d'optimiser le codage de ceux-ci. Dans le cas d'un format non spécialisé, il n'existe que des événements élémentaires se combinant pour former des actions plus complexes. La conversion d'un format spécialisé en un format générique s'accompagne donc immanquablement d'une augmentation du nombre d'événements à traiter.

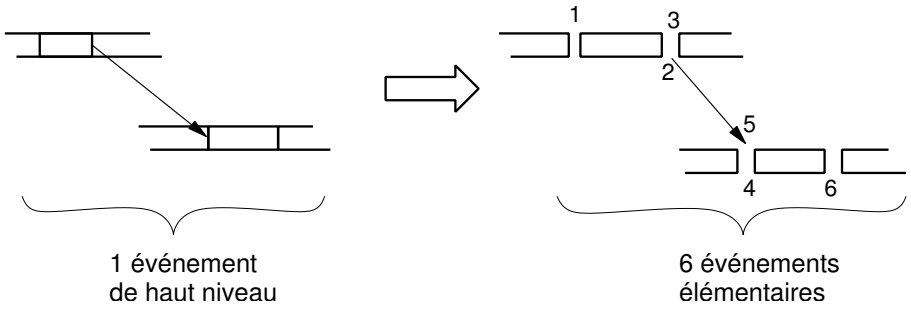



**Figure** VI.1 – Généricité des événements

La description d'un envoi de message dans un format d'enregistrement générique implique l'utilisation d'un plus grand nombre d'événements que dans un format plus spécialisé.

Pour illustrer ce phénomène, considérons une action de type « envoi de message » (voir figure VI.1) comprenant les opérations élémentaires suivantes :

1 changement d'état du processus émetteur ;
2 envoi du message ;
3 sortie du processus émetteur de l'état d'émission ;

4 changement d'état du processus destinataire ;
5 réception du message ;
6 sortie du destinataire de l'état de réception.

Dans un format élaboré, cet événement peut être représenté à l'aide d'un seul enregistrement dont la signification est connue par le programme de visualisation. Dans le cas d'un format générique, il faut décomposer l'événement en ces 6 actions élémentaires.

Cette constatation justifie de ne pas utiliser un format générique pour l'enregistrement de traces car une multiplication de la taille moyenne d'un fichier de trace par 2 ou 3 serait très préjudiciable.

## VI.4 Distinction des formats

Les remarques de la section VI.3 sur les formats de trace tendent à opposer les caractéristiques positives lors de l'enregistrement et celles de l'exploitation. La solution que nous utilisons est la distinction claire du format des informations enregistrées lors du traçage et du format des traces lues par les programmes de visualisation ou d'analyse. On enregistre les traces sous un *format brut* lors de l'exécution mais un *format élaboré* est utilisé pour l'exploitation des traces. On peut ainsi disposer des avantages des deux types de formats :

**Tableau** VI.2 – Formats des traces

| | **Format brut** | **Format élaboré** |
|---|---|---|
| codage | binaire | texte |
| caractéristique | dense | lisible |
| sémantique | pas de signification intrinsèque | auto-définie ou fixe |

Cette manière de procéder implique évidement l'existence d'un système de conversion entre le format brut et le format élaboré (voir figure VI.2).

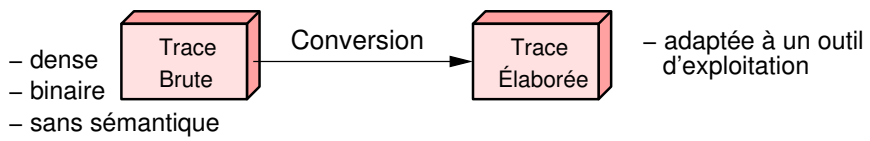
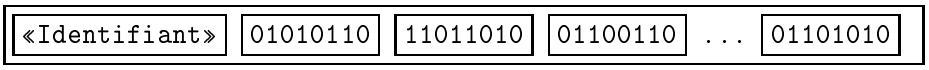



**Figure** VI.2 – Conversion d'une trace brute en une trace élaborée

### VI.4.1 Format brut

La trace brute se présente sous un format binaire pour être la plus dense possible et *sans sémantique particulière*, c'est-à-dire que les informations liées à la nature ou à la signification des champs d'un enregistrement sont absentes de la trace. Les enregistrements sont simplement préfixés par un *identifiant* constitué d'un doublet (`Niveau, Événement`). Le reste de l'enregistrement est constitué d'une série d'octets sans signification *a priori* :




Cette approche permet de disposer d'un format d'enregistrement à la fois compact et flexible.

Il est compact par l'absence d'informations non fondamentales. On a supprimé toutes les informations permettant d'associer une signification aux enregistrements contenus dans la trace. Sans information supplémentaire, il n'est même pas possible de distinguer les différents champs ou enregistrements. Un format où une description des champs ou des enregistrements serait présente serait beaucoup moins dense.

Il est flexible car il est ainsi possible d'enregistrer n'importe quel type de données. Il suffit d'être capable d'écrire et de relire de manière cohérente les informations enregistrées et de pouvoir les interpréter. L'enregistrement d'un événement consiste à conserver une copie de cet événement. Les informations nécessaires pour obtenir une « photographie » d'un événement sont imposées par l'usage que l'on veut faire de la trace. La manière d'enregistrer ces informations est dictée par des contraintes d'efficacité et par le respect du format de trace choisi. Notre objectif est de pouvoir paramétrer les événements entrant en jeu dans l'observation. Nous avons choisi de réaliser ces mécanismes par la génération de fonctions destinées à l'écriture et à la lecture des informations.

#### VI.4.1.1 Enregistrement des événements

Pour l'observation comportementale, les paramètres des objets que l'on désire enregistrer sont : un identifiant de l'événement, sa date et son lieu d'occurrence ainsi que les valeurs des paramètres s'y rapportant. Nous appelons lieu d'occurrence d'un événement une identification de l'entité physique (nœud) ou abstraite (*thread*) à laquelle est rattaché un événement. Diverses contraintes apparaissent pour l'implantation de l'enregistrement. Ces actions doivent avoir un coût minimum pour limiter l'intrusion. Il n'est donc pas envisageable de concevoir une unique fonction d'enregistrement complexe analysant les cas de figure rencontrés pour ensuite enregistrer les valeurs des variables pertinentes. Il faut concevoir une fonction par objet

à enregistrer. Comme ces enregistrements sont basés sur un paradigme événementiel, nous concevons une fonction par événement. Le nombre d'événements étant limité, le nombre de fonctions reste raisonnable (de l'ordre de quelques dizaines ou de quelques centaines).

Chacune de ces fonctions, appelée pendant l'exécution du programme lors de la rencontre d'un point de trace, a un certain nombre d'actions à effectuer :

- déterminer un tampon de trace approprié ;
- copier les valeurs des paramètres des événements à tracer dans le bon tampon ;
- contrôler la taille des événements et celle restante dans les tampons pour appeler les fonctions de sauvegarde des tampons en cas de débordement possible.

Ces fonctions peuvent être rébarbatives à implanter et sont très répétitives. Il n'est donc pas envisageable de laisser leur réalisation à l'utilisateur. Nous automatisons donc la génération de ces fonctions : le code composant les fonctions d'enregistrement est généré à partir des définitions d'événements effectuées par l'utilisateur. L'utilisation de macro-commandes limite l'intrusion due aux surcoûts des appels de fonctions.

Ces fonctions sont accessibles de la même manière que celles d'une bibliothèque classique et les prototypes de celles-ci correspondent aux définitions d'événements fournies par l'utilisateur (voir section VI.2). Leur utilisation par un programmeur est donc des plus simples. Comme les fonctions d'enregistrement et de relecture des traces sont automatiquement générées, leur cohérence est assurée.

#### VI.4.1.2 Relecture des événements

L'utilisation d'un format de trace brut dans lequel les événements enregistrés n'ont pas de signification *a priori* permet d'enregistrer des informations de toute nature en se basant sur des types de base ou des types définis par l'utilisateur. En contrepartie, il faut savoir comment décoder ces informations. Contrairement à un format comme XML, il n'y a pas de balisage susceptible de guider la lecture d'un fichier de trace. La seule information dont on dispose est que le début de chaque enregistrement correspond à un identifiant d'événement. La relecture implique donc de connaître en détail le format utilisé lors de l'enregistrement d'un événement pour relire la trace complète.

À partir de l'identifiant, il est possible de déterminer quel est l'événement enregistré. Il faut ensuite disposer des informations relatives à la manière de décoder les paramètres qui lui sont associés. Ces informations sont la *longueur* de cet enregistrement et la manière de segmenter la suite de bits lue dans le fichier pour convertir chaque champ en valeur significative.

Nous apportons ces informations à travers des fonctions de lecture décodant les informations de la suite de bits en fonction de l'enregistrement rencontré. Comme les événements enregistrés sont configurables et non pas fixes, il faut pouvoir adapter

les fonctions de décodage. Pour cela, ces fonctions sont générées automatiquement à partir des définitions des événements, de la même manière que les fonctions d'enregistrement. Ce mécanisme de génération automatique permet en outre d'assurer une cohérence entre les fonctions d'enregistrement et de lecture, difficile à conserver si les fonctions de décodage sont implantées séparément des fonctions d'enregistrement.

Voici par exemple un enregistrement décodé sous une forme textuelle simple :

```
Level(8)::Event(0)::Node(0)::Thread(1024)                    \
   ::DATE(Sec(34)::Usec(177832))::USER_SEQ_EVT_1(val==4)}
```

On peut identifier aisément :

- le niveau logique et le numéro de l'événement servant d'identifiant ;
- les numéros du nœud et du thread desquels cet événement est issu ;
- la date ($s$ et $\mu s$) à laquelle cet événement a été enregistré ;
- le ou les paramètres définis par l'utilisateur (ayant ici la valeur 4).

Cependant, cette relecture se base uniquement sur les noms et les valeurs des paramètres préalablement définis. Elle s'adapte donc aux événements tracés, mais n'est pas configurable. Nous devons donc définir un système de conversion permettant de générer des traces, dans un format plus élaboré, décrit par un utilisateur et utilisable par des outils d'exploitation.

### VI.4.2 Format élaboré

Le format élaboré est moins dense et plus descriptif pour pouvoir s'adapter à diverses formes d'exploitation : analyse statistique ou outil de visualisation. Il peut être un format strict destiné à un outil d'exploitation comme Vampir ou bien être de type auto-défini (SDDF, Pajé) et contenir les informations destinées à son interprétation.

Cette distinction des formats permet de gagner en performance et en place avec le format brut, et en expressivité avec le format élaboré. L'inconvénient est d'être obligé de convertir les enregistrements pour passer d'un format à un autre. Cette conversion doit donner le moyen d'apporter une signification aux événements, ce qui permettra leur interprétation et leur utilisation.

### VI.4.3 Conversion

Notre approche supprimant toute sémantique de la trace brute, il faut être en mesure de donner un sens aux enregistrements afin de pouvoir exploiter les événements qu'ils représentent. Il faut pour cela fournir une sémantique à toutes les informations contenues dans la trace. Cette sémantique permet, par exemple, au programme de

visualisation de déterminer comment afficher les différents événements et ainsi de retranscrire la signification des événements enregistrés. Nous avons vu qu'à l'aide des informations apportées par les définitions des événements —la liste, les noms et les types des champs de chaque enregistrement— nous pouvons relire les traces et en fournir une version textuelle lisible. Cependant, cette représentation élémentaire n'est significative que dans la mesure où les noms des événements ont un sens, ce qui est typiquement le cas pour l'utilisateur ayant défini ces noms. Pour qu'une trace soit exploitable par un outil de visualisation, il faut en plus disposer des informations nécessaires à la mise en correspondance des événements avec une représentation graphique.

Pour adjoindre cette sémantique manquante aux traces brutes, la solution que nous proposons est de fournir un *décodage adaptable* des données produites par notre traceur. Cela signifie qu'il sera possible de définir facilement le format de sortie du programme de décodage pour l'adapter aux besoins de l'utilisateur. Cette adaptation du décodage est assurée par un paramétrage des fonctions de décodage *via* des *règles de réécriture*.

### VI.4.3.1 Règles de réécriture

Une règle de réécriture permet de convertir un enregistrement de la trace brute en un ou plusieurs enregistrements de la trace élaborée. Pour chaque type d'enregistrement ,une règle est définie (voir figure VI.3). Elle prend la forme d'un couple (`Niveau, Événement`) pour identifier l'événement enregistré, d'une chaîne de caractères comportant un ou plusieurs caractères d'échappements « @ » et d'une série de noms de paramètres.

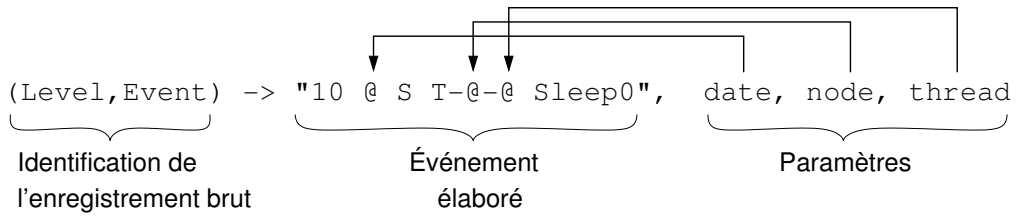



**Figure** VI.3 – Règle de réécriture d'un enregistrement.

Cette règle est utilisée pour générer une fonction chargée de réécrire l'enregistrement brut. La conversion se fait, à la manière d'une fonction du genre de `printf`, en remplaçant dans la chaîne de caractères de la règle chaque caractère d'échappement « @ » par les valeurs des paramètres de l'enregistrement brut. Voici un exemple possible d'un événement après réécriture par la règle précédente :

```
10 1.43252 S T-2-1026 Sleep0
```

Ce mécanisme de règles de réécriture permet de traduire facilement les événe-

ments de type 1 :N c'est-à-dire pour lesquels un enregistrement du format brut correspond à 1 ou plusieurs événements du format élaboré. Cela implique que tous les paramètres nécessaires à la construction de l'événement élaboré soient définis dans l'enregistrement brut. Ce schéma de réécriture 1 :N est le plus courant car, comme nous l'avons vu à la section VI.3.2.3, un événement de la trace brute est généralement plus concis qu'un événement de la trace élaborée.

## VI.5 Manipulation des traces

Après l'exécution du programme instrumenté et la génération de traces, un certain nombre de traitements peuvent être effectués sur celles-ci avant de les exploiter. L'exploitation des traces étant dans notre cas centralisé (de part l'utilisation de PAJÉ qui ne fonctionne pour le moment que de manière centralisée), il est nécessaire de regrouper les traces sur un serveur. Il est possible d'effectuer un certain nombre de traitements avant ou après la récupération des traces produites.

### VI.5.1 Collecte des traces

Pour exploiter les données, il faut les centraliser. Le transfert des données peut être effectué pendant l'exécution du programme observé en écrivant les tampons mémoire sur un système de fichiers partagé ou distribué. Beaucoup d'outils, par exemple, enregistrent les données sur un fichier partagé par NFS. Ce système présente l'avantage d'être simple et bien adapté aux machines SMP. Malheureusement, cette solution ne peut être appliquée pour une grappe avec un grand nombre de nœuds. Les contentions sur le serveur de fichiers vont rapidement bloquer les écritures sur disque et perturber très fortement le programme observé. Il est donc préférable d'écrire les tampons dans des fichiers sur des disques locaux des nœuds qui seront récupérés après la fin du programme. Comme nous nous plaçons dans le cadre d'observations *post mortem*, cette collecte après exécution n'est pas une restriction forte pour l'exploitation des traces.

### VI.5.2 Filtrage

Lors de l'observation d'une exécution, certaines informations non pertinentes peuvent être masquées pour alléger la représentation ou le traitement des données. Il faut pour cela offrir des mécanismes permettant d'effectuer aisément de telles manipulations sur les traces produites. Le filtrage peut être qualitatif (exemple : ne pas afficher les nœuds inactifs) ou temporel (exemple : garder uniquement les

informations de la date $d_1$ à la date $d_2$).

Le filtrage temporel est important dans la mesure où les exécutions de programmes parallèles sont souvent constituées de différentes phases (*initialisation, calcul, synchronisation, ..., calcul, synchronisation, génération des résultats*). Il est fréquent de vouloir isoler une phase de l'exécution afin de l'étudier pour en améliorer les performances. Un aspect complexe de la réalisation du filtrage temporel est qu'à l'intérieur d'une fenêtre de temps, il est possible qu'un événement soit en relation avec un autre événement lui-même situé en dehors de cette fenêtre. Il est donc indispensable de conserver tous les événements extérieurs à la fenêtre qui peuvent intervenir. Pour connaître ces événements, il faut avoir une connaissance des liens de causalité qui existent entre eux. Pour cela, l'utilisation d'un simulateur tel que celui de PAJÉ est requise. Un filtrage temporel effectué sans interprétation des traces est possible, mais il peut potentiellement compromettre certains événements.

Deux manières de filtrer peuvent être envisagées : le filtrage définitif (destructif) ou le filtrage temporaire (simple masquage des informations superflues). Le filtrage définitif permet de supprimer des informations non pertinentes pour diminuer le volume de données manipulées. En contrepartie, les informations perdues ne peuvent plus être exploitées et il est nécessaire de relancer l'exécution du programme pour les obtenir à nouveau. Au contraire, le filtrage temporaire ne fait que masquer les informations non pertinentes à un instant donné, ce qui permet de réutiliser ultérieurement ces données.

Un filtrage temporaire est mis en œuvre dans l'outil PAJÉ pour clarifier la vue d'une exécution trop complexe en masquant des objets non pertinents. Un filtrage effectué plus tôt dans la chaîne de traitement des traces va permettre de réduire le volume de données à gérer afin de réduire les temps de traitement. Il faut, par contre, ne plus avoir besoin des informations car il n'est pas possible de les récupérer ultérieurement.

Il est simple de filtrer les traces d'un format textuel, mais cela peut être assez inefficace. Il est également utile de pouvoir filtrer des traces, encore dans un format binaire brut, sur chaque nœud d'exécution.

Nous avons donc mis en œuvre des mécanismes permettant de filtrer facilement les traces brutes produites. Il s'agit principalement de mécanismes permettant de trier ou filtrer aisément selon les identifiants d'événements, les dates, les numéros de nœud ou de *thread*.

### VI.5.3 Agrégation de données

Il est parfois difficile ou inutile de visualiser certaines informations de même type venant d'un grand nombre de sources différentes. Pour pallier ce problème, il est possible de regrouper ces informations de manière synthétique. L'*agrégation*

d'informations consiste à remplacer des valeurs par une seule, représentative de l'ensemble de ces dernières. On peut, par exemple, prendre comme représentation la somme, la valeur maximale, minimale, la moyenne ou tout autre indicateur de tendance.

Une manière classique de procéder est de fournir une procédure d'agrégation. L'utilisateur passe en paramètre la fonction qui doit être appliquée aux valeurs initiales (min, max, moyenne, comptage, etc.). De la même manière que pour le filtrage, l'agrégation est destructive ou non selon la manière et le moment où elle est effectuée.

PAJÉ fournit des systèmes d'agrégation pour permettre une vision concise d'un grand nombre d'informations. Cette faculté (« scalabilité ») permet de procéder à des visualisations de programmes s'exécutant sur un grand nombre de nœuds de calcul.

### VI.5.4 Tri

Les enregistrements des événements sont estampillés de manière à pouvoir les ordonner temporellement. En revanche, rien ne garantit que leur écriture sur disque soit ordonnée, plusieurs tampons de traces indépendants pouvant cohabiter. De plus, le résultat du réassemblage d'enregistrements provenant de plusieurs nœuds d'exécution ne sera pas non plus ordonné. La plupart des outils d'exploitation exigent que les événements de la trace lue soient triés chronologiquement.

La solution la plus simple est de concaténer les traces de chaque nœud, puis de trier le résultat. Une technique plus efficace est de trier localement chaque fichier de trace puis d'effectuer une fusion ordonnée des fichiers triés. Cette manière de procéder est plus rapide non seulement en raison du gain en complexité de l'algorithme, mais surtout par la parallélisation simple et efficace des tris.

D'une manière générale, toute opération pouvant s'effectuer sans réassemblage préalable des traces, c'est-à-dire avec la seule connaissance des événements locaux, peut avantageusement être effectuée localement si elle conserve la quantité d'informations présentes dans la trace. Si elle engendre des pertes d'informations, comme une agrégation, alors celles-ci ne pourront être récupérées.

Nous avons mis en place un mécanisme de tri des enregistrements pouvant être effectué soir localement sur chaque nœud pour les événements locaux, soit de manière globale sur la trace complète.

## VI.6 Conclusion

Dans ce chapitre, nous avons montré l'intérêt tiré de la séparation des formats de trace. Cela permet de bénéficier des avantages de compacité d'un format brut binaire et de l'expressivité d'un format élaboré à destination d'un outil de visualisation ou d'analyse statistique. Afin de facilement gérer l'adaptabilité de ces deux formats, nous avons défini des mécanismes de configuration du processus de traçage.

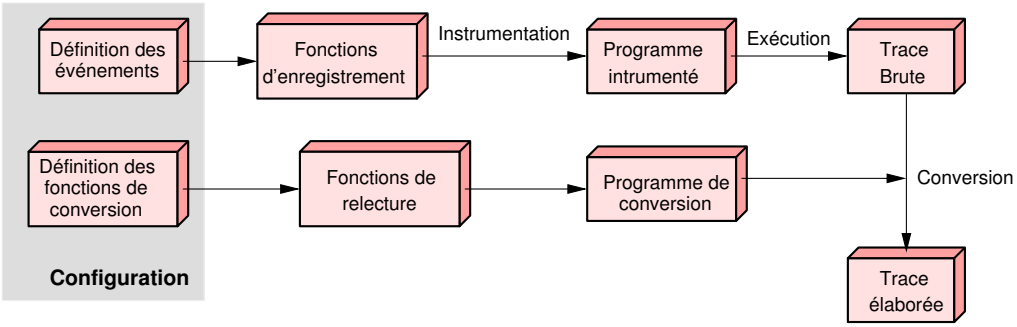



**Figure** VI.4 – Configuration du processus de traçage

La flexibilité du processus de traçage est assurée par les possibilités de configuration à la fois de l'enregistrement, par la définition des événements, et de la conversion des traces en un format élaboré, par l'utilisation de règles de réécriture.

Les diverses phases de configuration au cours du processus de traçage que nous avons mis en œuvre dans l'outil TUMIT (voir figure VI.4) permettent la flexibilité du processus de traçage. Celle-ci se retrouve à deux niveaux :

- dans la définition des événements, pour adapter les événements enregistrés aux besoins de l'utilisateur. Cela permet notamment d'adapter le traçage à différents modèles de programmation.
- dans la configuration des fonctions de relecture par les règles de réécriture permettant de convertir la trace brute en une trace élaborée, exploitable par un outil de visualisation.

Nous allons finalement présenter au chapitre suivant un exemple complet de traçage et de visualisation dans le cadre de l'observation d'un programme ATHAPASCAN.

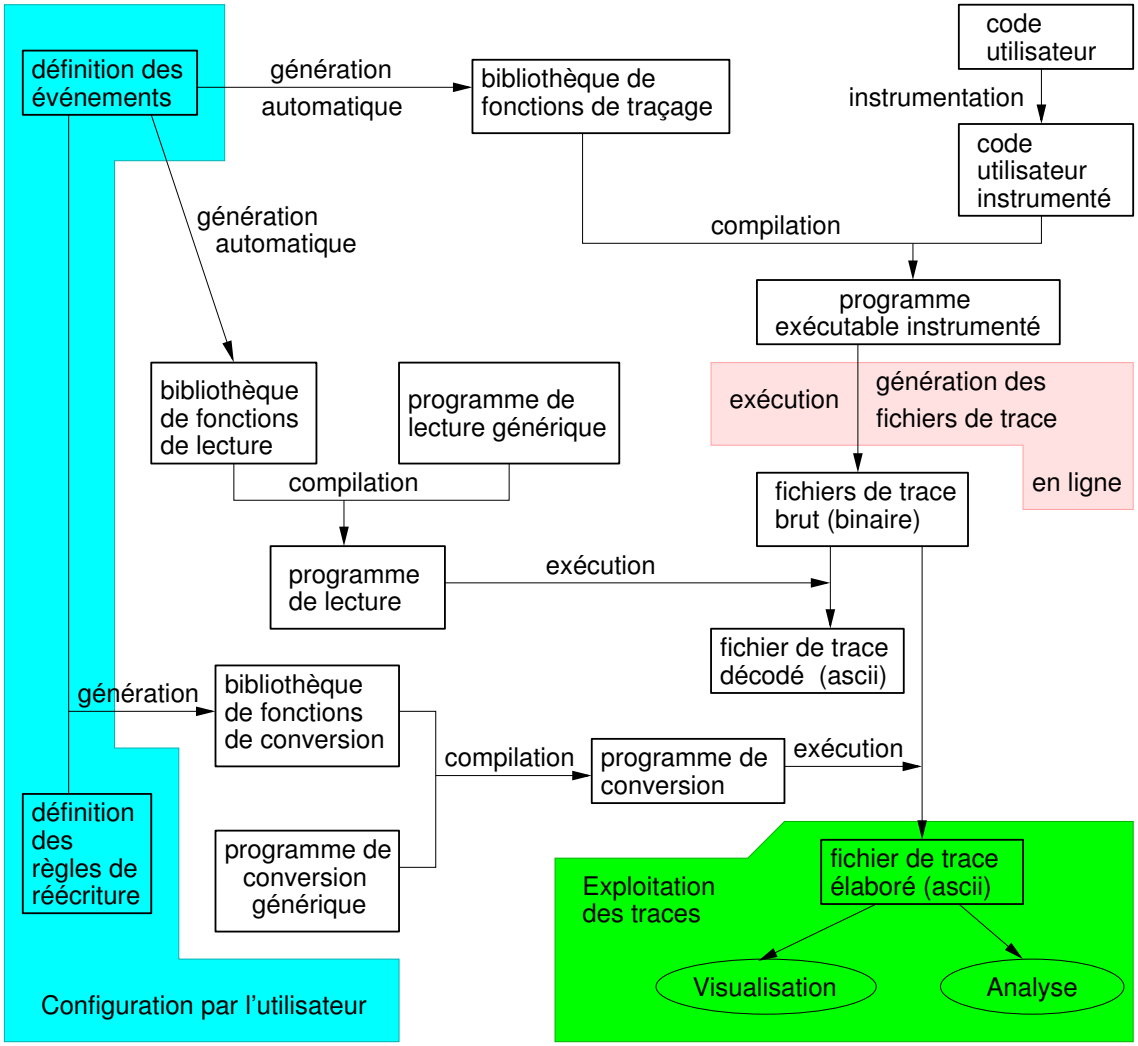



**Figure** VI.5 – Structure détaillée du processus de traçage

Les zone encadrées correspondent : aux zones de configuration du traçage, à la génération on-line des traces, et à l'exploitation des traces. Le processus complet est moyennement complexe, mais les tâches les plus rébarbatives sont automatisées. Cela permet de simplifier en partie l'observation, tout en laissant une grande liberté de configuration à l'utilisateur.

Chapitre

# Validation expérimentale

VII

*« Longue est la route par le précepte, courte et facile par l'exemple. »*
Sénèque

L'objectif de ce chapitre est de présenter un exemple complet et commenté du processus d'observation —illustré par la figure VI.5 page 90— et de souligner les principaux enseignements que nous en avons tiré.

Nous montrons ainsi les étapes successives d'une observation complète utilisant notre système de traçage TUMIT menant à la représentation graphique d'une exécution à l'aide de PAJÉ.

Cet exemple récapitule notamment comment et à quels moments interviennent les mécanismes de configuration présentés dans les chapitres précédents et mis en œuvre dans TUMIT. Nous nous intéressons à l'instrumentation de la bibliothèque ATHAPASCAN. Nous présentons l'enchaînement des opérations permettant d'observer le comportement des programmes utilisant cette bibliothèque :

- configuration du traçage (définition des événements) ;
- instrumentation du code de la bibliothèque ATHAPASCAN ;
- définition d'un modèle graphique ;
- configuration de la conversion d'une trace brute en une trace élaborée (définition des règles de réécriture).

## VII.1 Configuration et instrumentation

La configuration du traçage a pour but de refléter les abstractions du modèle de programmation étudié —ATHAPASCAN dans notre cas—, nous allons donc introduire quelques un des mécanismes utilisés dans ATHAPASCAN. Cette configuration est effectuée au travers de la définition des événements qui pourront être enregistrés. Nous allons présenter les définitions des événements que nous voulons enregistrer,

puis un exemple de l'instrumentation effectuée par les utilisateurs ayant la maîtrise du code, c'est-à-dire les développeurs d'ATHAPASCAN.

### VII.1.1 Observation d'ATHAPASCAN

Les paradigmes que nous voulons traiter sont ceux du modèle de programmation ATHAPASCAN. Pour donner un exemple d'observation multi-niveaux, nous voulons également pouvoir observer le comportement des *threads* de la couche de portabilité INUKTITUT (*Pthreads*) sur lesquels sont bâtis les mécanismes mis en œuvre par ATHAPASCAN (voir figure II.2).

ATHAPASCAN [GRCD98] [Rev04] est une bibliothèque de programmation parallèle de haut niveau. L'interface de programmation (API) qu'elle fournit permet de décrire le parallélisme du programme de manière explicite. Celui-ci est exprimé en utilisant des appels de procédures asynchrones distants qui créent des objets nommés *tâches*. Les tâches communiquent entre elles à l'aide d'une mémoire distribuée virtuelle et les synchronisations sont déduites des types d'accès aux objets partagés faits par les tâches (en écriture ou en lecture). La simplicité d'ATHAPASCAN est essentiellement due à sa sémantique séquentielle : chaque opération de lecture sur un objet partagé renvoie la dernière valeur écrite comme le définit l'ordre séquentiel d'exécution. Afin d'obtenir des exécutions efficaces des programmes, l'ordonnancement doit être adapté à l'application et à l'architecture cibles. Le programmeur peut utiliser des algorithmes généraux déjà implantés ou concevoir sa propre stratégie d'ordonnancement. Ces algorithmes peuvent tirer avantage de certains attributs spécifiques associés aux tâches et aux données partagées.

Les programmes ATHAPASCAN sont ainsi décrits indépendamment de l'architecture cible et de la stratégie d'ordonnancement choisie. L'utilisateur peut donc laisser la gestion des communications et des synchronisations entre tâches à ATHAPASCAN et se concentrer sur l'algorithmique. De plus, l'interface de programmation repose simplement sur deux mots clé : `Fork` et `Shared`. L'instruction `Fork` permet de créer un nouvelle tâche qui sera prise en charge par le système pour être exécutée, localement ou sur un processeur distant. La création d'une tâche est asynchrone et le noyau exécutif de la bibliothèque peut décider de l'exécuter sur n'importe quel processeur. Le mot clé `Shared` permet de créer une nouvelle donnée partagée.

Au cours de l'exécution du programme, les tâches créées vont être ordonnancées par ATHAPASCAN selon l'algorithmique choisi. ATHAPASCAN va déterminer une date et un lieu d'exécution pour chacune d'entre elles. Les *exécuteurs* ATHAPASCAN sont des *threads* chargés d'exécuter les tâches de travail. Afin de minimiser les temps d'inactivité, un exécuteur peut « *voler* » des tâches de travail à un autre exécuteur s'il se retrouve inactif. Il est important de bien définir les conditions de vol de travail car leur coût peut entraîner des problèmes de performance en cas d'utilisation abusive.

Pour favoriser la localisation de problèmes de performance, nous nous sommes intéressés aux actions relatives à l'exécution de programmes ATHAPASCAN, à l'ordre dans lequel elles se déroulent, et à l'évolution des états des exécuteurs. Nous nous attardons par exemple sur les actions impliquées dans le vol de tâches.

### VII.1.2 Définition des événements

La description des événements est effectuée *via* leur définition (le nom de chaque événement, et pour chacun des paramètres, son nom et son type). Ces définitions permettent de générer les fonctions d'enregistrement utilisées au cours l'instrumentation du programme. Elles sont également utilisées pour la génération des fonctions de relecture des traces brute sous forme textuelle élémentaire.

La définition complète des événements impliqués dans l'observation de la couche ATHAPASCAN compte pour l'instant environ 60 événements répartis en 5 niveaux (*gestion des tâches ; vol de travail ; vol de travail à distance ; ordonnancement et communications*).

L'extrait de définitions présenté dans le tableau VII.1 concerne les événements liés à l'activité des exécuteurs et au mécanisme de vol de tâches de travail entre ceux-ci.

**Tableau** VII.1 – Définition des événements ATHAPASCAN (extrait)

Cet extrait de définitions d'événements, utilisées pour l'instrumentation de la bibliothèque ATHAPASCAN, concerne les actions relatives au vol de travail et à l'état d'activité des exécuteurs. On peut distinguer 3 événements reliés aux requêtes de vol de travail et les événements correspondant aux actions de mise en sommeil, de réveil et d'envoi de messages aux exécuteurs (*worker*). `WORKER_STEAL_REQ`, par exemple, est l'événement correspondant à une requête de vol de travail. Ses paramètres explicites, (`thief_eid` et `victim_id`), permettent d'identifier l'exécuteur « voleur », qui a pris du travail, et la « victime », à qui une tâche a été enlevé.

```
// Work stealing events
LEVEL ( ATH_STEALING,  21)
EVENT ( ATH_WORKER_STEAL_0,   long thief_eid  )
EVENT ( ATH_WORKER_STEAL_REQ, long thief_eid, long victim_id )
EVENT ( ATH_WORKER_STEAL_1,   long thief_eid, int ok_bad)
EVENT ( ATH_WORKER_SLEEP_0,   long eid )
EVENT ( ATH_WORKER_SLEEP_1,   long eid )
EVENT ( ATH_WORKER_SIGNAL,    long eid )
EVENT ( ATH_WORKER_BROADCAST, long eid )
EVENT ( ATH_WORKER_WAKEUP_TIMEOUT, long eid )
EVENT ( ATH_WORKER_WAKEUP_SIGNAL,  long eid )
```

Nous avons regroupé ces événements au sein d'un même niveau (`ATH_STEALING`) afin de faciliter leur manipulation. Ces manipulations peuvent être un masquage des événements au moment de l'exécution lors de l'observation d'informations de plus bas niveau. Cela permet de simplifier la trace et de limiter l'intrusion en éliminant des informations non pertinentes. Par exemple, pour masquer tous les événements du niveau `ATH_STEALING`, il suffit d'affecter la variable `InukTracer::CONDITION_LEVEL_msk` à `1`.

On peut noter que, par convention, certains événements sont suffixés par `0` et `1` afin de distinguer le début et la fin d'une même action. Cela permet de connaître facilement le temps passé dans l'exécution de cette action, ce qui est un des objectifs du débogage pour les performances.

### VII.1.3 Instrumentation

Comme nous l'avons expliqué à la section III.3, l'instrumentation consiste à placer les mécanismes d'observation, c'est-à-dire les appels aux fonctions d'enregistrement, aux points de trace du code.

Le tableau VII.2 présente un court extrait du code d'ATHAPASCAN. On peut distinguer des appels aux procédures `pack` et `post` permettant de construire (d'emballer) un message et de l'envoyer. Entre ces deux instructions, se trouve le *point de trace* des événements `ATH_WORKER_RMT_STEAL_SEND` et `ATH_WORKER_RMT_STEAL_0`.

**Tableau** VII.2 – Extrait de code A1 instrumenté

Cette section de code correspond à la préparation d'un message (1er appel à la méthode `pack`) et à son envoi (fonction `post`). Les instructions destinées au traçage sont protégées par des directives de compilation (`#if defined(A1_USE_TRACE)` et `#endif`) afin de pouvoir les désactiver totalement. Ces instructions comprennent l'incrémentation d'une variable (`serial`) pour estampiller le message, son insertion dans le message (second appel à `pack`) et l'appel à deux fonction d'enregistrement : `ATH_WORKER_RMT_STEAL_0();` et `ATH_WORKER_RMT_STEAL_SEND(...)`.

```
...
call->pack ( &tid, sizeof(int) );

#if defined(A1_USE_TRACE)
  static long serial = 0;
  ++serial;
  call->pack ( &serial, sizeof(long) );
  ATH_WORKER_RMT_STEAL_0();
  ATH_WORKER_RMT_STEAL_SEND(Core::Communicator::getRank(), site, serial );
#endif

call.post (channel);
 ...
```

L'instrumentation est assez rapide et simple à réaliser pour un utilisateur connaissant bien le code qu'il veut observer. Le temps nécessaire à la définition des événements et de la mise en place des points de trace est de l'ordre de l'heure.

### VII.1.4 Exécution et génération des traces

Une fois les bibliothèques ATHAPASCAN et INUKTITUT instrumentées et compilées, l'exécution d'un programme lié avec ces bibliothèque provoque la génération des fichiers de trace (un fichier par nœud d'exécution). Cette trace est la trace brute dont les événements correspondent aux abstractions du modèle de programmation.

Nous avons pris entre autres comme exemple un programme effectuant un produit scalaire de vecteurs par blocs. Ce programme est très petit (environ 150 lignes de code) et il fait appel aux bibliothèques INUKTITUT et ATHAPASCAN. Des tests ont été effectués pour des durées d'exécution allant de quelques secondes à quelques minutes.

Nous avons tout d'abord constaté que le surcoût engendré par le traçage est évidement fortement dépendant du nombre d'occurrence des événements tracés. Le traçage de tous les événements —des couches INUKTITUT et ATHAPASCAN — engendre des surcoûts très forts (multiplication du temps d'exécution par un facteur de l'ordre de 10) et les tailles de traces produites sont très importantes (de l'ordre du méga-octet par seconde). L'intrusion sera donc prohibitive pour mesurer finement les durées d'exécution des actions observées. Le problème se pose par exemple lors du traçage des événements de la bibliothèque de *threads* d'INUKTITUT. Nous avons instrumenté beaucoup d'actions liées à la gestion des *threads* comme par exemple les prises et libérations de verrous. Or le nombre d'actions de ce type est très important au cours d'une exécution, même courte. Le traçage complet ne peut donc servir que pour observer des comportements peu sensibles à l'intrusion, ce qui est souvent assez restrictif dans le cas de mesures de performances de programmes parallèles, mais reste potentiellement intéressant pour comprendre la structure générale du déroulement d'un programme. Il est donc indispensable de masquer des événements pour retrouver un taux d'intrusion plus raisonnable. En supprimant totalement les points de trace des événements liés aux MUTEX dans INUKTITUT, on obtient des surcoûts bien plus faibles, de l'ordre de quelques pour cent (2 à 15% selon les exécutions). La quantité de trace générées est également bien moindre (de l'ordre d'une dizaine de Ko par secondes).

Nous avons effectué des essais de masquage d'événements à l'exécution et de suppression de points de trace à la compilation. Il apparaît, sans grande surprise, que le coût des tests des événements masqués est très inférieur à celui de l'enregistrement de ces événements. Il reste cependant important. Bien que le masquage (à l'exécution) des événements de trace dans la bibliothèque INUKTITUT réduise très fortement les surcoûts —on passe d'un surcoût d'un facteur de l'ordre de 10 à un surcoût d'environ 50 à 100%— le surcoût reste important en raison du très grand nombre d'événements tracés.

Une question importante — mais que nous n'avons pas abordée — est de savoir quel pourcentage de surcoût est acceptable dans le cadre du débogage pour les performances. Ce problème est très complexe car en principe toute observation est susceptible de modifier le comportement d'un programme. Comme il est impossible de supprimer totalement l'intrusion due à l'observation, il peut être intéressant de *corriger* cette intrusion [Mai96]. Dans le cas du traçage événementiel, il faut modifier les dates des enregistrements d'une trace de manière à se rapprocher du déroulement de l'exécution réelle. Pour effectuer cette modification, il faut cependant connaître le temps perdu lorsque le programme rencontre un point de trace et cela se révèle très complexe. On a donc généralement recours à des estimations statistiques.

Pour finir sur les résultats obtenus lors de nos diverses expérimentations sur le masquage et la suppression de points de trace, il est à noter que l'influence très forte des événements sur les surcoûts et les tailles de traces générées confirme l'intérêt d'un système de traçage flexible. Il est en effet important de pouvoir observer une exécution de programme de la manière la plus adaptée et la plus légère possible, non seulement pour des raisons de confort d'utilisation mais également de rapidité de traitement. Ces deux points nous semblent particulièrement pertinents dans le cadre de la mise au point de programmes, activité répétitive qui n'est pas toujours des plus attractives.

## VII.2 Configuration de la conversion

Pour représenter graphiquement une exécution, il faut tout d'abord définir un modèle graphique. Il doit être aussi proche que possible de l'idée que l'utilisateur se fait, intuitivement, de la structure du modèle de programmation dans le but d'offrir une représentation compréhensible facilement. La définition du modèle graphique se fait par la description d'une hiérarchie de types, construite à l'aide d'instructions Pajé.

### VII.2.1 Modèle graphique

Nous voulons visualiser graphiquement une représentation des paradigmes de programmation Athapascan et Inuktitut (exécuteurs, *threads*, requêtes, etc.). Il a été décidé de regrouper les informations selon un critère de spatialité (tous les événements liés à un exécuteur sont ensembles) plutôt que selon leur nature ( tous les événements de synchronisation ensembles).

Le tableau VII.3 présente la définition de la hiérarchie de types d'objet graphiques utilisée pour représenter les abstractions Athapascan et Inuktitut [1]. Cette hiérarchie est représentée sous la forme d'un arbre à la figure VII.1.

[1] Afin de limiter leur nombre, pour des raisons de lisibilité, certaines définitions d'états d'entités utilisés pour le traçage d'Athapascan et d'Inuktitut ont été supprimés dans l'exemple.

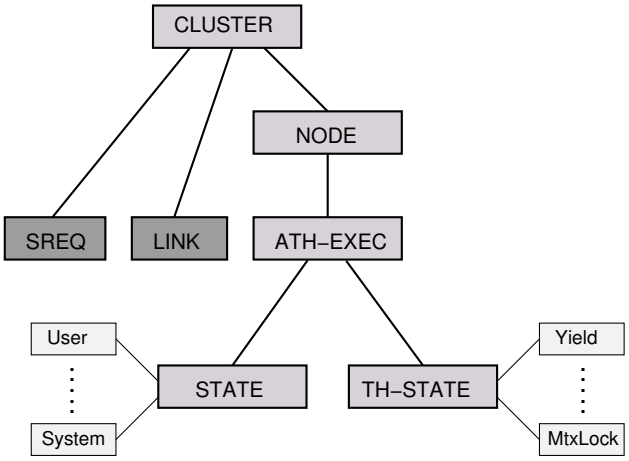



**Figure** VII.1 – Hiérarchie utilisée pour représenter une exécution ATHAPASCAN.

Un CLUSTER est composé de différents nœuds (NODE) sur lesquels s'exécutent plusieurs exécuteurs (ATH-EXEC). L'état (STATE) de chaque exécuteur peut prendre différentes valeurs : User,. . . ,System. L'état (TH-ÉTAT) du *thread* sous-jacent à l'exécuteur ATHAPASCAN peut prendre les valeurs : (Yield,. . . ,MtxLock). Les types LINK et SREQ servent à représenter les diverses communications et les requêtes de vol de travail. Les événements ATHAPASCAN et ceux liés aux *threads* INUKTITUT sont associés aux états correspondants.

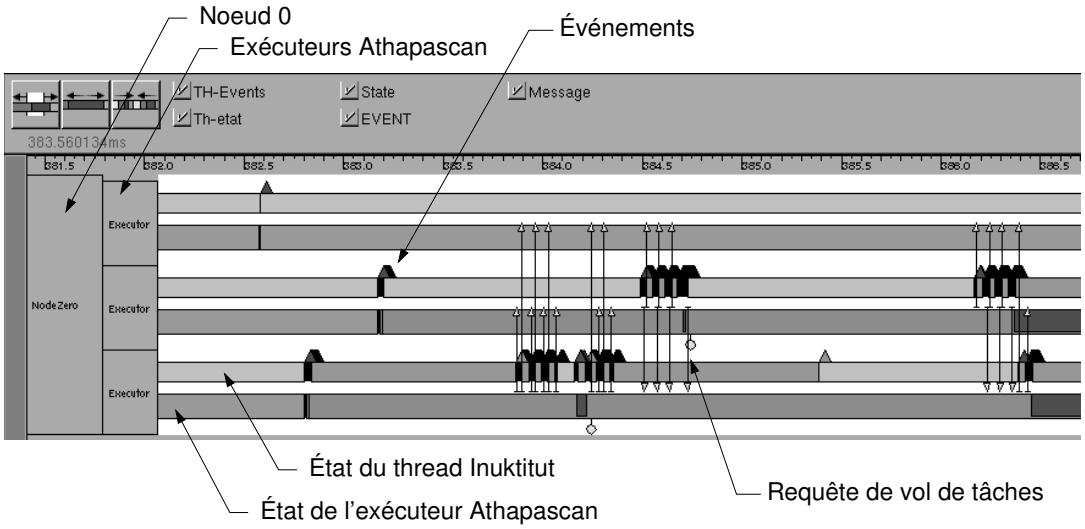



**Figure** VII.2 – Composants graphiques utilisés pour la représentation d'une exécution d'un programme ATHAPASCAN

Par défaut dans PAJÉ, les états des entités sont représentés par des rectangles de couleur, les événements ponctuels sont représentés par des triangles. Il est possible de changer l'objet graphique associé à un type, mais il est souvent préférable de conserver une forme intuitive facile à interpréter.

**Tableau** VII.3 – Définition de la hiérarchie de types

Cet ensemble d'instructions définit la hiérarchie de types utilisée pour représenter l'exécution d'un programme. Elle correspond à la hiérarchie de la figure VII.1. Les numéros de la première colonne identifient les instructions PAJÉ (*1 : définition d'un type de conteneur ; 2 : définition d'un type d'événement ; 3 : définition d'une entité ; 5 : définition d'un type de lien ; 6 : définition d'un état associé à une entité.*)

```
1  CLUSTER       0          "Cluster"
1  NODE          CLUSTER    "Node"
1  ATH-EXEC      NODE       "Executor"
3  STATE         ATH-EXEC   "State"
3  TH-STATE      ATH-EXEC   "Th-State"
2  EVENT         ATH-EXEC   "Event"
2  TH-EVTS       ATH-EXEC   "Th-Events"
5  LINK          CLUSTER    ATH-EXEC  ATH-EXEC  "Message"
5  MSG           CLUSTER    ATH-EXEC  ATH-EXEC  "Message"
6  Req           SREQ       "Steal request"
6  RmtReq        LINK       "Rmt Steal request"
6  Ack           LINK       "Ack for Steal request"
6  SigCom        LINK       "Activate-Execute"
6  Yield0        TH-STATE   "Yield_0"
6  Yield1        TH-STATE   "Yield_1"
6  LockMutex     TH-EVTS    "Mutex Lock"
6  UnLockMutex   TH-EVTS    "Mutex Un Lock"
6  WakeupTimeout EVENT      "Wakeup on timeout"
6  WakeupSignal  EVENT      "Wakeup on signal"
6  BroadcastCond EVENT      "Broadcast Condition"
6  User          STATE      "User's task exec"
6  Sys           STATE      "Athapascan system"
6  Steal         STATE      "Stealing"
...
6  ReceiveGraph  STATE      "Receiving Graph"
```

### VII.2.2 Règles de réécriture

Les règles de réécriture (voir tableau VII.4) vont permettre de convertir les événements bruts en une trace utilisable par PAJÉ. Ces règles sont très simples, la plupart des conversions sont de type « 1:1 ». Ici, seul l'événement correspondant à une requête de vol de travail doit être réécrit sous la forme d'un lien composé d'une source et d'une cible, ce qui se traduit par 2 instructions PAJÉ.

Le tableau VII.5 présente un extrait d'une trace obtenue après conversion. On distingue les deux instructions PAJÉ16 et 17 à la même date décrivant un lien

**Tableau** VII.4 – Extrait de règles de réécriture pour ATHAPASCAN

Ces règles de réécriture correspondent aux événements bruts définis précédemment. Les principaux paramètres utilisés sont la date (`_d`), le numéro de nœud (`_node`) et l'identificateur d'exécuteur (`eid`).

```
 ...
// Work stealing events
(21,0) -> "11 @ STATE E@-@    Steal           ", _d, _node, thief_eid
(21,1) -> "16 @ SREQ CLUSTER0 Req E@-@ Key-@  ", _d, _node, thief_eid, _d
       -> "17 @ SREQ CLUSTER0 Req E@-@ Key-@  ", _d, _node, victim_id, _d
(21,2) -> "12 @ STATE E@-@                    ", _d, _node, thief_eid
(21,3) -> "11 @ STATE E@-@    Sleep           ", _d, _node, eid
(21,4) -> "12 @ STATE E@-@                    ", _d, _node, eid
(21,5) -> "9  @ EVENT S@-@    SignalCond      ", _d, _node, eid
(21,6) -> "9  @ EVENT E@-@    BroadcastCond   ", _d, _node, eid
(21,7) -> "9  @ EVENT E@-@    WakeupTimeOut   ", _d, _node, eid
(21,8) -> "9  @ EVENT E@-@    WakeupSignal    ", _d, _node, eid
(21,9) -> "11 @ STATE E@-@    CloCopy         ", _d, _node, eid
(21,10)-> "12 @ STATE E@-@                    ", _d, _node, eid
 ...
```

**Tableau** VII.5 – Extrait de trace ATHAPASCAN au format PAJÉ

Cet exemple est extrait d'une trace d'exécution ATHAPASCAN convertie au format PAJÉ (format élaboré). On peut distinguer des instructions de changement d'état (numéros 10, 11 et 12), et un lien représentant une requête de vol de travail (numéros 16 et 17 pour l'émission et la réception). `E0-0` et `E0-2` sont les identificateurs de 2 exécuteurs.

```
  ...
 10  0.553964  STATE   E0-2      User
 11  0.553975  STATE   E0-0      Steal
 10  0.553984  STATE   E0-2      Sys
 16  0.553993  LINK    CLUSTER0  Req   E0-0  Key-0.553993
 17  0.553993  LINK    CLUSTER0  Req   E0-2  Key-0.553993
 10  0.554051  STATE   E0-2      User
 12  0.554078  STATE   E0-0
 10  0.554132  STATE   E0-2      Sys
```

*conteneur (src. et dest.)* — *clé d'appariement*

*identificateur d'instruction* — *date* — *type d'entité* — *conteneur* — *état*

représentant une requête de vol. Les autres instructions concernent les événements relatifs aux changements de valeur de l'entité représentant l'état des exécuteurs d'ATHAPASCAN. L'instruction `10` est un changement d'état simple, tandis que les instructions `11` et `12` correspondent à une notion d'empilement et de restauration d'état (*push/pop*).

## VII.3 Visualisation

La figure VII.3 est une vue de PAJÉ présentant une trace TAKTUK obtenue avec les définitions présentées dans cet exemple. Cette visualisation représente une exécution d'un programme ATHAPASCAN. Il s'agit d'un programme effectuant un produit scalaire par blocs. On a utilisé dans cet exemple 50 blocs de 1000 éléments et 5 exécuteurs ATHAPASCAN. On peut immédiatement distinguer les moments d'inactivité des exécuteurs (zones sombres) puis il est possible de « zoomer » pour identifier les origines de l'inactivité.

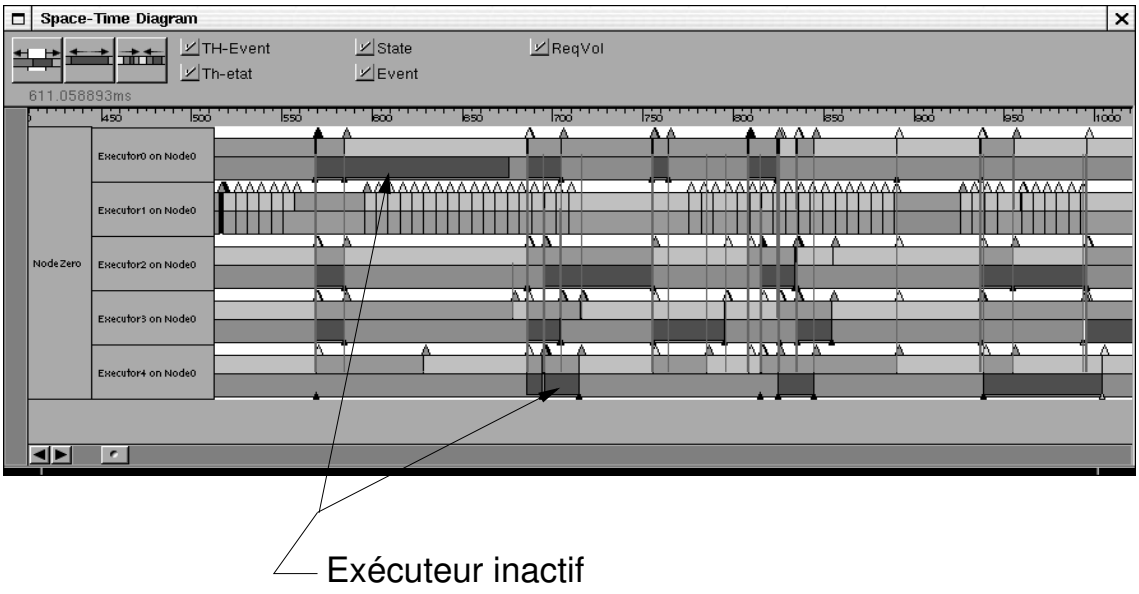



**Figure** VII.3 – Visualisation dans PAJÉ d'une trace ATHAPASCAN

## VII.4 Conclusion

Nous avons expliqué dans ce chapitre et grâce à un exemple commenté, comment sont utilisés, en pratique, les différents mécanismes de configuration : définition des événements et règles de réécriture. Nous avons montré que TUMIT, notre système de traçage flexible et de conversion de traces, a permis d'effectuer le traçage de programmes ATHAPASCAN en produisant des traces et en les convertissant en un format adapté à l'outil de visualisation PAJÉ. Grâce à la génération automatique des fonctions d'enregistrement, l'instrumentation a été effectuée relativement rapidement et

simplement tout en conservant une grande souplesse d'utilisation. Les résultats sur les surcoûts générés par le traçage nous incitent à considérer que l'utilisation d'un système de traçage flexible et facilement adaptable est pertinente et à privilégier. Il est en effet important de pouvoir cibler précisément les observations effectuées et de cerner plus facilement les problèmes de performance rencontrés. Les diverses expérimentations nous ont permis de confirmer la simplicité de la configuration des traces par des règles de réécriture. Il reste un problème important à traiter : la correction de l'intrusion du traçage afin d'obtenir des traces plus représentatives de la réalité de l'exécution.

Chapitre

# Conclusion et perspectives

VIII

Le travail réalisé au cours de cette thèse a tout d'abord permis de dresser un panorama des techniques d'observation et de définir les contraintes liées à la flexibilité d'un système de collecte de traces. Nous avons ensuite conçu et implanté TUMIT, un système de traçage d'applications parallèles flexible, adaptable au contexte d'utilisation.

L'origine de ce travail se trouve dans la volonté de développer des techniques de débogage pour les performances afin de faciliter la mise au point d'applications parallèles. Notre approches se base sur l'observation comportementale afin de fournir des informations issues des couches logicielles impliquées dans la conception d'une application parallèle. Ce choix a été fait pour sa capacité d'aide à la compréhension du déroulement d'un programme. Pour l'exploitation des données du processus d'observation, nous nous appuyons sur PAJÉ, un outil de visualisation de trace d'exécutions de programmes parallèle. PAJÉ offre une très grande flexibilité d'utilisation pour la visualisation de différents modèles de programmation. Nous travail a consisté à concevoir et implanter un système de collecte de traces pour fournir des informations d'observation exploitables par PAJÉ. Afin de rester cohérent avec PAJÉ, nous avons cherché à pourvoir cet outil de traçage d'un maximum de *flexibilité*.

Nous avons initié la conception du traceur en examinant les critères de flexibilité et les caractéristiques qui nous semblent pertinents pour un tel système. Après avoir étudié, sans succès, la possibilité d'utiliser certains systèmes de traçage existants, nous avons développé un système de traçage événementiel pour l'observation *post mortem*. Le critère de flexibilité qui nous a intéressé est celui de la possibilité d'adapter ce traceur à différents modèles de programmation et intergiciels, notamment ceux développés au laboratoire ID-IMAG.

Les problèmes soulevés par la flexibilité d'un traceur se retrouvent à deux niveaux. Il faut pouvoir configurer le traceur pour l'adapter au contexte d'utilisation de manière simple pour l'utilisateur sans induire une intrusion incompatible avec une observation de qualité. Il faut ensuite permettre une manipulation facile des données produites afin de pouvoir les exploiter avec un outil de visualisation ou d'analyse.

Les caractéristiques des formats d'enregistrement des données au niveau de la prise de trace sont incompatibles avec celles nécessaires à la flexibilité du format de l'outil d'exploitation. Pour cette raison, nous avons utilisé des *formats de trace distincts*, chacun adapté à la phase du processus d'observation dans laquelle il est utilisé : un format de trace brut pour optimiser la compacité des données et l'efficacité de l'enregistrement ; un format élaboré mieux adapté aux outils d'exploitation. Une conversion est évidement nécessaire pour passer de l'un à l'autre. Chacun de ces formats devant être adaptable, nous avons donc conçu et implanté un mécanisme permettant de configurer la conversion *via* des *règles de réécriture*.

Grâce à la flexibilité du processus d'observation, nous avons pu effectuer l'instrumentation des bibliothèques utilisées au laboratoire ID-IMAG afin de permettre l'observation des programmes les utilisant. Nous avons ainsi instrumenter les différentes bibliothèques composant INUKTITUT, TAKTUK et ATHAPASCAN et observer des exécutions de programmes les utilisant. Les résultats liés aux exécutions observées ont montrés une relation particulièrement forte entre les événements observés et l'intrusion engendrée. Cette constatation n'est guère surprenante, mais elle renforce l'intérêt pour un système de traçage flexible et facilement configurable.

L'utilisation pratique du système de règles de réécriture, sur de multiples exemples, a permis de valider leur utilisation en mettant en avant leur simplicité. Nous avons cependant relevé certaines contraintes liés à leur utilisation : nous n'avons pas défini de procédure permettant de manipuler les données à l'aide d'instructions de réécriture. Nous avons en effet considéré que cela pouvait être fait dans le code du programme observé appelant les fonctions de trace. Il est raisonnable de vouloir ajouter des instructions permettant des manipulations plus complexe, il faut cependant veiller à conserver la simplicité des règles de réécriture.

**Perspectives** Notre objectif est de concevoir des outils pour la mise au point d'applications parallèles. Nous fournissons avec TUMIT et PAJÉ un système d'observation flexible afin de observer facilement le comportement d'exécutions. Afin de poursuivre ce travail, plusieurs amélioration doivent être effectuées et certaines voies possibles de développement peuvent être explorées.

Bien que l'instrumentation d'un code bien connu soit assez rapide à réaliser, il faudra sûrement définir des moyens simples pour mettre en œuvre le système de traçage dans des codes indépendants d'INUKTITUT. Nous avons en effet perdu beaucoup de temps dans la configuration automatique du traçage, malgré l'utilisation d'outils tels que `autoconf` et `automake`.

Afin de faciliter la mise en œuvre de l'observation, il faudrait, par exemple, proposer aux utilisateurs des interfaces graphiques pour les aider tout au long du processus d'observation sans pour autant automatiser ce dernier. Il pourrait s'agir d'une interface pour aider à l'instrumentation du programme, à partir des définitions des événements ; ou encore de proposer un système graphique intuitif pour le masquage des points de trace (totalement, par niveau ou bien événement par événement).

L'observation de programmes sur une grappe de PC se heurte au problème de l'absence d'*horloge globale*. La datation des événements sur une grappe de PC se fait localement par l'horloge locale de chaque machine. Ces horloges locales ne sont pas parfaites et *dérivent* au cours du temps. Cette dérive est due en grande partie à des défauts physiques du *quartz* chargé de synchroniser l'horloge. La conséquence de cette dérive est qu'il est impossible d'obtenir directement des dates cohérentes entre deux nœuds distincts. Une synchronisation périodique (en utilisant le protocole NTP [Mil] par exemple) de toutes les horloges serait trop coûteuse et perturberait notablement l'exécution du programme observé. Une solution est de corriger les dates des événements de manière *post mortem*, ce qui supprime l'intrusion des synchronisations. Pour effectuer cette correction, il est impératif de connaître la dérive de chaque horloge. Une technique consiste, en se basant sur des propriétés physiques des quartz des horloges, à considérer la dérive comme linéaire sur un intervalle de temps limité [MT95, Mai96]. Grâce à une méthode d'échantillonnage, par une série de « ping-pong » entre une machine de référence et chaque nœud, des décalages de chaque horloge il est possible de déterminer les paramètres de la dérive : pente et décalage initial par rapport à l'horloge de référence. Ces paramètres servent ensuite à corriger les dates de chaque événement. L'application de cette méthode dans le cadre des grappes de PC pose un problème de « scalabilité » : effectuer des dizaines ou des centaines de fois l'échantillonnage peut être long. En plus d'être contraignant, le temps passé à échantillonner peut rendre caduques les considérations de linéarité. Pour limiter la durée d'échantillonage, nous essayons d'effectuer l'échantillonnage à partir d'une technique de diffusion (*multicast*), ce qui permettrait d'en réduire considérablement la durée. Des expérimentations [Gar02] [Kal03] ont commencé et doivent être achevées pour vérifier l'efficacité de l'utilisation de la diffusion

La hiérarchie de types graphiques utilisée par Pajé fournit une description d'un modèle graphique. Il serait intéressant de se baser directement sur celle-ci pour définir les événements impliqués dans le processus de traçage, au lieu de se baser comme nous le faisons sur une définition annexe des événements. On disposerait ainsi d'un moyen de configurer le processus d'observation bien plus cohérent dans l'optique d'arriver à une représentation graphique. Le revers ce cette approche est de perdre l'indépendance du système de collecte et de conversion vis-à-vis de l'outil de représentation. Il serait donc plus complexe de traduire des traces en un format autre que celui de Pajé.

L'un des inconvénients de l'observation *post mortem* est son absence de réactivité. Il n'est pas possible d'éliminer des informations que l'on juge non pertinentes à un moment donné sans les perdre et sans devoir recommencer les phases les ayant générées. Afin de proposer des techniques de filtrage ou d'agrégation dynamiques, il serait particulièrement intéressant de distribuer ces mécanismes. En réalisant, par exemple, un filtrage dynamique distribué, il serait possible de limiter fortement les informations échangées entre les nœuds d'exécution et la machine sur laquelle est effectuée l'observation.

Les questions de « scalabilité » ont été évoquées au travers des problèmes liés à

la représentation de grandes quantités de données. Des problèmes du même ordre se posent pour le traçage d'applications parallèles. Le principal est celui de la récupération des informations enregistrées sur chaque nœud d'exécution. La récupération de traces dont la taille peut être de l'ordre du giga-octet sur chaque nœud va poser des problèmes conséquents de rapatriement liés à la bande passante des réseaux d'interconnexion. Des perfectionnements dans cette direction seraient très utiles non pas pour la qualité de l'observation, mais pour l'amélioration des conditions d'utilisation d'outils d'observation, ce qui nous semble être un point capital pour favoriser l'utilisation de tels programmes.

Un autre point particulièrement intéressant est l'augmentation des capacités de traitement de Pajé. La distribution du simulateur —à laquelle a été consacré un stage de DEA [Sch03]— permettant la reconstruction d'une exécution par Pajé pourrait être partiellement faite localement sur chaque nœud d'exécution. Cela permettrait d'assurer une meilleure réactivité et vitesse d'exécution.

Finalement, l'étude des techniques d'observation (traçage, profilage, détection automatique de problèmes de performance) nous incite à penser que l'utilisation combinée de toutes ces approches de l'observation au sein d'un outil de développement unifié serait particulièrement intéressante pour la mise au point de programmes parallèles.

# Index

# Bibliographie


[AMD02] AMD. « *AMD Athlon Processor x86 Code Optimisation guide* », fév. 2002. `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/22007.pdf`

[AMS02] Philippe Augerat, Cyrille Martin et Benhur Stein. « *Scalable monitoring and configuration tools for grids and clusters* ». in *proc. of 10th Euromicro Workshop on Parallel, Distributed and Network-Based Processing*, pages 147–153. IEEE Computer Society Press, Silver Spring, 2002

[AP97] Eric Anderson et Dave Pattersonn. « *Extensible, Scalable Monitoring for Clusters of Computers* ». *11th Systems administration conference LISA'97*, 1997

[Apa01] « *APART WP3 : Module Index* », 2001. `www.fz-juelich.de/apart-1/wp3/modmain.html`

[Avi] « *MCF specification, technical part/AV-compressions* ». `http://mcf.sourceforge.net/oldpage/formats.htm`

[Ayd03] Ruth A. Aydt. « *SDDF Performance Data Metaformat* », 1992-2003. `ftp://vibes.cs.uiuc.edu/pub/Pablo.Release.5/SDDF/Documentation/SDDF.ps.gz`

[BBF00] Céline Boutros, Xavier Bonnaire et Bertil Folliot. « *A flexible monitoring plateform to build cluster management services.* ». in *Proc. of IEEE International Conference on Cluster Computing*, pages 258–265, 2000

[Ben00] Anne Benoit. « *Analyse des interactions entre le traceur* Athapascan-*0 et les applications tracées* », 2000. rapport de DEA informatique systèmes et communications - Université Joseph Fourier

[Ben03] Anne Benoit. « *Méthodes et algorithmes pour l'évaluation des performances des systèmes informatiques à grand espace d'états* ». PhD thesis, Institut National Polytechnique de Grenoble, France, juin 2003

[Ber97] Pierre-Eric BERNARD. « *Parallélisation et multiprogrammation pour une application irégulière de dynamique moléculaire opérationnelle* ». PhD thesis, Institut National Polytechnique de Grenoble, France,, octobre 1997

[BGM03] BRIAT, GAUTIER et MARTIN. « *Inuktitut* », 2003. `http://www-id.imag.fr/Logiciels/inuktitut/`

[BGPP97] BRIAT, GINZBURG, PASIN et PLATEAU. « *Athapascan runtime : efficiency for irregular problems* ». in *Proceedings of the Europar'97 Conference*, pages 590–599, 1997

[BH00] Bryan BUCK et Jeffrey K. HOLLINGSWORTH. « *An API for Runtime Code Patching* ». *The International Journal of High Performance Computing Applications*, 14(4) :317–329, Winter 2000

[BMS03] R. BELL, A.D. MALONY et S. SHENDE. « *ParaProf : A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis* ». in *Proc. EUROPAR 2003 conference.*, LNCS 2790, pages 17–26, Klagenfurt August 2003

[Bri] « *Brink and Abyss : Pentium 4 Performance Counter Tools For Linux* ». `http://www.eg.bucknell.edu/~bsprunt/emon/brink_abyss/brink_abyss.shtm#Abyss`

[Bul03] « *Bull* », 2003. `http://www.bull.com`

[Car99] Alexandre CARISSIMI. « *Le noyau exécutif Athapascan-0 et l'exploitation de la multiprogrammation légère sur les grappes de stations multiprocesseurs* ». Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France, novembre 1999. `ftp://ftp.imag.fr/pub/bibliotheque/theses/1999/Carissimi.Alexandre-da-Silva`

[Cav99] Gerson G.H. CAVALHEIRO. « *Athapascan 1 : Interface générique pour l'ordonnancement dans un environnement d'exécution parallèle* ». Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France, novembre 1999. `ftp://ftp.imag.fr/pub/bibliotheque/theses/1999/Cavalheiro.Gerson-Geraldo-Homrich/`

[CBR95] CHRISTALLER, BRIAT et RIVIÈRE. « *Athapascan-0 concepts structurants simples pour une programmation parallèle efficace* ». *Calculateurs parallèles*, 7(2), 1995. Edition Hermes Paris

[CDG+] G. CAVALHEIRO, M. DOREILLE, F. GALILEE, T. GAUTIER et J-L. ROCH. « *Scheduling parallel programs on non-uniform memory architectures* ». in *Workshop on "Parallel Computing for Irregular Applications WPCIA1"*, Orlando USA. HPCA Conference

[CdKdOS00a] J. Chassin de KERGOMMEAUX et B. de OLIVEIRA STEIN. « *Pajé, an Extensible and Interactive and Scalable Environment for Visualizing Parallel Program Executions* ». Rapport de recherche 3919, INRIA, Avril 2000. thème 1

[CdKdOS00b] J. Chassin de KERGOMMEAUX et B. de OLIVEIRA STEIN. « *Pajé : an Extensible Environment for Visualizing Multi-Threaded Programs Executions* ». in A. BODE, W. LUDWIG, T. Karl et R. WISMÜLLER, éditeurs, *Euro-Par 2000 Parallel Processing, Proc. 6th International Euro-Par Conference*, volume 1900 de *LNCS*, pages 133–140. Springer, 2000

[CdKdOS03] Jacques Chassin de KERGOMMEAUX et Benhur de OLIVERA STEIN. « *Flexible performance visualization of parallel and distributed applications* ». *Future Generation Computer Systems (FGCS)*, 19 :735–747, 2003

[CdKdOSB00] J. Chassin de KERGOMMEAUX, B. de OLIVEIRA STEIN et P.E. BERNARD. « *Pajé, an interactive visualization tool for tuning multi-threaded parallel applications* ». *Parallel Computing*, 26(10) :1253–1274, aug 2000

[CdKDOW97] Jacques Chassin de KERGOMMEAUX, Benhur DE OLIVEIRA et Phillipe WAILLE. « *Mise au point d'applications parallèles irrégulières* ». in *ICaRE'97 Conception et mise en oeuvre d'applications parallèles irrégulières de grandes tailles*. École CNRS, décembre 1997

[CdKGdOS03] J. Chassin de KERGOMMEAUX, C. GUILLOUD et B. de OLIVEIRA STEIN. « *Flexible performance debugging of parallel and distributed applications* ». in Harald KOSCH, László BÖSZÖRMÉNYI et Hermann HELLWAGNER, éditeurs, *Proc. of Euro-Par 2003*, volume 2790 de *LNCS*, pages 38–46. Springer Verlag, août 2003

[CLAa99] José C. CUNHA, João LOURENÇO et Tiago R. ANTÃO. « *An Experiment in Tool Integration : the DDBG Parallel and Distributed Debugger* ». *Euromicro Journal of Systems Architecture*, 45(11) :897–907, 1999

[Dee] « *DEEP/mpi* ». `http://www.psrv.com/deep_mpi_top.html`

[DLM+01] DONGARRA, LONDON, MOORE, MUCCI et TERPSTRA. « *Using PAPI for hardware performance monitoring on Linux systems* ». in *Linux Clusters : The HPC Revolution*, June 2001

[Dor99] Mathias DOREILLE. « *Athapascan 1 : vers un modèle de programmation parallèle adapté au calcul scientifique* ». Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France, décembre 1999. `ftp://ftp.imag.fr/pub/bibliotheque/theses/1999/Doreille.Mathias/`

[dOS99] B. de Oliveira Stein. « *Visualisation interactive et extensible de programmes parallèles à base de processus légers* ». PhD thesis, Université Joseph Fourier, Grenoble, 1999. In French. `http://www-mediatheque.imag.fr`

[dOSdK] Benhur de Olivera Stein et Jacques Chassin de Kergommeaux. « *Pajé trace file format* ». ID-imag ; UFSM `http://www-id.imag.fr/Logiciels/paje/usermanual.html`

[DZR98] Luiz DeRose, Ying Zhang et Daniel A. Reed. « *SvPablo : A Multi-Language Performance Analysis System* ». in *10th International Conference on Computer Performance Evaluation - Modeling Techniques and Tools - Performance Tools'98*, Palma de Mallorca, Spain, September 1998

[Ear] « *The Earth Simulator Center* ». `http://www.es.jamstec.go.jp/esc/eng/ESC/index.html`

[FCL02] Felix Freitag, Jordi Caubet et Jesus Labarta. « *On the Scalability of Tracing Mechanisms* ». in B.Monien et R.Feldmann, éditeurs, *EuroPar 2002*, pages 97–104. LNCS, Springer-Verlag, 2002

[Fer98] Paulo Henrique Lemelle Fernandes. « *Méthodes numériques pour la solution de systèmes markoviens à grand espace d'états* ». PhD thesis, Institut National Polytechnique de Grenoble, France, février 1998

[FK99] Ian Foster et Carl Kesselman, éditeurs. « *The GRID, blueprint for a new computing infrastructure* ». Morgan Kaufmann, 1999

[Gar02] Bertrand Garrigues. « *Horloge globale pour grappe de PC de grande taille* ». Rapport de stage année speciale ENSIMAG, juin 2002

[GCdKdOSA01] C. Guilloud, J. Chassin de Kergommeaux, B. de Oliveira Stein et P. Augerat. « *Outil visuel d'administration système pour grappe de processeurs de grande taille* ». in *Renpar'2001*, pages 163–168. ASTI, 2001

[gdb] « *gdb* ». `http://sources.redhat.com/gdb/`

[GFB+03] Mark K. Gardner, Wu-chun Feng, Michael Broxton, Adam Engelhart et Gus Hurwitz. « *MAGNET : A Tool for Debugging, Analysis and Adaptation in Computing Systems* ». in *Proceedings of the 3rd IEEE/ACM International International Symposium on Cluster Computing and the Grid (CCGrid'2003)*, 12-15 May 2003

[gkr] « *gkrellm* ». http://web.wt.net/~billw/gkrellm/gkrellm.html

[GL96] William D. GROPP et Ewing LUSK. « *User's Guide for* mpich, *a Portable Implementation of MPI* ». Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6

[GLDS96] W.D. GROPP, E. LUSK, N. DOSS et A. SKJELLUM. « *A high-performance, portable implementation of the MPI message passing interface standard* ». *Parallel Computing*, 22(6) :789–828, septembre 1996

[gpr] « *gprof* ». http://www.gnu.org/manual/gprof-2.9.1/gprof.html

[Gra01] Emmanuel GRANGE. « *Visualisation de l'activité d'une grappe de processeurs* ». Rapport de Maîtrise, Juillet 2001

[GRCD98] F. GALILEE, J-L. ROCH, G. CAVALHEIRO et M. DOREILLE. « *Athapascan-1 : On-line Building Data Flow Graph in a Parallel Language* ». in *Proceedings of the IEEE International Conference on Parallel Architectures and Compilation Techniques, PACT'98*, pages 88–95, Paris France, Oct 1998

[Gro95] William GROPP. « *An Introduction to Performance Debugging for Parallel Computers* ». Rapport Technique MCS-P500-0295, Argonne National Lab, April 1995

[Hek97] Jessica P. et al. HEKMAN. « *Linux in a Nutshell* ». O'Reilly, 1997. ISBN : 2-7122-0752-1

[HM95] Steven T. HACKSTADT et Allen D. MALLONY. « *Visualizing Parallel Programs and Performance* ». *IEEE Computer Graphics and Applications*, 15(4) :12–14, 1995

[HMG+97] Jeffrey K. HOLLINGSWORTH, Barton P. MILLER, M. J. R. GONCALVES, Oscar NAIM, Zhichen XU et Ling ZHENG. « *MDL : A Language and Compiler for Dynamic Program Instrumentation* ». in *proc. of International Conference on Parallel Architectures and Compilation Techniques*, November 1997

[Hyp03] « *Hyper-Threading Technology* », 2003. www.intel.com/products/ht/hyperthreading_more.htm

[ID] « *Laboratoire ID-imag : Rapport d'activité 1997-2001* ». www-id.imag.fr/Laboratoire/Rapport/

[Inc] SGI Silicon Graphics INC. « *Linux Advanced Cluster Environment ACE* ». http://oss.sgi.com/projects/ace/

[INR] « *Institut National pour la Recherche en Informatique et en Automatique* ». http://www.inrialpes.fr

[Int] INTEL. « *SpeedStep* ». www.intel.com/support/processors/mobile/pentiumiii/ss.htm

[Int98] INTEL. « *Using the RDTSC Instruction for Performance Monitoring* », 1998. http://cedar.intel.com/software/idap/media/pdf/rdtscpm1.pdf

[Int01] INTEL. « *IA-32 Intel Architecture Software Developer's Manual vol.3 : Systeme Programming Guide* ». Intel, 2001. http://www.intel.com/design/pentium4/manuals/245472.htm

[JJZ01] Eric E. JOHNSON, Ha JIHENG et M. Baqar ZAIDI. « *Lossless Trace Compression* ». *IEEE transaction on computers*, 50(2) :158–173, Fevrier 2001

[Kal03] Alâa KALAÎ. « *Implantation d'un système de correction de dérive d'horloge pour grappe de PC* ». Rapport de stage, INPG Département Télécomunication, décembre 2003

[KCD+97] Péter KACSUK, José C. CUNHA, Gábor DÓZSA, João LOURENÇO, Tibor FADGYAS et Tiago ANTÃO. « *A Graphical Development and Debugging Environment for Parallel Programs* ». *Parallel Computing*, 22(13) :1747–1770, 1997

[Koj03] « *Kojak* », 2003. www.fz-juelich.de/zam/kojak/

[KOKM02] Seon Wook KIM, Patrick OHLY, Robert H. KUHN et Dmitri MOKHOV. « *A Performance Tool for Distributed Virtual Shared-Memory Systems* ». in *Proc. 4th IASTED Int. Conf. Parallel and Distributed Computing and Systems*, pages 755–760. Canada : Acta Press, 2002

[KWA+01] Eric KORPELA, Dan WERTHIMER, David ANDERSON, Jeff COBB et Lebofsky MATT. « *SETI@home : Massively Distributed Computing for SETI* ». *computer*, 2001. http://www.computer.org/cise/articles/seti.htm

[Lam] « *Lam-Mpi* ». http://www.lam-mpi.org/

[Lam78] Leslie LAMPORT. « *Time, Clocks, and the Ordering of Events in a Distributed System* ». *Communications of the ACM*, 21(7) :558–565, 1978

[LDM+01] LONDON, DONGARRA, MOORE, MUCCI, SEYMOUR et SPENCER. « *End-User Tools for Application Performance Analysis Using Hardware Counters* ». in *Acts of International Conference on Parallel and Distributed Computing Systems*, August 2001

[Let01] Vincent LETHUILLIER. « *Implémentation d'une technique de datation globale pour une grappe de PC* ». Rapport de Maîtrise, Université Joseph Fourier, Grenoble, France, Juillet 2001

[LGP+96] Jesus LABARTA, Sergi GIRONA, Vincent PILLET, Toni CORTES et Luis GREGORIS. « *DiP : A Parallel Program Development Environment* ». in *Euro-Par, Vol. II*, pages 665–674, 1996

[lin] « *linpack* ». `http://www.netlib.org/linpack/`

[lms] « *lm_sensor* ». `secure.netroedge.com/~lm78/`

[LO97] M. LOUKIDES et A. ORAM. « *Programmer avec les outils GNU* ». O'Reilly, 1997. ISBN 2-84177-010-9

[Lu02] Jingyi LU. « *Traçage d'application parallèles* ». Rapport de stage RICM 2ème année, Septembre 2002

[M4] GNU. « *manuel M4* ». `http://www.gnu.org/manual/m4-1.4/`

[Ma95] Barton P. MILLER et AL. « *The Paradyn Parallel Performance Measurement Tools* ». *IEEE Computer*, 28(11) :37–46, November 1995. Special issue on performance evaluation tools for parallel and distributed computer systems

[Mai96] E. MAILLET. « *Le traçage d'applications parallèles : conception et ajustement de qualité* ». PhD thesis, Institut National Polytechnique de Grenoble, France, Grenoble, 1996. `ftp://ftp.imag.fr/pub/bibliotheque/theses/1996/Maillet.Eric/`

[Mal] MALONY. « *Tau : Tuning and Analysis Utilities* ». http ://www.cs.uoregon.edu/research/paracomp/tau/

[Mar03] Cyrille MARTIN. « *Déploiement et contrôle d'applications parallèles sur grappes de grandes tailles* ». PhD thesis, Institut National Polytechnique de Grenoble, France, 2003

[Mas] Matt MASSIE. « *Ganglia Cluster Monitoring Toolkit* ». `www.millennium.berkeley.edu/ganglia/`

[MB02] Cyrille MARTIN et Wilfrid BILLOT. « *Lancement d'applications sur des grappes de grande taille* ». in *actes RENPAR*, 2002

[MCLD01] Shirley MOORE, David CRONK, Kevin LONDON et Jack DONGARRA. « *Review of Performance Analysis Tools for MPI Parallel Programs* ». in *proc. of 8th European PVM/MPI Users' Group Meeting*, pages 241–248. Lecture Notes in Computer Science 2131, September 23-26 2001

[MID01] MIDAM. « *Kid Paddle «Carnage Total»* », volume 2. dupuis, 2001. page 17

[Mil] MILLS. « *Network Time Protocol* ». `www.eecis.udel.edu/~mills/ntp.htm`

[Mil93] Barton P. MILLER. « *What to Draw ? When to Draw ? An Essay on Parallel Program Visualization* ». *Journal of parallel and distributed computing*, 18 :265–269, 1993

[Moh92] B. MOHR. « *Standardization of Event Traces Considered Harmful or Is an Implementation of Object-Independent Event Trace Monitoring and Analysis Systems Possible ?* ». in J.J. DONGARRA et B. TOURANCHEAU, éditeurs, *Proc. of the CNRS-NSF Workshop on Environments and Tools For Parallel Scientific Computing*, volume 6 de *Elsevier, Advances in Parallel Computing*, pages 103–124, St. Hilaire du Touvet, France, september 1992

[mpi] « *MPI Forum* ». `http://www.mpi-forum.org/`

[MR95] Tara M. MADHYASTHA et Daniel A. REED. « *Data Sonification : Do You See What I Hear ?* ». *IEEE Software*, 12(2) :45–56, March 1995

[MR01] Cyrille MARTIN et Olivier RICHARD. « *Parallel launcher for clusters of PCs* ». in London IMPERIAL COLLEGE PRESS, éditeur, *PARCO 2001, World Scientific*, 2001

[MS00] A.D. MALONY et S. SHENDE. « *Performance Technology for Complex Parallel and Distributed Systems* ». in *Proc. Third Austrian-Hungarian Workshop on Distributed and Parallel Systems, DAPSYS 2000*, pages 37–46, 2000

[MS01] N. MATTHEW et R. STONES. « *Programmation Linux* ». éditions Eyrolles, 2ème édition, 2001. ISBN : 2-212-09129-x

[MSB+03] A.D. MALONY, S. SHENDE, R. BELL, K. LI, L LI et N. TREBON. « *Advances in the TAU Performance System* ». *PADC, Kluwer, Norwell, MA*, 2003

[MT95] Eric MAILLET et Cécile TRON. « *On Efficiently Implementing Global Time for Performance Evaluation on Multiprocessor Systems* ». *Journal of Parallel And Distributed Computing*, 28 :84–93, 1995

[MW03] Bernd MOHR et Felix WOLF. « *KOJAK-A Tool Set for Automatic Performance Analysis of Parallel Programs* ». in Harald KOSCH, László BÖSZÖRMÉNYI et Hermann HELLWAGNER, éditeurs, *Proc. of Euro-Par 2003*, volume 2790 de *LNCS*, pages 1301–1304. Springer Verlag, August 2003. ISBN3-540-40788-X

[Myr] « *Myrinet* ». `http://www.myri.com`

[NAW+96] W. E. NAGEL, A. ARNOLD, M. WEBER, H. C. HOPPE et K. SOLCHENBACH. « *VAMPIR : Visualization and Analysis of MPI Resources* ». *Supercomputer*, 12(1) :69–80, 1996

[Ott01] F.G. OTTOGALLI. « *Observations et analyses quantitatives multiniveaux d'applications à objets reparties* ». PhD thesis, Université Joseph Fourier, Grenoble, 2001. In French. `http://www-mediatheque.imag.fr`

[PAL] PALLAS. « *VAMPIR : Visualization and Analysis of MPI Programs* ». http://www.pallas.com/e/products/vampir/index.htm

[PAP02] PAPITEAM. « *PAPI User's Guide* ». University of Tennesse, 2002. http://icl.cs.utk.edu/projects/papi/

[PAP03] « *PAPI homepage* », 2003. icl.cs.utk.edu/papi/

[PCL] « *PCL Perf Counter Lib* ». http://www.fz-juelich.de/zam/PCL/

[Pel01] F. PELLEGRINI. « *SCOTCH 3.4 User's guide* ». Research report rr-1264-01, LaBRI, 2001. www.labri.fr/Perso/~pelegrin/scotch/

[per] « *perfex* ». http://www.cs.utk.edu/~browne/dod/cewes/tools/perfex.html

[Per03] « *Perfctr* », 2003. http://user.it.uu.se/~mikpe/linux/perfctr/

[ps] « *ps* ». http://procps.sourceforge.net/

[Rab] « *Rabbit* ». http://www.scl.ameslab.gov/Projects/Rabbit/

[Rad] Jacek RADAJEWSKI. « *bWatch* ». www.sci.usq.edu.au/staff/jacek/bWatch/

[RAM+01] Bruno RICHARD, P. AUGERAT, N. MAILLARD, S. DERR, S. MARTIN et C. ROBERT. « *I-Cluster : Reaching TOP500 performace using mainstream hardware* ». Rapport Technique HPL-2001-206, hp-labs, 2001. http://www.hpl.hp.com/techreports/2001/HPL-2001-206.html

[Ray94] Michel RAYNAL. Ré-exécution et analyse de calculs répartis. Rapport de recherche INRIA N°2257, avril 1994

[RCDG] Jean-Louis ROCH, Gerson CAVALHEIRO, Mathias DOREILLE et François GALILÉE. « *Athapascan-1 Une interface de programmation pour les applications parallèles* ». poster : http://www-id.imag.fr/Logiciels/ath1/a1_intro.ps.gz

[RDBC+03] Michel RONSSE, Koen DE BOSSCHERE, Mark CHRISTIAENS, Jacques Chassin de KERGOMMEAUX et Kranzlmuller DIETER. « *Record/Replay for nondeterministic program executions* ». *Coomunication of the ACM*, 46(9) :62–67, September 2003

[Rev04] Remi REVIRE. « *Ordonnancement de graphe dynamique de tâches sur architecture de grande taille. Régulation par dégénération séquentielle et distribuée* ». PhD thesis, Institut National Polytechnique de Grenoble, France, 2004. À soutenir

[RGR03] Jean-Louis ROCH, Thierry GAUTIER et Rémi REVIRE. « ATHAPASCAN : *an API for Asynchronous Parallel Programming* ». Rapport Technique 276, INRIA, février 2003

[RM02] Philip C. ROTH et Barton P. MILLER. « *Deep Start : A Hybrid Strategy for Automated Performance Problem Searches* ». in *proc of Euro-Par'02*, Paderborn, Germany, August 2002

[Rob96] Paul ROBERT. « *Le Petit robert* ». dictionnaires Le Robert, 1996. ISBN : 2-85036-469-X

[RSS+95] Daniel A. REED, Keith A. SHIELDS, Will H. SCULLIN, Luis F. TAWERA et Christopher L. ELFORD. « *Virtual Reality and Parallel Systems Performance Analysis* ». *IEEE Computer*, 28(11) :57–67, 1995

[Sch70] Lucien SCHNEIDER. « *Dictionnaire Français-Esquimau du parler de l'ungava* ». Les Presses de l'université de Laval, 1970

[Sch03] Baltazar SCHIRMER. Rapport de DEA Informatique Systèmes et Communications, Université Joseph Fourier, juin 2003

[SET] « *seti@HOME* ». http ://setiathome.ssl.berkeley.edu/

[Sey01] London Moore Mucci SEYMOUR. « *The PAPI Cross-Platform Interface to Hardware Performance Counters* ». *Department of Defense Users' Group Conference Proceedings*, 2001

[SGI] SGI. « *Performance Co-Pilot* ». `http://oss.sgi.com/projects/pcp/`

[SM01] S. SHENDE et A.D. MALONY. « *Integration and Application of the TAU Performance System in Parallel Java Environments* ». in *Proc. of the Joint ACM Java Grande - ISCOPE 2001 Conference*, June 2001

[SMC+98] S. SHENDE, A.D. MALONY, J. CUNY, K. LINDLAN, P. BECKMAN et S. KARMESIN. « *Portable Profiling and Tracing for Parallel Scientific Applications using C++* ». in *Proceedings of ACM SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, pages 134–145, August 1998

[Str01] B. STROUSTRUP. « *Le Langage C++* ». CampusPress, 2001. ISBN : 2-7440-1089-8

[Tak] « *Takakow : une application parallèle de dynamique moléculaire audessus d'Athapascan* ». `http://www-apache.imag.fr/~gautier/SimBio/takakaw.html`

[tcp] « *tcpdump* ». `http://www.tcpdump.org/`

[TF02] Hong-Linh TRUONG et Thomas FAHRINGER. « *SCALEA : A Performace Analysis Tool for Distributed and Parallal Programs* ». in B.MONIEN et R.FELDMANN, éditeurs, *EuroPar 2002*, pages 75–85. LNCS, Springer-Verlag, 2002

[topa] « *top* ». `http://procps.sourceforge.net/`

[Topb] `http://www.top500.org/`

[Tot] TOTALVIEW. « *Totalview* ». `http://www.etnus.com/Products/TotalView/`

[ups] « *Upshot* ». `http://www-fp.mcs.anl.gov/~lusk/upshot/`

[Vam] « *VampirTrace* ». `www.pallas.com/e/products/vampir/index.htm#vampirtrace`

[vax83] « *VAXclusters* », 1983. `http://www.montagar.com/dfwcug/VMS_HTML/timeline/1983-1.htm`

[Ver] VERRALL. « MPI-POVRAY ». `http://www.verrall.demon.co.uk/mpipov`

[VTu03] « *VTune* », 2003. `www.intel.com/software/products/vtune/`

[Wap02] WAPITI. « *Super Ordinateur* ». *Wapiti, le mensuel pour les 7-13 ans*, Octobre 2002

[WBS+00] Ching-Farn Eric WU, Anthony BOLMARCICH, Marc SNIR, David WOOTTON, Farid PARPIA, Anthony CHAN, Ewing L. LUSK et William GROPP. « *From Trace Generation to Visualization : A Performance Framework for Distributed Parallel Systems* ». in *Supercomputing*, 2000

[WMM] WINDISCH, MOHR et MALONY. « *A Brief Technical Overview of the TAU Tools* ». www.cs.uoregon.edu/research/paracomp/tau/tau-desc.ps.gz

[xlo] « *xload* ». `http://s.sudre.free.fr/Software/Xload.html`

[XML] « *http://www.xml.org/* »

[xmp] « *xmpi* ». `http://www.lam-mpi.org/software/xmpi/`

[xos] « *xosview* ». `http://xosview.sourceforge.net/`

[Xtr] « *XtremWeb, A Global Computing Experimental Platform* ». `http://www.lri.fr/~fedak/XtremWeb/introduction.php3`

[ZFV02] F. ZARA, F. FAURE et J-M. VINCENT. « *Physical cloth simulation on a PC cluster* ». in *Fourth Eurographics Workshop on Parallel Graphics and Visualization*, pages 105–112, September 2002

[ZMN99] Xu ZHICHEN, Barton P. MILLER et Oscar NAIM. « *Dynamic Instrumentation of Threaded Applications* ». in *proc. of 7th ACM SIGPLAN, Symposium Principles and Practice of Parallel Programming*, pages 49–59, Atlanta, Georgia, May 1999

*Un poète doit laisser des traces de son passage,*
*non des preuves.*
*Seules les traces font rêver.*

René Char.