Environnements de visualisation pour l'évaluation des performances des systèmes parallèles : étude, conception et réalisation

Yves Arrouye

Thèse présentée par

Yves ARROUYE

pour obtenir le grade de

Docteur de l'Institut National Polytechnique de Grenoble
(arrêté ministériel du 30 mars 1992).

Spécialité : informatique.

Environnements de visualisation pour l'évaluation des performances des systèmes parallèles : étude, conception et réalisation

Date de soutenance : fin novembre 1995

Président du jury :

Guy MAZARÉ

Rapporteurs :

Jean-Michel FOURNEAU
Claude JARD

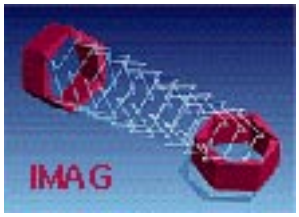
Examinateurs :

Pierre CUBAUD
Farid OUABDESSELAM
Brigitte PLATEAU

Thèse préparée au sein du Laboratoire de Génie Informatique
(Institut de Mathématiques Appliquées de Grenoble)
sous la direction du professeur Brigitte PLATEAU.

Environnements de visualisation pour l'évaluation des performances des systèmes parallèles : étude, conception et réalisation

Yves ARROUYE



A P A C H E
ALGORITHMIQUE PARALLELE ET PARTAGE DE CHARGE

Le projet APACHE dans le cadre duquel ce travail a été réalisé est soutenu conjointement par le CNRS, l'INPG, l'INRIA et l'UJF.

LES APPLICATIONS parallèles sont de plus en plus complexes et, même si les environnements de développement s'améliorent, de plus en plus difficiles à mettre au point que ce soit du point de vue de leur correction ou, dans le cadre qui nous intéresse, de leurs performances. Un outil intéressant pour la mise au point des performances d'une application est la visualisation de son exécution.

Le travail présenté dans ce mémoire porte sur l'étude, la conception et la réalisation d'un environnement interactif extensible de visualisation pour l'évaluation de performance de systèmes parallèles. Un tel environnement permet de reconstituer le comportement d'une application d'après des traces d'exécution et facilite, par le calcul et la présentation d'indices de performances, la compréhension de son fonctionnement et la détection de ses problèmes de performance.

Nous commençons par préciser les concepts importants dans un environnement de visualisation et présentons ensuite l'état de l'art en la matière. Nous décrivons ensuite la conception et la réalisation de l'environnement Scope. Cet environnement est validé par rapport aux objectifs initiaux et son utilité est démontrée par l'étude de sessions d'évaluation de performance d'applications concrètes.

Mots clés : parallélisme, évaluation de performance, visualisation interactive, simulation dirigée par les traces, manipulation de traces d'exécution, environnements interactifs, environnements extensibles, programmation visuelle.

PARALLEL applications are getting increasingly complex every day. Even with modern programming environments, they are extremely difficult to tune, whether it is for correctness or for performance. For this last point, an interesting tool is the visualization of the application execution.

We present here our work concerning the study, the design and the implementation of an extensible interactive visualization environment for the performance evaluation of parallel systems. Such an environment lets one reconstruct the behaviour of an application based on execution traces. It also computes and displays various performance indices representative of an application execution. These performance data facilitate the understanding of the execution of the applications and the detection of their performance problems.

We start by defining the important concepts of a visualization environment. This presentation is followed by a tour of the state of the art in the domain. Then we describe the design and implementation of the Scope environment. This environment is validated using the goals we presented first, and its usefulness is demonstrated by the performance evaluation of concrete applications.

Keywords: parallelism, performance evaluation, interactive visualization, trace-driven simulation, execution traces manipulation, interactive environments, extensible environments, visual programming.

Table des matières

Introduction **1**

I Présentation 9

1 Objectifs **11**
1.1 Extensibilité 12
1.2 Généricité 13
1.3 Personnalisation 15
1.4 Interactivité 16
1.5 Puissance 18
1.6 Convivialité 20
1.7 Conclusion 21

2 État de l'art **23**
2.1 Principes 24
2.1.1 Traces d'exécution 24
2.1.2 Simulation 30
2.1.3 Analyse de données 30
2.1.4 Visualisation 31
2.2 Outils et environnements existants 33
2.2.1 ParaGraph 33

2.2.2 XPVM 36
2.2.3 Upshot 37
2.2.4 Pablo 39
2.2.5 Paradyn 44
2.3 Comparaison des environnements 48
2.4 Conclusion 48

Bibliographie **56**

II L'environnement Scope **57**

1 Architecture de Scope **59**
1.1 Environnement de développement 60
1.2 Organisation générale 61
1.2.1 Découpage logique 61
1.2.2 Principes directeurs 63
1.3 Programmation visuelle 65
1.3.1 Principes 66
1.3.2 Eve (environnement visuel extensible) 67
1.4 Composants spécifiques 78
1.5 Composants génériques 78
1.5.1 Analyse des données 79
1.5.2 Visualisation 79
1.6 Contrôle des sessions 80
1.6.1 Choix du modèle de programmation 80
1.6.2 Déroulement de la session d'évaluation 82
1.7 Conclusion 85

2 Lecture de traces et simulation **87**
2.1 Lecture de traces 88
2.1.1 Lecteurs de traces 88
2.1.2 Protocoles de la lecture de trace 90
2.1.3 Production des événements 91
2.1.4 Présentation des événements 94
2.2 Simulation 94
2.2.1 Principes 95
2.3 Support des applications sur Transputer 98
2.3.1 Prise et format de trace 99

2.3.2 Lecture de traces . . . 103
2.3.3 Simulation . . . 106
2.4 Support des applications PVM . . . 110
2.4.1 Prise et format de trace . . . 111
2.4.2 Lecture de traces . . . 113
2.4.3 Simulation . . . 115
2.4.4 Performances des composants réalisés . . . 120
2.5 Réutilisation des composants logiciels . . . 128
2.6 Conclusion . . . 130

3 Déplacement dans le temps 131
3.1 Présentation du problème . . . 132
3.2 Construction d'un film d'exécution . . . 133
3.3 Organisation du déplacement . . . 134
3.3.1 Tâches incombant à l'environnement . . . 134
3.3.2 Mécanismes de sauvegarde et de restauration . . . 136
3.3.3 Interaction avec l'utilisateur . . . 137
3.4 Stratégies et heuristiques . . . 139
3.4.1 Stratégies d'enregistrement . . . 139
3.4.2 Heuristiques d'adaptation . . . 140
3.5 Performances . . . 142
3.6 Conclusion . . . 142

4 Visualisation 145
4.1 Principes . . . 146
4.1.1 Types de visualisations . . . 146
4.1.2 Caractéristiques des visualisations . . . 149
4.2 Visualisations de Scope . . . 151
4.2.1 Personnalisation . . . 152
4.2.2 Interactivité . . . 154
4.2.3 Extensibilité . . . 155
4.2.4 Représentations tridimensionnelles . . . 157
4.3 Visualisation de graphes . . . 161
4.3.1 Principe . . . 161
4.3.2 Personnalisation . . . 162
4.3.3 Placement des sommets . . . 163
4.3.4 Interaction . . . 165
4.4 Conclusion . . . 165

5 Schémas d'évaluation **167**
5.1 Composants d'analyse et de visualisation 168
5.1.1 Conditions d'expérience 169
5.1.2 Construction d'un histogramme 170
5.1.3 Calcul du surcoût lié aux échanges de données 173
5.2 Schémas et composants intelligents 179
5.2.1 Principes 181
5.2.2 Surveillance de données 182
5.3 Conclusion 182

6 Utilisation de l'environnement **185**
6.1 Présentation 185
6.2 Conditions d'expérience 187
6.3 Histogramme des temps d'attentes 188
6.4 Détection automatique d'un problème 192
6.5 Conclusion 195

A Fonctionnement des simulateurs **197**
A.1 Support des applications sur Transputer 197
A.2 Support des applications PVM 200

B Coût de développement **209**

Bibliographie **216**

III Conclusion **217**

Bilan et perspectives **219**

IV Annexes **229**

A Modèles de programmation **231**
A.1 Programmation par envoi de messages 232
A.1.1 CSP (processus séquentiels communicants) 232
A.1.2 PVM (machine parallèle virtuelle) 235
A.1.3 MPI (interface d'envoi de messages) 239
A.2 Communication par variables partagées 243
A.3 Modèles du parallélisme de données 244

A.3.1 HPF (Fortran hautes performances) 245
A.4 Programmation client-serveur et objet 246
A.4.1 Athapascan 246

Bibliographie **250**

V Bibliographie commentée **251**

Table des figures

2.1 Instants de prise de trace 25
2.2 Événements pour une communication CSP 26
2.3 ParaGraph 35
2.4 XPVM 37
2.5 Upshot 38
2.6 Upshot (nouveautés) 40
2.7 Pablo 41
2.8 Pablo (visualisation expérimentale) 42
2.9 Pablo (réalité virtuelle) 43
2.10 Paradyn 45
2.11 Paradyn (détection d'un problème) 46
2.12 Paradyn (illustration d'un problème) 47

1.1 Découpage de Scope en parties 64
1.2 Programmation visuelle. 67
1.3 Délai de propagation en programmation visuelle. 70
1.4 Multiplexage des entrées d'un composant. 71
1.5 Types de ports et connexions en Eve. 71
1.6 Structures de contrôles en programmation visuelle. 73
1.7 Bibliothèque de composants. 75
1.8 Ajout d'un composant dans un programme visuel. 76
1.9 Construction d'un programme visuel 77
1.10 Constituants d'une trace Scope 81

1.11 Programme visuel associé à un format de trace 83
1.12 Panneau de contrôle de session 84

2.1 Principe de la simulation 95
2.2 Prise de trace avec TAPE 100
2.3 Composition d'une trace TAPE 102
2.4 Lecture d'une trace TAPE 105
2.5 Hiérarchie des classes d'événements de TAPE 107
2.6 Hiérarchie des classes d'événements de TAPE/PVM 116
2.7 Incertitude des temps de communication en PVM 118
2.8 Nombre d'événements PVM lus par seconde 123
2.9 Nombre d'événements PVM simulés par seconde 124
2.10 Nombre de mises à jour du graphe d'état PVM par seconde 125

3.1 Construction d'un film d'exécution 135
3.2 Déplacement dans le temps 136
3.3 Contrôles du déplacement dans le temps 138
3.4 Sauvegardes inappropriées 141

4.1 Visualisations statistiques 147
4.2 Visualisations spécifiques 148
4.3 Gestionnaire d'associations de formes 153
4.4 Caméra virtuelle de Scope 158
4.5 Méthodes de rendu des représentations tridimensionnelles 159
4.6 Une séquence de l'animation tridimensionnelle de l'état d'une application parallèle 162
4.7 Arrangements d'un graphe en trois dimensions 164
4.8 Inspecteurs (attributs d'une application OUF/TRITA) 166

5.1 Histogramme de distribution sur un intervalle 171
5.2 Interface du composant de construction d'histogramme 172
5.3 Échantillonnage par intervalles de temps 175
5.4 Échantillonnage par phases 175
5.5 Interface du composant de calcul de surcoût 178
5.6 Interface du composant de surveillance de données 183
5.7 Action du composant de surveillance de données 184

6.1 Algorithme de la FFT 2D parallèle 186
6.2 Temps d'attente en réception d'une FFT 2D (1/3) 189
6.3 Temps d'attente en réception d'une FFT 2D (2/3) 190

6.4 Temps d'attente en réception d'une FFT 2D (3/3) 191
6.5 Obtention du surcoût lié aux échanges de données 193
6.6 Configuration du composant de surveillance 194
6.7 Suspension du déroulement de la trace 195
6.8 Surcoût de communication trop important 196

A.1 Simulation de la naissance et de la mort d'une tâche OUF/TRITA . 198
A.2 Simulation d'une communication entre deux tâches OUF/TRITA . 199
A.3 Simulation de la naissance et de la mort d'une tâche PVM 201
A.4 Simulation de la transformation de données de PVM 202
A.5 Simulation du passage d'une barrière de PVM 203
A.6 Simulation de la fin d'une émission de PVM 204
A.7 Simulation d'une communication de PVM avec blocage 205
A.8 Simulation d'une communication de PVM sans blocage 206

A.1 Modèle de programmation CSP. 233
A.2 Communications bloquantes synchrones 234
A.3 Communications bloquantes asynchrones 236
A.4 Communications non bloquantes 237
A.5 Opérations globales simples 239
A.6 Opérations globales complexes 242
A.7 Modèle de programmation Athapascan 247

Liste des tableaux

2.1 Comparaison des environnements 49

2.1 Protocole des lecteurs de traces 92
2.2 Protocole des événements 93
2.3 Protocole des horloges 93
2.4 Événements générés par TAPE 103
2.5 États des tâches Transputer 108
2.6 États des liens Transputer 108
2.7 Attributs du graphe d'état Transputer 109
2.8 Attributs des nœuds Transputer 109
2.9 Attributs des liens Transputer 110
2.10 Événements générés par TAPE/PVM 113
2.11 États des tâches PVM 119
2.12 États des liens PVM 119
2.13 Attributs du graphe d'état PVM 120
2.14 Attributs des tâches PVM 121
2.15 Attributs des liens PVM 121
2.16 Validation des mesures de performances 127

5.1 Coût du composant de construction d'histogramme 174
5.2 Coût du composant de calcul de surcoût 180

A.1 États des nœuds Transputer 198
A.2 États des liens Transputer 198

A.3 États des tâches PVM 200
A.4 États des liens PVM 201

B.1 Taille du code de l'environnement 210

Introduction

LES APPLICATIONS scientifiques deviennent chaque jour un peu plus complexes et demandent de plus en plus de puissance de calcul. Alors que ces applications atteignaient — voire dépassaient — les limites des systèmes sur lesquels elles fonctionnaient il y a quelques années, le parallélisme a apporté un élément de réponse au problème toujours plus aigu de l'augmentation des demandes en ressources de calcul. Cependant les applications parallèles, à l'instar des systèmes sur lesquels elles sont destinées à être exécutées, sont extrêmement complexes et difficiles à réaliser.

Il y a encore quelques années les machines parallèles étaient la plupart du temps « nues », sans système d'exploitation. Les applications étaient développées sur une station de travail détachée de la machine parallèle puis chargées sur cette dernière pour leur exécution. Aujourd'hui la plupart d'entre elles sont fournies avec un système complet facilitant leur exploitation et le développement des applications sur la machine elle-même. Les environnements de programmation disponibles comprennent de plus en plus d'outils dont certains pour la mise au point des applications parallèles. Il n'est pas pour autant beaucoup plus facile à l'heure actuelle de comprendre ce qui se passe pendant l'exécution d'une application sur une machine parallèle que ce ne l'était avec les générations de machines précédentes.

Le besoin de comprendre la manière dont les tâches d'une application parallèle interagissent au cours de leur exécution se fait sentir à deux niveaux : pour la

mise au point des applications, c'est-à-dire pour garantir leur correction, et pour l'évaluation et l'amélioration de leurs performances, cadre dans lequel s'inscrit notre travail et dont la finalité est de s'assurer qu'une application utilise au mieux les ressources de la machine sur laquelle elle est exécutée. Il est en effet primordial qu'une application parallèle soit le plus efficace possible et donc s'exécute rapidement. Une augmentation de la rapidité de l'application permet soit de diminuer le coût de son exécution sur des machines extrêmement coûteuses soit de lui faire traiter un même problème avec des données plus importantes ou un modèle plus précis sans accroissement de la durée de traitement. On comprend tout l'intérêt de l'optimisation pour les performances si l'on considère des applications comme la prévision météorologique pour lesquelles il est hautement souhaitable que le temps d'obtention de la prévision soit inférieur au délai de cette dernière...

L'évaluation des performances d'une application parallèle consiste en l'analyse de son comportement et de son utilisation des ressources à sa disposition. Le comportement d'une application est représenté par l'évolution de son état au cours de son exécution, un changement d'état intervenant lorsque l'application effectue un effet de bord, comme par exemple lorsque deux tâches échangent des données, se synchronisent ou passent d'une phase de calcul à une autre. Les ressources dont l'utilisation est évaluée sont le temps de calcul, l'espace mémoire, le réseau de communication de la machine parallèle, etc. Connaissant les ressources à la disposition d'une application et la manière dont celle-ci les utilise, il est possible de construire des indicateurs ou indices de performance représentatifs du comportement de l'application. La recherche de problèmes de performances se fait alors en observant la variation de ces indicateurs au cours de l'exécution de l'application dans le but de trouver une anomalie dans son comportement.

Un certain nombre de questions se posent alors. Quels sont les données nécessaires à la production de ces indices de performance ? Comment ces données sont-elles obtenues ? Par quels moyens les transforme-t-on en indicateurs ? Avec quels outils et sous quelle forme effectue-t-on l'observation de ces informations et leur analyse ?

La réponse générale qui est faite à la première question est : tout ce qu'il est possible de savoir sur l'application. C'est en effet l'approche de tous les outils d'analyse de performance d'applications parallèles qui ont été développés ces dernières années. Ne sachant pas *a priori* quelles informations l'utilisateur désire connaître ces outils demandent à avoir le plus d'informations possibles pour être à même de répondre à un grand nombre de questions sur l'exécution de l'applica-

tion. Cette approche est un peu grossière mais elle est en partie nécessaire à cause de l'indéterminisme des applications parallèles : si l'on ne prend que quelques informations et que l'on souhaite en savoir plus après avoir découvert un problème de performance, rien ne garantit qu'une nouvelle exécution de l'application pour obtenir plus d'informations exhibera ce même problème. Les outils d'évaluation de performance se prévalent contre ce danger en éliminant le besoin de réexécuter une application pour obtenir un complément d'information.

L'obtention des données nécessaires à la production des indices de performance est faite au cours de l'exécution de l'application par des outils de prise d'information appelés traceurs pour les performances. Les traceurs surveillent l'application et enregistrent chacun de ses changements d'état sous forme d'événements décrivant exactement la transition effectuée par l'application (qui a provoqué le changement d'état, à quel instant, pourquoi, dans quel contexte, etc.). La suite de ces événements est appelée trace de l'exécution de l'application et c'est elle qui sert de base à l'évaluation de performance.

Les indicateurs de performance d'une application sont obtenus à partir des événements décrivant son comportement au cours du temps. Suivant le type d'indicateur désiré les outils employés varient mais ce sont la plupart du temps des outils statistiques et d'analyse de données. Les environnements d'évaluation de performance emploient également des simulateurs permettant de reconstruire la suite des états de l'application à partir des événements de la trace.

Le dernier problème est celui de l'analyse des informations produites à partir de la trace. Le volume de données qu'elles représentent peut réellement être énorme : si une application comporte un millier de tâche et change d'état tous les dixièmes de seconde — ce qui est en fait une application assez lente —, cette application tracée produira neuf millions d'événements en un seul quart d'heure d'exécution. Même si les informations contenues dans ces événements sont réduites au point d'avoir un indicateur et un seul pour chaque type d'information et par tâche il n'en reste pas moins le délicat problème du suivi des variations d'un millier d'indicateurs...

La seule réponse à peu près satisfaisante qui ait été donnée à ce problème est de présenter les informations tirées de la trace sous forme graphique en partant du constat qu'un graphique est quelque chose de familier qui plus est est capable de représenter un assez grand nombre d'informations sous forme synthétique. C'est de cette idée que sont nés il y a quatre ans les premiers environnements de visua-

lisation pour l'évaluation de performance des systèmes parallèles. Ces environnements ont connu un succès immédiat et sont aujourd'hui encore les seuls outils permettant de voir réellement le comportement d'une application parallèle.

Notre travail porte sur l'étude, la conception et la réalisation d'un tel environnement de visualisation. Ce travail est réalisé dans le groupe de travail ALPES (ALgorithmes Parallèles et Évaluation de Systèmes, TRON *et al.* 1992) du projet APACHE (Algorithmique Parallèle et pArtage de CHargE, PLATEAU 1994), dont les objectifs sont l'étude, la conception et la mise au point d'un environnement de programmation pour machines parallèles. Cet environnement offrira un accès à toute la chaîne du développement et de la mise au point des applications parallèles, de la programmation des application avec un langage de haut niveau fortement parallèle et efficace à l'optimisation des programmes pour les performances en passant par le placement statique des composantes des applications, la répartition dynamique de charge durant leur exécution et la prise de mesures pour l'évaluation des performances ou pour la réexécution déterministe des applications à des fins de mise au point. Dans un tel environnement les outils de visualisation pour l'évaluation des performances sont à la fois en fin de chaîne puisque la visualisation ne peut être faite qu'une fois l'application exécutée pour produire des traces et en début de cycle car la détection d'un problème de performance entraîne une modification de l'application pour le corriger, puis une recompilation, un nouveau placement, etc.

Au cours de notre travail nous nous sommes attachés à définir les objectifs de l'évaluation de performance à travers la visualisation afin de réaliser un environnement qui permette d'atteindre ces objectifs. Nous avons également accordé beaucoup d'importance à la généricité des outils développés et à la facilité d'extension de l'environnement pour garantir sa pérennité et ses capacités d'évolution au fur et à mesure de l'avancée de l'état de l'art en matière de visualisation pour les performances ou de programmation parallèle. Nous avons également mis un soin particulier dans la définition de l'interface de l'environnement et de son interaction avec l'utilisateur pour que celui-ci puisse analyser ses applications avec le maximum d'efficacité et le minimum de contraintes et de distractions. Enfin, plutôt que de reproduire dans notre environnement la majorité des idées ou des capacités des outils existants, notamment au niveau des visualisations, nous avons cherché à développer de nouveaux outils, de nouvelles techniques et des manières originales d'aborder la conception d'un environnement de visualisation pour l'évaluation de performance de systèmes parallèles.

Le fruit de ces réflexions et de ces efforts est un prototype d'environnement à l'architecture et aux capacités originales, que nous présentons dans ce mémoire.

Contenu de ce document

La première partie de ce mémoire présente le problème de la visualisation pour l'évaluation de performance des applications parallèles.

Le chapitre 1, « Objectifs », décrit les capacités que nous souhaitons trouver dans un environnement de visualisation pour l'évaluation de performance. Ces objectifs nous serviront à la fois de critères d'évaluation des environnements existant et de cahier des charges pour l'environnement que nous avons conçu au cours de notre travail.

Le chapitre 2, « État de l'art », présente l'état de l'art en la matière. Nous abordons tout d'abord les principes généraux des environnements d'évaluation de performance auxquels nous nous intéressons, de la production des traces d'exécution en amont de l'environnement à leur lecture et leur traitement pour la production de visualisations. Nous présentons ensuite quelques environnements d'évaluation de performance représentatifs de l'état de l'art. La qualité de ces environnements est évaluée principalement en fonction des objectifs que nous avons préalablement fixés.

La deuxième partie est consacrée à la description de notre travail c'est-à-dire la conception et la réalisation d'un environnement d'évaluation de performance, Scope, qui soit à même de réaliser les objectifs que nous avons définis.

Le chapitre 1, « Architecture de Scope », détaille l'architecture de l'environnement et les critères qui ont orienté nos choix de conception. Nous présentons dans ce chapitre le langage de programmation visuelle qui est le cœur de l'environnement et permet par exemple son extension ou le support transparent de multiples modèles de programmation et des outils spécifiques qui peuvent leur être associés.

Le chapitre 2, « Lecture de traces et simulation », couvre la lecture de traces et la simulation telles qu'elles sont faites dans Scope. Nous présentons les principes permettant à Scope de manipuler des traces sans pour autant être dépendant de leur syntaxe ou de leur sémantique, ainsi que la manière dont sont conçus les simulateurs disponibles dans l'environnement. Puis nous détaillons la réalisation

des outils nécessaires à la lecture de traces et à la simulation de deux modèles de programmation, ces réalisations permettant de valider la capacité de Scope à traiter différents modèles. Une évaluation des performances d'un lecteur de traces et d'un simulateur nous permettent d'évaluer l'efficacité de ces composants et l'influence de nos choix de conception sur la rapidité de l'environnement.

Le chapitre 3, « Déplacement dans le temps », traite du problème délicat de l'exploration interactive d'une exécution grâce à la possibilité de se déplacer vers un instant arbitraire de cette exécution. Après avoir présenté un moyen d'effectuer ces déplacements nous envisageons plusieurs stratégies pour le rendre efficace. Elles sont complétées par des heuristiques dont le but est de rendre les stratégies plus « intelligentes » et mieux adaptées aux besoins ou au comportement de l'utilisateur, afin de diminuer le coût du déplacement dans le temps.

Le chapitre 4, « Visualisation », traite des visualisations permettant de représenter les informations disponibles sur le comportement d'une application pendant son exécution. Nous présentons dans ce chapitre les mécanismes présents dans Scope permettant de fournir des visualisations cohérentes, personnalisables et extensibles. Nous décrivons ensuite les visualisations développées pour Scope et détaillons la réalisation d'une visualisation tridimensionnelle présentant un graphe et supportant une grande liberté de personnalisation ainsi que diverses interactions avec l'utilisateur.

Le chapitre 5, « Schémas d'évaluation », s'intéresse aux schémas d'analyse et de visualisation permettant de calculer et d'exploiter des données de performance. Nous présentons la réalisation de deux composants après avoir motivé leur besoin et justifié leur rôle pour l'évaluation de performance d'applications parallèles. Nous évaluons pour chacun d'eux leur coût de développement et leur participation au temps nécessaire au traitement d'une trace d'exécution dans notre environnement. Nous nous intéressons ensuite aux composants « intelligents » capables de prise de décision et en présentons les principes et l'intérêt. Nous donnons alors un exemple de réalisation d'un tel composant.

Le chapitre 6, « Utilisation de l'environnement », décrit l'utilisation des composants présentés au chapitre précédent. Leur exploitation est démontrée à travers deux exemples de l'étude des performances d'une application parallèle réelle dont nous faisons varier les paramètres. Nous montrons également avec ces exemples la manière dont on construit un schéma d'évaluation de performance adapté à sa tâche à partir de composants de base.

L'annexe A, « Fonctionnement des simulateurs », décrit en détail les mécanismes internes des simulateurs réalisés pour les modèles de programmation supportés par Scope. Elle complète le chapitre 2 qui ne présente que les principes généraux de ces simulateurs.

L'annexe B, « Coût de développement », présente quelques données quantitatives et qualitatives sur l'effort de développement que Scope représente.

La troisième partie du mémoire est réservée au bilan de notre travail ainsi qu'aux perspectives de recherche qui s'offrent d'une part pour compléter et améliorer le prototype d'environnement d'évaluation de performance que nous avons réalisé et d'autre part pour la conception de nouvelles techniques pour l'évaluation de performance des applications parallèles.

La fin de ce document contient des références que le lecteur pourra trouver utiles au cours de la lecture de ce mémoire.

L'annexe A, « Modèles de programmation », présente les principaux modèles de programmation actuellement utilisés pour l'écriture des applications parallèles. Lorsqu'un de ces modèles est cité au cours de ce mémoire la discussion se poursuit en supposant connues les principes exposés dans la partie correspondante de cette annexe.

Cette annexe est suivie d'une bibliographie commentée reprenant une partie des références bibliographiques que l'on peut trouver à la fin de chacune des parties précédentes.

I

Présentation

1

Objectifs

L'ÉVALUATION de performance d'applications parallèles à travers l'utilisation d'un outil de visualisation dirigé par les traces est une tâche qui devient courante. Mais les outils ou environnements de visualisation pour les performances qui sont actuellement disponibles ont des interfaces et des capacités très variées, ne sont pas tous capables de faire les mêmes choses ou encore diffèrent dans la manière d'obtenir ce que l'on cherche. Cette prolifération d'outils différents est principalement due au fait que chaque utilisateur ou communauté d'utilisateurs a sa propre manière de voir les choses, ses propres besoins et différentes méthodes d'évaluation des performances. En l'absence de réelles possibilités d'adaptation d'un outil à leurs besoins spécifiques, ou parce qu'une telle adaptation est une tâche trop coûteuse, les chercheurs produisent régulièrement de nouveaux outils.

Ce chapitre présente les capacités que nous souhaitons trouver dans un environnement de visualisation pour l'évaluation de performance. Cette liste de capacités consiste en quelque sorte en un cahier des charges, ou série d'objectifs à réaliser pour avoir un environnement d'évaluation de performance complet.

Nous avons divisé ces objectifs en catégories représentatives d'une classe de capacités donnée. Ces catégories sont certes arbitraires mais elles englobent les différents points qui définissent les caractéristiques d'un environnement facile d'utilisation, puissant et pérenne. Nous décrivons ci-dessous ce que nous entendons par chacune de ces catégories et expliquons en quoi elles sont nécessaires

dans un environnement d'évaluation de performance et quels avantages elles apportent par leur présence.

1.1 Extensibilité

Nous désignons par *extensibilité* la capacité d'un environnement d'évaluation de performance à supporter l'augmentation de ses possibilités que ce soit simplement pour ajouter des outils à l'environnement ou pour lui permettre de manipuler de nouveaux types de données. Le besoin de possibilités d'extension de l'environnement est motivée par plusieurs observations.

La plus évidente est que l'on constate qu'il est probable que les modèles de programmation utilisés par les développeurs d'applications parallèles vont évoluer et subir des changements. Les modèles actuels ne sont pas encore stabilisés, de nouvelles techniques de programmation font leur apparition et les habitudes de programmation évoluent également. De nouveaux modèles de programmation sont mis au point pour répondre à des besoins spécifiques, que ce soit au niveau du pouvoir d'expression du parallélisme que du support de certaines catégories d'applications. Dans un domaine aussi mouvant il faut pouvoir s'adapter rapidement aux changements et être à même de les supporter efficacement à des fins d'analyse de performance.

Notons également qu'il est désirable de tester de nouvelles techniques d'analyse et de représentation au sein d'un environnement d'évaluation de performance. Des exemples de telles techniques sont l'ajout du son dans l'environnement (le son fournissant une dimension d'information supplémentaire à laquelle nous sommes particulièrement sensibles, et ce sans pour autant demander une attention particulière), l'intégration d'outils extérieurs (applications d'analyse de données et de présentation d'informations) ou encore le prototypage et la validation de nouveaux outils de diagnostic de performance et d'aide à l'évaluation.

Un dernier besoin facile à prévoir est celui de la latitude d'action laissée à l'utilisateur dans le cadre de ses évaluations de performance. Même si un environnement est très complet tant au niveau des visualisations proposées qu'en ce qui concerne ses possibilités de manipulation de données et d'indices de performance, il est à prévoir que l'utilisateur aura toujours des besoins spécifiques. Nous souhaitons par exemple que ce dernier soit à même de réaliser une évaluation des

performances spécifiques à son application, c'est-à-dire en exploitant sa connaissance de l'application et du comportement qu'il en attend. Une telle évaluation peut nécessiter la mise en place de schémas d'analyses particuliers, ou l'ajout d'informations pour les visualisations ; ou en tout autre chose. Le simple fait de ne pas connaître *a priori* les besoins d'un tel schéma d'évaluation est une raison pour justifier l'extensibilité du système.

On trouve dans les applications extensibles différents moyens d'augmenter les capacités, allant de l'interface de programmation de l'application à la mise à disposition d'un langage de programmation simple permettant de la contrôler et de créer de nouvelles fonctions. On remarque d'autre part que les utilisateurs, suivant qu'ils sont ce que l'on nomme couramment « naïfs » ou « experts », ont des besoins d'extension très différents.

L'utilisateur naïf a des besoins simples et ne désire absolument pas s'investir dans l'apprentissage d'une interface d'extension de l'environnement un peu compliquée : il veut simplement accroître ses possiblités de façon simple, par exemple pour ajouter un filtrage ou un calcul d'indice de performance ; pour lui la facilité d'extension est sans doute plus importante que la puissance des mécanismes proposés. L'utilisateur expert cherche par contre à tirer le maximum des possibilités d'extension qui lui seront fournies, que ce soit pour associer ses indices de performances à de nouvelles visualisations ou pour ajouter à l'environnement les capacités spécifiques dont il a besoin. Nous considérons qu'un environnement n'est réellement extensible que s'il supporte correctement les besoins de ces deux catégories d'utilisateurs.

En résumé nous attendons d'un environnement d'évaluation de performance pour le parallélisme qu'il soit capable de supporter plusieurs modèles de programmation (et l'ajout de nouveaux modèles), qu'il puisse intégrer de nouveaux outils et qu'il soit à même de fournir des capacités d'extension à la mesure des attentes des différentes catégories d'utilisateurs potentielles de l'application.

1.2 Généricité

Dans le cadre qui nous intéresse ici la *généricité* est la capacité des composantes d'un environnement à être réutilisées pour diverses opérations ou dans différents contextes. L'intérêt de disposer d'outils génériques est multiple. En pre-

mier lieu les possibilités d'exploitation de ces outils sont augmentées puisqu'ils ne sont pas limités à une utilisation dans un cadre unique. Ensuite l'apprentissage de l'utilisation d'un outil générique n'est fait qu'une fois par l'utilisateur, et si cet outil est ensuite réutilisé cette prise en main n'est plus nécessaire. Enfin, le développement d'outils génériques est bien plus rentable que celui d'outils spécifiques puisque leur coût de création est « amorti » du fait de leurs possibilités de réutilisation.

De manière pratique on obtient des composants génériques en les réalisant de manière à ce qu'ils puissent traiter des données indépendemment de la sémantique qui leur est associée dans le cadre de l'évaluation de performance. En laissant à l'utilisateur d'un outil la tâche d'interprétation de la signification des données qui sont traitées et produites par ce dernier, il devient facile de réutiliser cet outil du moment que les données qui lui sont fournies ont des types corrects et que la manipulation de ces données par l'outil a un sens.

La presque totalité des composantes d'un environnement d'évaluation peut être conçue de manière à être générique. Les outils d'analyse de données, par exemple, sont génériques par nature. Pour effectuer un calcul de moyenne ou d'histogramme il n'est pas nécessaire de connaître la sémantique des valeurs servant aux calculs : celle-ci est connue de la personne qui fournit les données pour l'analyse et utilise les résultats des calcul effectués. Si les calculs effectués par un outil sur des données particulières ne signifient rien, il est de la responsabilité de l'utilisateur de ne pas fournir de telles données, mais l'outil d'analyse n'a pas à se préoccuper de tels cas. Il en va de même pour les représentations, ou visualisations. Des visualisations comme la représentation d'histogrammes ou de courbes, de matrices de données, de graphes, pour ne citer que celles-ci, sont totalement réutilisables : l'outil de visualisation a pour seule tâche la présentation des données qui lui sont fournies, et c'est là aussi l'utilisateur qui interprète les représentations obtenues.

Le nombre d'outils génériques disponibles par rapport au nombre total d'outils de l'environnement doit être le plus grand possible afin de multiplier les possibilités de réutilisation et favoriser le développement de nouveaux schémas d'analyse de performance à partir de ces outils. Il est souhaitable que l'environnement encourage l'utilisateur à user de cette généricité et à tester lui-même de nouvelles associations d'outils dans le cadre de son évaluation de performance.

Les outils génériques permettent un important gain de temps d'apprentissage

pour l'utilisateur mais surtout ils supportent d'être utilisés dans des situations diverses et éventuellement avec des outils qui n'existaient pas lors de leur conception, et avec lesquels ils sont néanmoins compatibles parce que réutilisables.

1.3 Personnalisation

Nous parlons de *personnalisation* pour désigner les actions de l'utilisateur sur certains paramètres de l'environnement et des outils qui le composent. Si un environnement est personnalisable, il peut plus facilement être adapté aux besoins de l'utilisateur et à sa façon de percevoir ce qui lui est présenté (données et visualisations), laquelle peut être très différente de celle de la personne qui a défini l'apparence par défaut des visualisations.

Le niveau le plus simple de personnalisation que l'on peut trouver dans un environnement est celui des préférences utilisateurs. On appelle ainsi un jeu de paramètres (par exemple l'échelle de temps qui est utilisée, ou encore les outils qui sont mis en œuvre par défaut) qui sont modifiables par l'utilisateur et changent l'apparence ou le fonctionnement de base de l'environnement. En agissant sur ces paramètres il est possible de personnaliser l'environnement de manière globale.

Un second niveau consiste à permettre le changement de paramètres pour les différents outils disponibles dans l'environnement. Si un même outil peut être présent sous forme de plusieurs instances, chacune d'entre elle pourra être paramétrée individuellement. Des exemples d'une telle personnalisation pour une visualisation sont le choix des couleurs et des formes utilisés pour la représentation des informations.

Un mécanisme de personnalisation intelligent doit pouvoir assurer une certaine cohérence sauf avis contraire de l'utilisateur. Prenons par exemple le cas de différentes visualisations qui toutes sont susceptibles de présenter les mêmes données, sous différentes formes. Il est souhaitable de pouvoir partager des paramètres entre ces diverses visualisations de manière par exemple à ce que si l'on associe une certaine couleur à un type de données dans une des visualisations, ce même type de données soit représenté dans la même couleur dans les autres visualisations. Il est alors possible de modifier les paramètres de l'environnement tout en gardant des repères visuels importants : l'association qui faite entre la couleur et le type de données étant valable dans toutes les visualisations, le lien entre

celles-ci est plus facile à faire et l'observation des informations sous plusieurs formes s'en trouve facilitée.

Une façon qui nous semble intéressante pour pouvoir répercuter ainsi l'effet de la personnalisation d'un paramètre sur plusieurs outils est d'utiliser des gestionnaires de personnalisation partagés entre ces outils. Un tel gestionnaire s'occupe uniquement de maintenir un répertoire des paramètres modifiables et de leurs valeurs, autorise l'utilisateur à les changer et valide les modifications qui leurs sont apportées. Les outils qui veulent utiliser ces paramètres font part de leur intention au gestionnaire, et en deviennent en quelque sorte clients. En retour, le gestionnaire de paramètres signale tout changement des valeurs des paramètres à ses clients, qui peuvent alors agir de manière à rendre ce changement perceptible à leur niveau.

Cette solution présente plusieurs intérêts. Pour l'utilisateur, cela signifie bien évidemment qu'il n'est pas nécessaire de changer de multiples fois un même paramètre pour qu'il soit pris en compte par plusieurs outils. Normalement cela signifie aussi que seuls les paramètres qui l'intéressent lui sont présentés, puisqu'il peut y avoir un nombre quelconque de gestionnaires pour représenter l'ensemble des paramètres de l'environnement et de ses outils. Pour les outils, cela représente un gain d'investissement puisque la gestion d'un jeu de paramètres donné n'est à réaliser qu'une fois. Enfin, si les paramètres sont standardisés, ce sytème contribue à donner l'assurance que le traitement d'un paramètre donné par les différents outils sera cohérent.

Il ne faut en aucun cas sous-estimer l'intérêt de la personnalisation. Bien employé, un tel mécanisme permet d'apporter une grande souplesse dans l'utilisation de l'environnement. Il aide également l'utilisateur dans sa tâche d'analyse en l'autorisant à travailler comme il le souhaite, dans un environnement présentant les informations comme il l'entend, plutôt que de le forcer à subir un cadre fixe qui n'est peut-être pas le mieux adapté à sa manière de percevoir les choses.

1.4 Interactivité

Nous dirons d'un environnement d'évaluation de performance qu'il fait preuve d'*interactivité* s'il supporte activement une interaction entre l'utilisateur et les outils qu'il offre. Il ne s'agit pas de l'interaction classique entre une interface

graphique et un utilisateur mais plutôt de la manière dont l'utilisateur peut agir sur les informations qui lui sont présentées et de la réaction à ses actions de la part des outils présentant lesdites informations.

Considérons les visualisations présentes dans les environnements d'évaluation de performances. Elles donnent des informations sous forme synthétique, et cachent un certain nombre de données, que ce soient celles ayant servi à la synthèse ou d'autres n'ayant pas un rapport direct avec le propos d'une visualisation particulière. Si l'on désire accéder non plus simplement à la représentation synthétique mais à l'ensemble des données sous-jacentes, que ce soit pour la totalité des informations présentées ou simplement pour quelques-une d'entre elles, on peut envisager d'utiliser suffisamment de visualisations pour être certain que toutes les données sont représentées dans l'une ou l'autre d'entre elles. L'inconvénient est alors que les données que l'on souhaite observer se retrouvent dispersées dans plusieurs visualisations ; si l'on souhaite en fait avoir ces données pour une seule information, il est également dommage d'être obligé de les demander pour l'ensemble des informations et d'extraire ensuite ce que l'on dédire de la masse de données obtenues. Si une visualisation supporte ce que l'on nomme couramment l'inspection, au contraire, il est possible de choisir une des informations représentées et d'obtenir l'ensemble des données s'y rapportant. Ces données sont alors regroupées, facilitant leur appréhension, et de plus on n'obtient pas plus d'information que ce que l'on désire réellement. L'inspection est une interaction entre l'utilisateur et la visualisation : le premier choisit l'information qu'il souhaite inspecter et la visualisation lui fournit en retour les données associées, et seulement celles-ci.

Il existe beaucoup d'interactions utiles que nous souhaitons trouver dans un environnement d'évaluation de performance. Outre l'inspection précédemment citée, nous pensons par exemple à la manipulation de représentations tridimensionnelles. Ces représentations montrent les informations sous un point de vue particulier, en plaçant l'observateur à un point donné dans l'espace de la représentation. En déplaçant l'observateur il est possible de changer de point de vue, de se rapprocher ou de s'éloigner de la scène tridimensionnelle présentée (et même de se retrouver à l'intérieur de la scène) ; en « tournant » autour de la scène on peut mettre en évidence différentes informations, ou mieux observer leurs relations. D'autres interactions utiles concernent le changement du placement ou de la visibilité des informations dans une visualisation.

Nous souhaitons qu'un environnement interactif permette l'utilisation de la

manipulation directe des informations. La manipulation directe consiste à appliquer un « outil » sur des informations. La visualisation contenant les informations auxquelles on applique un outil réagit en fonction du type de l'outil et des informations concernée et conduit l'interaction avec l'utilisateur de manière adaptée. Il est évident que des visualisations distinctes vont réagir différemment à un outil d'inspection, par exemple en ne fournissant pas les informations de la même manière. Mais ce qui est important est que l'effet d'un outil donné, comme l'obtention d'informations supplémentaires avec un outil d'inspection, soit le même pour toutes les visualisations supportant l'usage de cet outil. Cela renforce encore le principe de cohérence dont nous avons déjà parlé, et facilite l'usage de l'environnement et l'efficacité avec laquelle l'utilisateur peut obtenir les informations exactes dont il a besoin.

En résumé il est essentiel qu'un environnement d'évaluation de performance supporte une interaction intelligente avec l'utilisateur, au moins pour l'inspection des informations qui lui sont présentées. Nous souhaitons que cette interaction se fasse par manipulation directe car cela est une façon de faire intuitive et aisée à apprendre. Les outils de manipulation directes doivent être standardisés, c'est-à-dire que si deux visualisations supportent l'emploi d'un même outil elles ne doivent pas le faire de manières différentes : même si elles peuvent étendre l'effet de la manipulation, elles ne peuvent pas le réduire.

1.5 Puissance

La *puissance* d'un environnement est quelque chose d'assez difficile à définir. De manière globale, elle exprime la facilité avec laquelle l'environnement peut manipuler les données nécessaires à l'évaluation de performance, c'est-à-dire une trace et les informations qui en dérivent.

L'extensibilité, la généricité, de même que les mécanismes de personnalisation et d'interaction offerts, participent tous à la puissance d'un environnement. Ce sont même souvent les critères essentiels qui sont utilisés pour dire d'un outil qu'il est plus ou moins puissant. Un autre critère utilisé en conjonction avec les précédents est celui de la capacité de l'environnement à supporter des problèmes de grandes tailles sans détérioration de performances et tout en restant utilisable (par exemple, la capacité à fournir des informations exploitables aussi bien pour des applications comportant quelques dizaines ou quelques milliers de tâches).

Nous souhaitons ajouter également la notion de support des besoins de l'utilisateur, dont nous donnons ici deux exemples significatifs.

Le but d'une d'une session d'évaluation de performance est de contrôler la correction des performances d'une application. Si un problème de performance est mis en évidence la question qui se pose alors est de connaître ses causes et son origine afin de de pouvoir le corriger. Pour ce faire il est nécessaire de pouvoir revenir à l'instant correspondant à l'origine du problème. S'il faut pour cela de rejouer la trace depuis son début, alors la recherche de cet instant peut se révéler très longue et pénible. Nous souhaitons donc qu'il soit possible de se déplacer à un instant arbitraire de l'exécution, que ce soit en amont ou en aval de l'instant présent (par exemple celui où le problème a été découvert). Une telle possibilité facilite non seulement la recherche des prémisses d'un problème mais également une exploration non-linéaire de l'exécution d'une application.

Puisque la tâche de l'utilisateur la plus courante est la recherche d'un problème de performance, l'environnement doit être capable de l'assister dans cette recherche. On remarque en effet qu'il est possible de définir la plupart des problèmes de performances d'après des critères simples. Par exemple, le fait qu'une barrière de synchronisation prenne un temps important indique un problème de performance. De même, dans une application à envoi de messages écrite de manière à ce que les communications soient recouvertes par les calculs, les temps d'attentes lors de la réception d'un message devraient être presque nuls (les messages devraient être arrivés lorsqu'une tâche veut les récupérer) ; dans le cas contraire, on se trouve en présence d'un problème de performance, dû par exemple à un mauvais choix de la taille des messages à recouvrir.

Nous voulons trouver dans l'environnement des outils d'automatisation de la recherche des problèmes de performances. Le rôle de ces outils est de « surveiller» l'évolution de l'exécution de l'application à travers un certain nombre d'indices de performances et autres données issues de la trace, et d'avertir l'utilisateur dès qu'une anomalie est détectée. On peut également envisager que ces outils « intelligents » aient un certain contrôle sur l'environnement. Ce contrôle peut leur permettre, par exemple, d'arrêter le défilement de la trace — et donc son traitement — dès la détection du problème de performance qu'ils sont chargés de guetter : l'utilisateur peut alors observer les informations présentes pour confirmer la présence d'une anomalie avant de rechercher ses causes si besoin est. Des outils évoluent permettraient même de fournir une tentative d'explication du problème, et d'aider à sa localisation dans l'application évaluée. Le cycle « recherche et cor-

rection d'un problème de performance, exécution tracée, réévaluation » qui est celui de l'optimisation des performances d'une application s'en trouverait grandement facilité.

Pour récapituler nous dirons que la puissance d'un environnement se mesure par ses capacités d'extension, ses mécanismes de personnalisation et d'interaction et enfin la manière dont il peut aider l'utilisateur à mener sa tâche d'évaluation de performance.

1.6 Convivialité

Synonyme de simplicité ou de facilité d'emploi, la *convivialité* d'un environnement caractérise sa souplesse, la qualité du caractère informatif de ses outils et la facilité avec laquelle l'utilisateur peut réaliser les tâches qu'il désire. C'est sans doute aussi la notion la plus subjective de celles présentées dans ce chapitre.

Il faut bien se rendre compte que les capacités introduites dans les sections précédentes ne servent à rien si elle ne sont pas aisément exploitables. Le temps d'apprentissage de l'utilisation de l'environnement doit être faible et les actions de l'utilisateur doivent toujours rester simple et intuitives. Si ce principe n'est pas respecté, le risque est que l'utilisateur ne se serve jamais des outils dont le maniement est trop complexe, même si ces outils pourraient l'aider dans sa tâche. Il se peut même qu'il abandonne l'emploi de l'environnement, quelles que soient ses qualités potentielles, pour se tourner vers des outils plus simples d'utilisation avec lesquels il lui est plus facile de travailler.

La cohérence des moyens de manipulation de l'environnement et des informations qu'il présente, que nous avons déjà évoquée précédemment, est également quelque chose de très important. Il faut absolument qu'une même opération soit toujours conduite de la même manière, quel que soit le contexte présent ou la visualisation qui supporte l'opération.

L'interface de manipulation d'un environnement ne doit jamais être un frein à son utilisation. Son rôle est de simplifier les choses et de faciliter le travail de l'utilisateur. L'environnement doit inciter l'utilisateur à exploiter au maximum toutes ses capacités en les lui rendant facilement accessibles.

1.7 Conclusion

Nous avons présenté dans ce chapitre les points qui nous semblent importants pour qu'un environnement de visualisation pour l'évaluation de performance de systèmes parallèles soit puissant et facile d'emploi, pour qu'il soit à même de supporter les besoins actuels et à venir des utilisateurs de systèmes parallèles en matière d'évaluation de performance de leurs applications. Ils représentent ce que nous souhaitons trouver dans l'environnement d'évaluation de performance que nous utiliserons.

Les objectifs donnés dans ce chapitre nous ont servi de référence pour présenter l'état de l'art en matière d'environnements pour l'évaluation de performance, les capacités des environnements existants étant comparées à celles d'un « environnement idéal » réalisant les fonctions que nous avons évoquées ici.

2

État de l'art

LA VISUALISATION pour l'évaluation de performance des systèmes parallèles, et particulièrement la visualisation dirigée par les traces, a été la première technique permettant aux programmeurs d'applications d'avoir une vision globale du comportement de leurs réalisations. Les outils de visualisations se sont alors développés, presque à raison d'un outil par laboratoire utilisant le parallélisme et ayant des besoins en évaluation de performance. Aujourd'hui encore on continue à développer des environnements de visualisation pour pouvoir traiter de nouveaux problèmes ou pour tester un nouvel outil d'analyse.

Nous commençons par donner les principes de base communs à tous les environnements de visualisation pour l'évaluation de performance. Nous indiquons également, pour les points où cela est possible, les différents choix qui ont été effectués pour différents environnements.

Cette introduction est suivie d'une présentation des principaux environnements disponibles. Pour chacun des ces environnements nous donnons, outre leur principe, leurs points forts et leurs points faibles. Nous indiquons également pour chacun d'eux l'intérêt qu'il présente en fonction des différents objectifs que nous avons présentés au chapitre 1 page 11.

Enfin nous terminons cet état de l'art par une présentation synthétique de l'adéquation entre les capacités que nous souhaitons trouver dans un environnement de visualisation pour l'évaluation de performance et les différents outils pré-

sentés. Cette synthèse fournit une vue globale de l'intérêt de chacun des outils présentés par rapport à notre recherche de l'« environnement idéal » satisfaisant tous nos objectifs initiaux.

2.1 Principes

Les environnements d'évaluation de performance fonctionnent tous suivant un principe assez simple : à partir de données sur l'exécution d'une application parallèle, l'environnement produit un certain nombre d'informations (indices de performance, statistiques, suite des états de l'application) qui sont passées à des visualisations permettant de les présenter graphiquement.

Les outils auxquels nous nous intéressons sont tous basés sur l'utilisation d'une *trace d'exécution* décrivant les actions effectuées par l'application. La plupart d'entre eux produisent entre autres informations un *graphe d'état* de l'application parallèle ; ce graphe est obtenu par simulation des actions de l'application.

On peut ainsi découper globalement un environnement de visualisation pour l'évaluation de performances en plusieurs parties : la lecture de traces qui extrait les actions de l'application d'une trace d'exécution, la simulation — optionnelle — qui reconstitue l'état de l'application après chacune de ses actions, l'analyse de données qui produit des indices de performances et des statistiques et enfin la visualisation proprement dite qui présente toutes les informations utiles à l'évaluation de performance.

2.1.1 Traces d'exécution

Même si la production de la trace d'exécution n'est pas du ressort de l'environnement d'évaluation de performances elle conditionne grandement le nombre et la qualité des informations que celui-ci sera à même de produire. Il est donc important, lorsqu'on analyse les performances d'une application, de connaître la manière dont les traces ont été produites afin de pouvoir comprendre la raison de certains résultats et pondérer son évaluation en fonction de ces connaissances. Nous présentons donc le mécanisme de la prise de traces avant d'évoquer le problème de la lecture de traces par les outils d'analyse.

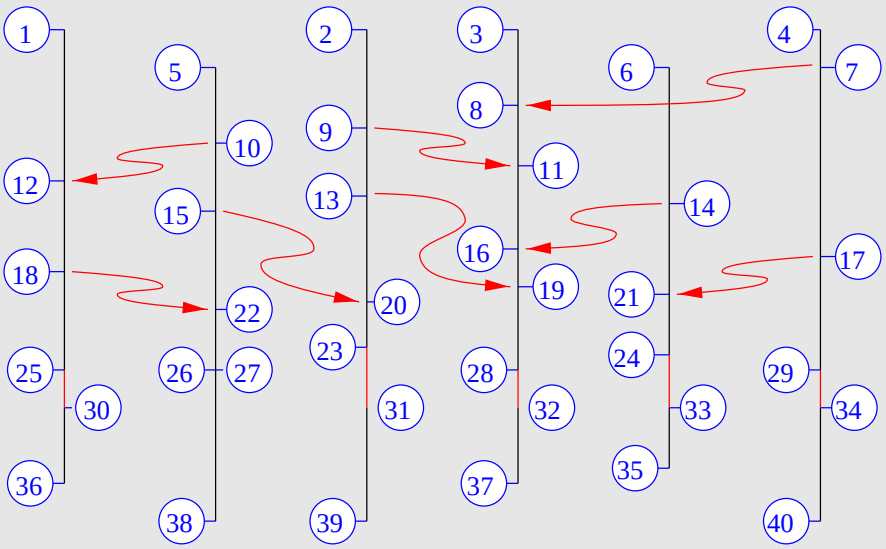


FIG. 2.1. – *Les événements sont générés aux instants des actions faisant changer l'application d'état. Dans cet exemple, où les barres verticales représentant les actions d'une tâche au cours du temps, nous avons marqué ces instants en donnant un numéro à chaque événement, ces numéros représentant l'ordre temporel entre les événements. Pour des raisons de lisibilité, seuls deux événements sont marqués pour chaque communication.*

Prise de traces

Une trace d'exécution est une suite d'*événements* décrivant les actions réalisées au cours de l'exécution d'une application. De tels événements correspondent par exemple aux instants où une tâche commence à s'exécuter ou se termine, ainsi qu'à l'appel des primitives de programmation parallèle du système sur lequel l'application fonctionne. Dans le cas d'un système à envoi de messages, par exemple, des événements correspondent aux communications entre tâches ou encore à l'appel de fonctions de synchronisation des différentes parties de l'application (figures 2.1 et 2.2 page suivante).

Les traces sont prises avec un *traceur*, qui est un outil spécialisé dans l'observation d'une application parallèle : chaque fois que celle-ci effectue une action intéressante (c'est-à-dire entraînant un changement d'état) le traceur enregistre un

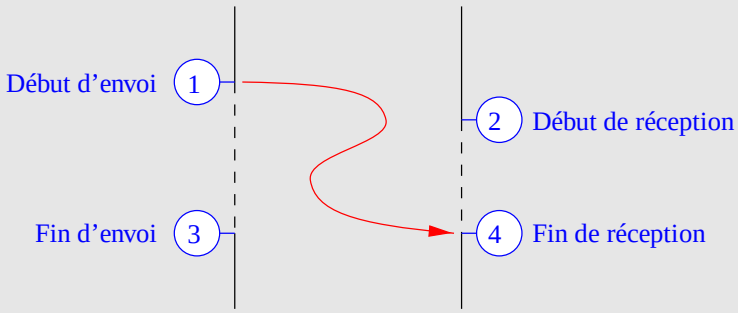


FIG. 2.2. – *On peut voir sur cette figure que suivant le modèle de programmation utilisé (ici CSP) il peut y avoir bien plus de deux événements pour une seule communication point à point.*

événement contenant des informations telles que la date à laquelle l'action est faite, l'identification de l'entité concernée et des données décrivant l'action elle-même (son type et des informations dépendant de ce dernier). Les événements sont la plupart du temps stockés pour une analyse *post-mortem* de la trace, celle-ci n'étant pas exploitable en temps réel.

Les traces prises pour l'évaluation de performance ont une caractéristique particulière qui est celle de leur volume : les événements correspondant aux actions contiennent toutes les informations disponibles sur celles-ci, comme par exemple l'ensemble des paramètres d'une fonction d'envoi de message. Cela permet ensuite aux outils d'évaluation de performance de calculer n'importe quel indice de performance puisqu'il a la totalité des informations que l'on peut obtenir sur une exécution. En revanche la prise et le stockage des traces pose des problèmes à cause du volume d'informations généré par le traceur.

Nous distinguons trois types de traceurs : les *traceurs matériels*, les *traceurs hybrides* et les *traceurs logiciels*. Tous fournissent des informations sur l'exécution d'une application parallèle, mais pas au même coût ou avec la même finesse.

Les traceurs matériels observent l'application et le système sous-jacent au niveau du matériel, et acheminent les événements constituant la trace avec un matériel qui leur est propre. Le plus grand avantage des traceurs de ce type est qu'ils sont invisibles pour l'application, et ne perturbent pas (ou très peu) son exécution. Ils sont également capables de tracer des événements au niveau de la machine

elle-même, autorisant l'obtention d'informations de très bas niveau, par exemple sur les actions de l'ordonnanceur d'un système multi-programmé ou encore sur la gestion des caches et des tampons systèmes. Malheureusement ces traceurs coûtent très cher et sont très difficiles à mettre au point. Ils sont de plus très étroitement liés avec l'architecture matérielle de la machine parallèle sur laquelle les traces sont prises et risquent de devenir inutilisables dès que celle-ci change un tant soit peu. Enfin si l'on souhaite comparer les performances d'une application exécutée sur deux machines différentes il faut construire entièrement deux traceurs dédiés, et on ne pourra exploiter dans tous les cas que les informations qui peuvent être obtenues sur les deux machines, perdant ainsi une partie de l'avantage de la finesse de trace atteignable avec un traceur matériel. Ce sont les raisons pour lesquelles on ne trouve plus de traceurs de ce type actuellement.

Un moyen de diminuer le coût des traceurs matériels tout en leur conférant une certaine portabilité est de les changer en traceurs dits hybrides (KLAR 1992). Les fonctions assurées par ces traceurs sont réparties entre une partie matérielle et une partie logicielle. La plupart du temps l'enregistrement des données propres aux événements (paramètres des fonctions tracées) est logicielle et la datation ainsi que le transport des événements sont matériels. L'avantage de la prise de trace logicielle est que, pour un système de programmation parallèle donné (langage ou bibliothèque de programmation), on ne réalise qu'un seul traceur qui est portable sur plusieurs machines. Ses inconvénients sont qu'une prise de trace logicielle est coûteuse et peut perturber l'exécution de l'application, et aussi que les traces sont prises à un niveau plus haut et que les informations disponibles sont moins précises. La datation et le transport matériels, de leur côté, assurent d'une part qu'il est possible de produire si besoin est une date dans un référentiel de temps global, ce qui permet de classer les événements dans l'ordre exact où ils ont été produits, et d'autre part de ne pas perturber le réseau de communication de la machine sur laquelle l'application est tracée en y faisant circuler des événements, ceux-ci utilisant un réseau de transport indépendant. Ces traceurs sont néanmoins encore très coûteux et leur partie matérielle souffre du même problème de portabilité que les traceurs entièrement matériels, à savoir qu'il faut l'adapter à chaque machine, cette adaptation n'étant pas toujours possible en l'absence d'une connaissance très poussée de l'architecture de la machine utilisée.

La solution la plus utilisée actuellement est celle des traceurs logiciels (GEIST *et al.* 1991, VAN RIEK 1991, VAN RIEK et TOURANCHEAU 1991, TOURANCHEAU et VAN RIEK 1992, ARROUYE 1992, MAILLET 1995). De tels traceurs effectuent toutes leurs tâches, c'est-à-dire la prise de mesures et leur envoi, de

manière logicielle, par *instrumentation* du code de l'application ou de la bibliothèque de programmation qu'elle utilise. Leur intérêt est leur grande portabilité puisqu'ils ne dépendent que des outils de programmation parallèles (langages ou bibliothèques de programmation) et absolument pas de la machine sur laquelle s'exécutent l'application. En revanche, ils posent un certain nombre de problèmes. Le premier d'entre eux est la perturbation de l'exécution liée à la prise des traces : chaque fois qu'un événement est générée, le traceur « vole » du temps à l'application et décale les événements suivants dans le temps. Cela peut finir par donner des traces incohérentes dans lesquelles l'ordre des événements indiqués n'est pas l'ordre des événements réels, ou représente une impossibilité (par exemple la réception d'un message avant son envoi dans un système à passage de messages).

À la perturbation liée à la production de l'événement lui-même peut s'ajouter une plus grande perturbation due au transport des événements pour leur enregistrement sur disque. Même si les événements sont gardés en mémoire autant que possible, il arrive un moment où l'espace nécessaire à ce stockage local est épuisé : il faut alors envoyer tous les événements enregistrés vers un site de stockage, ce qui produit une perturbation immense du fait du partage du réseau de communication avec l'application tracée et du temps de blocage de la tâche déclenchant ce transport. Outre la modification des dates des événements la perturbation due à la prise de trace peut également modifier l'ordre des événements, lequel peut varier à cause de l'indéterminisme des programmes parallèles. Enfin le dernier problème des traceurs logiciels pour les performances est que la plupart des machines actuelles ne disposent pas d'une horloge globale permettant de dater sans ambiguïté les événements produits sur des processeurs différents. Si les traces ne sont pas construites avec précaution, par exemple en synchronisant les horloges de la machine avant l'exécution de l'application, ou en reconstituant un temps global après la prise de traces proprement dite (JÉZÉQUEL 1990, MAILLET et TRON 1995), elles peuvent se révéler inutilisables.

Lecture de traces

La lecture de traces est l'opération qui permet d'extraire d'une trace la succession des événements représentant les actions effectuées par une application au cours de son exécution. Ces événements sont fournis à l'environnement d'évaluation de performance dans un certain ordre, celui-ci étant normalement l'ordre des dates croissantes des événements puisqu'il permet de reconstituer la succession

des états de l'application par simulation.

Le problème principal des traces d'exécution actuellement disponibles est la multiplicité des formats de traces : on trouve presque un format de trace par environnement de programmation ou d'évaluation de performance (outre les formats des lecteurs évoqués précédemment citons notamment BEMMERL *et al.* 1990, BEGUELIN, DONGARRA, GEIST, MANCHEK et SUNDERAM 1991, KILPATRICK et SCHWAN 1991, VAN RIEK 1991, ERLANGEN-NÜRNBERG UNIVERSITÄT 1992a, HERRARTE et LUSK 1992) et certains environnements d'évaluation de performance utilisent même des formats différents pour un même système suivant la machine utilisée. La situation est d'autant plus compliquée que chaque format de trace fournit des informations dont non seulement la syntaxe mais aussi la sémantique diffère des autres formats pour la simple raison que différents environnements supportent différents paradigmes de programmation pour lesquels on n'a pas besoin des mêmes informations.

Un format de trace a cependant joué un rôle particulier. C'est celui généré par la bibliothèque de communication PICL (*Portable Instrumented Communication Library,* GEIST *et al.* 1991, WORLEY 1992) qui a longtemps été le format le plus utilisé, principalement à cause de son exploitation par l'outil de visualisation ParaGraph (cf. § 2.2.1 page 33). Beaucoup de traceurs actuels sont accompagnés d'outils de conversion de leur format de traces en format PICL dans le seul but de permettre la relecture des traces par ParaGraph à des fins de visualisation.

Des essais de standardisation des formats de traces ont néanmoins été réalisés en proposant des formats auto-descriptifs (HON 1990, MALONY et NICHOLS 1991, ERLANGEN-NÜRNBERG UNIVERSITÄT 1992a, REED *et al.* 1992) qui servent à produire des traces contenant d'une part les informations de trace habituelles et d'autre part une description de haut-niveau de la syntaxe des événements contenus dans la trace. L'intérêt de cette approche est qu'il est possible de manipuler ces traces avec des outils d'analyse génériques (ERLANGEN-NÜRNBERG UNIVERSITÄT 1992b, ERLANGEN-NÜRNBERG UNIVERSITÄT 1992c) pour effectuer des opérations simples (comme par exemple des statistiques sur un champ correspondant à la taille des messages échangés par les tâches). Ils ne résolvent par contre pas le problème du traitement des traces pour reconstituer la suite des états d'une application, cette reconstitution nécessitant une connaissance poussée du modèle de programmation utilisé pour l'application.

Malgré l'existence de ces formats génériques, on utilise toujours une multi-

tude de formats de traces différents qui doivent être traités spécifiquement par les environnements d'évaluation de performance. Il est parfois possible de convertir une trace d'un format vers un autre, mais à condition que les modèles de programmation correspondent, sous peine de produire des traces non interprétables ou faussées.

2.1.2 Simulation

La simulation est le procédé par lequel un outil d'évaluation de performance reconstruit la succession des états de l'application au cours de son exécution. Cette reconstruction n'est pas obligatoire mais c'est elle qui permet de connaître certains indices de performance importants tels que le taux d'utilisation des processeurs, le temps pendant lequel une application est restée bloquée en attente de réception de données, etc. Aussi la plupart des environnements d'évaluation de performance actuels utilisent-ils un simulateur pour fournir ces indices.

Ces simulateurs dirigés par les traces sont dépendants de la sémantique des événements traités. Il faut donc, pour traiter plusieurs modèles de programmation, disposer d'un simulateur adapté à chacun de ces modèles.

2.1.3 Analyse de données

Les données brutes fournies par la trace ne sont pas toujours exploitables telles quelles. Le propos de l'analyse de données et de traiter ces données pour produire des indices de performance qui seront ensuite représentés dans les visualisations de l'environnement.

Ces indices de performance sont la plupart du temps des indicateurs simples comme des moyennes de volume de données échangés entre les tâches d'une application, des comptages de messages en attente, etc. Suivant les informations souhaitées, ces indicateurs sont obtenues en manipulant les données des événements eux-mêmes ou les indications fournies par un simulateur.

Les outils d'analyse de données sont souvent simples et les informations qu'ils fournissent ne sont pas difficiles à produire en elles-mêmes. La partie délicate de cette tâche réside dans le choix des indicateurs de performance qui seront don-

nés aux visualisations (MILLER 1993). De la pertinence de ce choix dépendent la qualité de la vision des performances d'une application et la facilité avec laquelle les indicateurs de son comportement peuvent être utilisés pour localiser des problèmes de performance.

2.1.4 Visualisation

La visualisation est la partie la plus utilisée des environnements d'évaluation de performance auxquels nous nous intéressons. C'est à travers elle que l'utilisateur voit le comportement de son application et peut décider si celle-ci a des problèmes ou non. En l'absence d'aides automatisées à la recherche de problèmes les possibilités d'évaluation d'une application dépendent principalement de la qualité et de la diversité des visualisations proposées par un environnement d'analyse de performance.

On peut séparer les visualisations en deux catégories. Les visualisations statistiques représentent les indices de performances qui sont produits par les modules d'analyse de données intégrés dans l'environnement. Des exemples courants de visualisations de ce type sont les histogrammes, les jauges, les représentations matricielles ou de contour, etc. Elles donnent une vue synthétique des indicateurs de comportement de l'application. La seconde catégorie est celle des visualisations représentant des informations spécifiques au domaine du parallélisme et au modèle de programmation utilisé pour l'écriture de l'application évaluée. Nous citerons pour exemples les animations de l'état de l'application, qui présentent après chaque événement une vue symbolique instantanée de l'état des différentes entités d'une application et des échanges de données qu'elles effectuent, et les diagrammes espace-temps représentant au cours du temps non seulement l'évolution des états de l'application mais également les interactions entre ses différentes parties.

Si les visualisations statistiques et autres représentations simples sont assez faciles à réaliser, il est très difficile de trouver des visualisations spécifiques à l'évaluation de performance qui apportent réellement un intérêt pour l'analyse du comportement des applications. Le premier problème que l'on rencontre est de déterminer ce qui va être représenté afin que la visualisation soit à la fois informative et simple à comprendre (MILLER 1993). D'autres problèmes concernent la quantité d'informations représentées dans une visualisation donnée. En effet des

visualisations qui sont très correctes pour un petit nombre de données peuvent rapidement devenir totalement inutilisables lorsque ce nombre augmente. Les applications parallèles modernes pouvant comporter beaucoup de tâches il est essentiel aujourd'hui de trouver de nouvelles visualisations pouvant représenter de grand nombres d'objets ; ces nouvelles visualisations doivent également être capables de donner plus d'information tout en restant très lisibles (REED 1991, HARING 1992, HACKSTADT et MALONY 1993).

Un autre problème lié à la visualisation est celui de l'accès aux informations de bas niveau et non pas seulement à celles qui sont représentées, ces dernières étant parfois trop synthétiques. Considèrons une visualisation montrant le graphe de tâches d'une application parallèle (une tâche est un sommet du graphe et les arêtes représentent les voies d'échanges de données) et dans laquelle chaque tâche a une couleur différente suivant son état. Cette visualisation fournit une information de haut niveau, l'état des tâches, mais ne permet pas de savoir autre chose alors que l'on souhaiterait par exemple connaître l'événement qui a fait qu'une tâche est bloquée parce que cette information supplémentaire peut aider à trouver les causes de ce blocage s'il est anormal. Cette possibilité d'accéder à plusieurs niveaux d'information, à interroger une visualisation pour en savoir que ce qu'elle présente de manière synthétique, est un des besoins les plus importants auxquels devront répondre les environnements de visualisation pour l'évaluation de performance des années futures.

Une dernière piste qui n'a pas encore été vraiment exploitée mais qui peut se révéler prometteuse est celle de la sonorisation des traces (FRANCIONI *et al.* 1991a, FRANCIONI *et al.* 1991b). La sonorisation n'est pas vraiment une technique de visualisation puisqu'elle exploite la dimension sonore, mais elle peut se révéler un plus très utile dans l'évaluation des performances d'une application. On sait depuis longtemps que l'ouïe, contrairement à la vue, n'a pas besoin d'être explicitement sollicitée pour réagir à de nombreuses informations ; utiliser le son revient donc réellement à ajouter plusieurs dimensions d'information puisqu'il est exploitable en même temps que la visualisation. L'idée de la sonorisation est d'associer des sons à certaines informations, ces sons formant une « mélodie » représentative du comportement de l'application évaluée. Si la mélodie est brusquement changée, l'utilisateur s'en aperçoit immédiatement et peut ainsi être averti de changements de phases dans son application ou encore d'anomalies : un schéma d'échange de données régulier ayant une mélodie répétitive, toute perturbation de ce schéma entraîne une rupture de la mélodie signalant le problème. L'intérêt principal du son est que nous y sommes très sensibles même s'il est complexe,

c'est-à-dire qu'une anomalie sonore dans une mélodie complexe est bien plus aisément perceptible qu'un changement dans une visualisation complexe. Il peut donc servir de support à la visualisation, celle-ci étant de toute manière obligatoire pour voir ce qui se passe réellement dans l'application évaluée (LEBLANC *et al.* 1990).

2.2 Outils et environnements existants

Il existe aujourd'hui un grand nombre d'outils et d'environnement pour l'évaluation de performance à travers la visualisation. Ils couvrent un grand nombre de domaines et considèrent la visualisation sous différents angles et avec des perspectives variées.

Nous avons choisi de présenter ici quelques environnements et outils de visualisation qui sont représentatifs soit de ce qui est actuellement utilisé soit d'idées qui nous semblent caractéristiques de ce qui sera sans doute conservé dans les outils des prochaines années. Une présentation plus exhaustive ainsi qu'une classification des outils disponibles pourra être trouvé dans les articles de KRAEMER et STASKO (1993) ainsi que ROMAN et COX (1993) mais également dans les références données dans la bibliographie commentée présente à la fin dans ce document.

2.2.1 ParaGraph

C'est l'outil ParaGraph (HEATH 1990, HEATH et ETHERIDGE 1991a, HEATH et ETHERIDGE 1991b, HEATH 1992) qui a popularisé l'utilisation de la visualisation pour l'évaluation des performances d'applications parallèles. Il fut le premier outil disponible publiquement et reste aujourd'hui encore l'un des plus utilisés.

ParaGraph utilise des traces produites par PICL (*Portable Instrumented Communication Library*, GEIST *et al.* 1991), une bibliothèque de programmation parallèle par envoi de messages qui supporte la génération de traces d'exécution. Les applications écrites avec cette bibliothèque sont découpées en phases et s'exécutent à raison d'une tâche par processeur, aussi ParaGraph confont-il les deux. Si aujourd'hui PICL n'est pas la plus utilisée des bibliothèques de programma-

tion parallèle, ParaGraph reste néanmoins un outil très prisé et un grand nombre de traceurs logiciels sont accompagné d'un outil de conversion de leur format de trace vers le format de PICL.

Outre sa disponibilité, le plus grand attrait de ParaGraph est la diversité des visualisations proposées (figure 2.3 page ci-contre). Il en existe actuellement plus de quarante, partagées entre vues instantanées et temporelles, représentations statistiques et spécifiques. ParaGraph les sépare en quatre catégories : les visualisations d'utilisation de la machine donnent des indications sur la manière dont les tâches exploitent les ressources à leur disposition, les visualisations de communication montrent le schéma de communication de l'application (que ce soit à travers un diagramme espace-temps, une animation du graphe d'état de l'application ou des matrices de communication), les visualisations des tâches indiquent l'activité de ces dernières et les visualisations diverses fournissent d'autres informations telles qu'une représentation textuelle de la trace, des statistiques ou encore une vue du chemin critique de l'application. Les visualisations des communications sont particulièrement travaillées et il est possible par exemple de voir une animation des communications sur la plupart des topologies de réseaux d'interconnexion des machines actuelles.

ParaGraph est limité à la visualisation d'applications comportant au plus 128 tâches. Mais ses visualisations supportent très difficilement un aussi grand nombre d'objets et deviennent rapidement illisibles[1]. Il est même assez illusoire de vouloir observer des applications ayant plus de 32 tâches. Une grande partie des visualisations statistiques de ParaGraph est malheureusement « boguée » et donne des résultats ininterprétables.

ParaGraph ne supporte qu'un modèle de programmation, celui de PICL. Pour l'adapter à un autre modèle, il faut réécrire une très grande partie de l'environnement ce qui représente une lourde tâche (GLENDINNING *et al.* 1992). Il est possible d'ajouter *une* nouvelle visualisation à ParaGraph si on le désire. Mais cela est assez compliqué et, ParaGraph disposant d'un éventail de visualisations très large, ne se justifie que pour réaliser une visualisation spécifique à un type d'application, par exemple.

1 . Ce problème est d'autant plus gênant que ParaGraph ne supporte pas qu'une fenêtre en cache une autre, celles-ci n'étant pas rafraichies lorsqu'elles sont découvertes : il faut donc que toutes les visualisations utilisées soient placées côte à côte sur l'écran. L'utilisateur est ainsi souvent obligé de choisir entre la lisibilité des visualisations — impliquant de grandes fenêtres — et le nombre de visualisations différentes utilisées pour son évaluation.

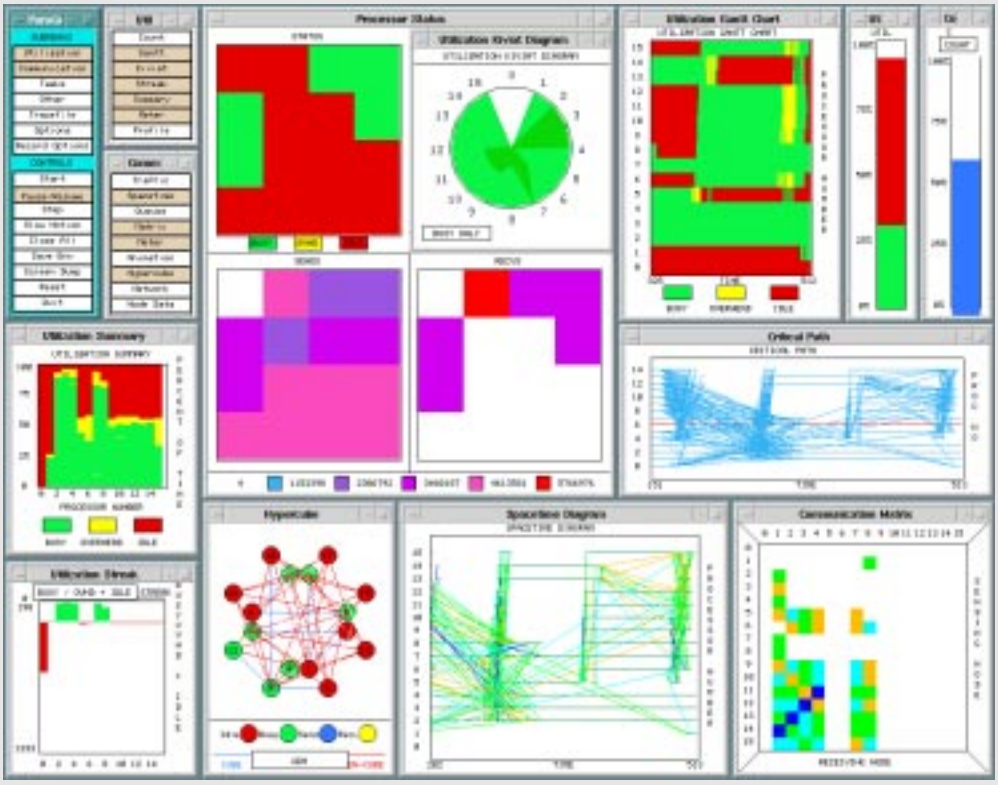


FIG. 2.3. – *Visualisation de l'exécution d'une FFT bi-dimensionnelle sur IBM SP2 avec ParaGraph.*

Les visualisations de ParaGraph sont personnalisables de manière globale et cette configuration ne concerne que des attributs tels que les couleurs employées ou encore le placement des tâches dans les visualisations (ParaGraph pouvant les organiser par défaut par ordre croissant ou par ordre du code de Gray, sauf pour l'animation du graphe d'état de l'application qui propose plusieurs topologies logiques courantes). Elles ne sont absolument pas interactives et il est plutôt difficile de retrouver une information d'une visualisation à l'autre. Qui plus est les visualisations donnent une vue globale du fonctionnement d'une application mais ne permettent pas d'obtenir des indices de performances précis car seules les bornes des intervalles de valeurs sont présentées. Les seules données chiffrées exactes que l'on peut obtenir sont des valeurs globales fournies en fin de traitement de la trace par ParaGraph.

ParaGraph est néanmoins un outil très apprécié par la communauté d'évaluation de performance en parallélisme, que ce soit pour le nombre de ses visualisations ou pour la très grande facilité avec laquelle on commence à l'utiliser, ce qui permet de voir très rapidement le comportement de ses applications. Même si de nombreux outils ont vu le jour après lui, aucun n'a encore atteint sa popularité.

2.2.2 XPVM

XPVM est un outil de visualisation de programmes parallèles écrits avec le système PVM. Son originalité réside dans le fait qu'il récupère chaque événement dès qu'il est généré par l'application et met immédiatement à jour ses visualisations de l'exécution.

Le format de traces utilisé par XPVM est celui que la version 3.3 de PVM fournit si on lui demande de tracer les tâches. Ce format est auto-descriptif et est donc facilement convertible en d'autres formats de traces adaptés au modèle de programmation de PVM. Par contre XPVM n'exploite pas la partie auto-descriptive du format et se contente de lire directement les événements dont il connaît la syntaxe et la sémantique.

XPVM est assez pauvre en visualisations puisqu'il n'en propose que cinq. Les deux plus utiles sont une animation de l'activité de la machine parallèle virtuelle sur laquelle fonctionne l'application et un diagramme espace-temps. La qualité de ces visualisations est néanmoins supérieure à ce qui est offert dans ParaGraph, et surtout le diagramme espace-temps est interactif, c'est-à-dire qu'il est possible de cliquer sur une information affichée et d'obtenir l'événement qui lui est associé. Les visualisations ne sont absolument pas personnalisables et XPVM ne fournit pas de support pour l'extension de l'environnement.

XPVM a surtout un grave inconvénient qui est le rapatriement direct des événements : la perturbation induite par cette action est très importante et de plus la prise de trace prend beaucoup de temps puisqu'elle nécessite beaucoup de communications avec XPVM. C'est ce problème qui fait que XPVM est assez peu utilisé et n'a pas fait l'objet de gros efforts de développement depuis sa version initiale jusqu'en juillet 1995.

Mieux conçu que ParaGraph mais incapable de rivaliser avec lui pour l'instant, XPVM se veut devenir à terme l'environnement simple de référence pour la

FIG. 2.4. – *Visualisation de l'exécution d'une FFT mono-dimensionnelle sur IBM SP2 avec XPVM. La vue du haut représente l'état de la machine parallèle alors que celle du bas est un diagramme espace-temps (états des tâches et communications au cours du temps). On notera la ligne « View Info » en bas, donnant des informations sur la communication sélectionnée sur le diagramme espace-temps, laquelle apparaît sous forme de flèche épaissie.*

visualisation de l'exécution d'applications PVM.

2.2.3 Upshot

Upshot (HERRARTE et LUSK 1992) est un outil de visualisation très simple mais assez intéressant. Il permet de construire rapidement une visualisation per-

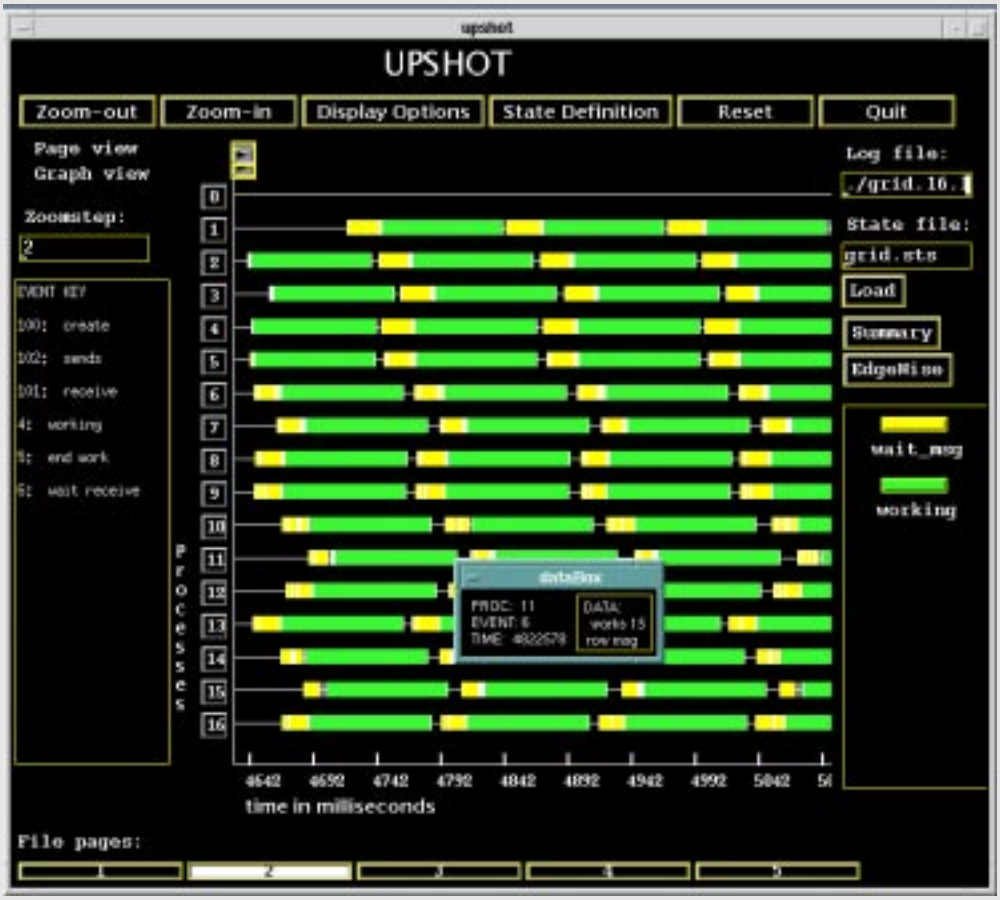


FIG. 2.5. – *Vue de la visualisation d'Upshot. On notera en bas les boutons permettant de se déplacer d'une « page » d'exécution à l'autre ainsi que la petite fenêtre d'inspection d'un événement.*

sonnalisée pour l'application évaluée et supporte le déplacement vers un instant arbitraire de son exécution.

Le format de trace utilisé par Upshot est extrêmement simple et un utilitaire est fourni pour aider à la génération de fichiers dans le format voulu, facilitant la conversion de traces existantes. Upshot n'interprète pas toutes les données de la trace, en particulier il utilise tels quels les types d'événements qui lui sont fournis et conserve les informations utilisateur qui leur sont associées. Un fichier d'états

peut être associé à un fichier de trace. Ce fichier décrit des états en donnant pour un numéro d'état donné un nom utilisé pour se référer à cet état, les types d'événements correspondant au début et à la fin du passage dans cet état, et enfin une couleur pour la représentation de l'état lors de la visualisation.

L'outil ne dispose que d'une visualisation qui est une sorte de diagramme espace-temps (figure 2.5 page précédente). Lors de la lecture de la trace il place les événements sur les lignes correspondant aux tâches, et colorie les états. L'exécution est découpée en « pages » et il est possible de passer d'une page à une autre à tout moment. Les événements représentés sur le diagramme fourni par Upshot peuvent être interrogés pour obtenir les informations qui leur ont été associés.

Upshot est un outil simple mais suffisamment intéressant pour mériter d'être utilisé. Il est très utile en particulier pour visualiser des informations sur une application en l'absence de traceur, par exemple si l'on a découpé son programme en parties logiques et que l'on veut observer la manière dont il exécute ces parties au cours du temps. Le fait de pouvoir associer des informations aux événements puis de les récupérer lors de l'inspection d'un événement permet de faire rapidement de la mise au point pour les performances spécifique à une application.

La version actuelle d'Upshot étant tout de même assez limitée, une nouvelle version est en phase finale de développement. Celle-ci offre deux visualisations supplémentaires et permet aussi de représenter des interactions entre les tâches (comme un envoi de message) sur le diagramme espace-temps, ainsi que des états imbriqués (figure 2.6 page suivante).

2.2.4 Pablo

Pablo (REED 1991, REED *et al.* 1992) est un environnement d'évaluation de performances général et complexe. Il a été conçu pour pouvoir traiter des traces de manière générique, pour pouvoir être facilement extensible et pour durer. Pablo n'est pas simplement un environnement de visualisation. Son but est de contrôler la totalité du processus d'évaluation de performance, de l'instrumentation du code à la visualisation de son exécution.

Les traces manipulées par Pablo sont stockées dans un format auto-descriptif, SDDF (pour *Self-Describing Data Format*). L'environnement étant générique, il n'a aucune connaissance de ce qu'il manipule et il conserve toutes les données de

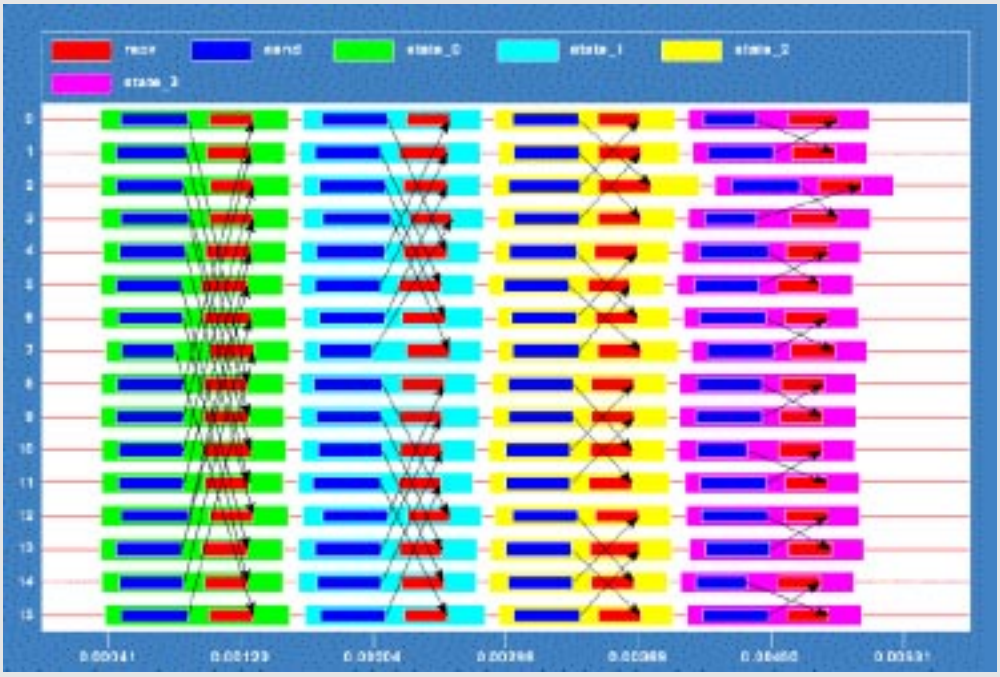


FIG. 2.6. – *La nouvelle visualisation espace-temps d'Upshot permet de représenter des interactions entre tâches et des états imbriqués comme le montre cette visualisation de l'exécution d'un algorithme de FFT avec 16 tâches.*

la trace et les indicateurs de performance qu'il calcule dans ce format.

Pablo est le premier environnement de visualisation pour l'évaluation de performance qui a été diffusé avec un langage de programmation visuelle intégré. Ce langage permet de construire des graphes d'analyse et de visualisation en connectant des composants entre eux, ces composants pouvant être des filtres d'analyse de données ou des visualisations. Les données passent d'un composant à l'autre, chaque composant les transformant avant de les transmettre aux composants auxquel il est connecté. Il permet également l'extension de l'environnement puisqu'il suffit de créer un nouveau composant pour qu'il soit intégrable à l'environnement ; cette intégration nécessite par contre une modification de l'environnement pour qu'il reconnaisse ce nouveau composant. Pablo est fourni avec un certain nombre de tels composants et il suffit de les connecter pour être prêt à mener une session d'évaluation de performance. Le système de programmation visuelle de

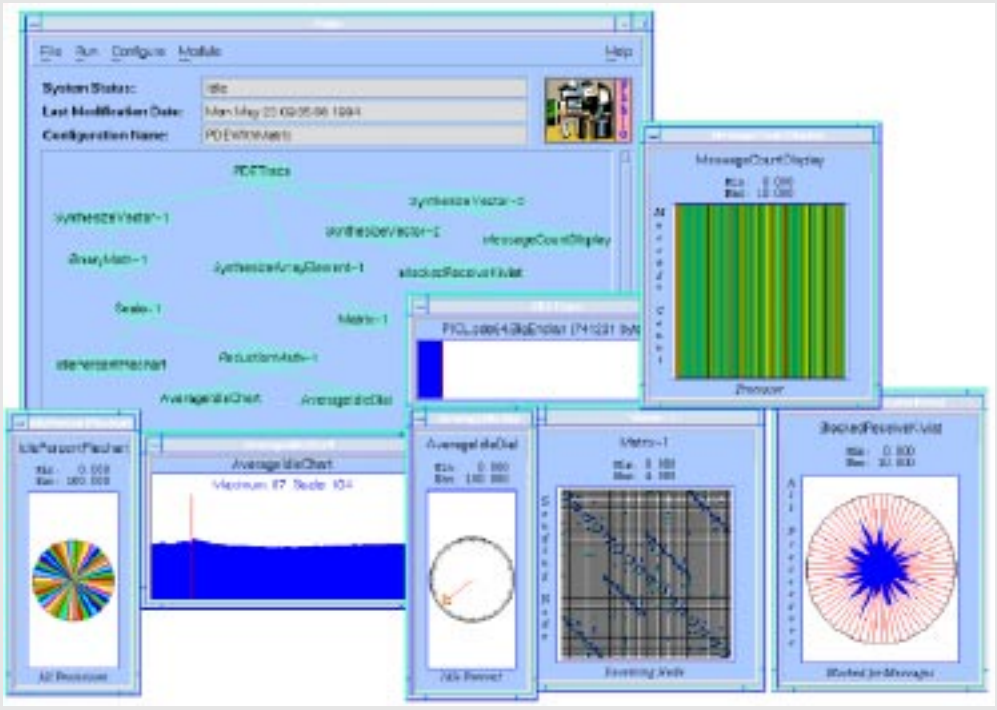


FIG. 2.7. – *L'environnement d'évaluation de performance Pablo. On notera la fenêtre représentant le programme visuel décrivant le graphe d'analyse et de représentation utilisé pour la session d'évaluation de performance en cours.*

Pablo utilise également SDDF pour la manipulation des données puisque c'est dans ce format que les données de la trace sont disponibles, et parce qu'il permet de réaliser des composants polymorphiques (c'est à dire acceptant par exemple des nombres sous forme d'entiers ou de réels et se chargeant des conversions nécessaires).

Les visualisations de Pablo sont uniquement statistiques. En effet, l'environnement ne connaît aucun modèle de programmation et ne dispose pas de simulateur permettant de reconstruire la suite des états d'une application à partir des événements générés pendant son exécution. Les visualisations ne sont pas non plus interactives. En revanche Pablo est le seul environnement à disposer d'un ensemble de composants de sonorisation qui peuvent être utilisés en complément des visualisations.

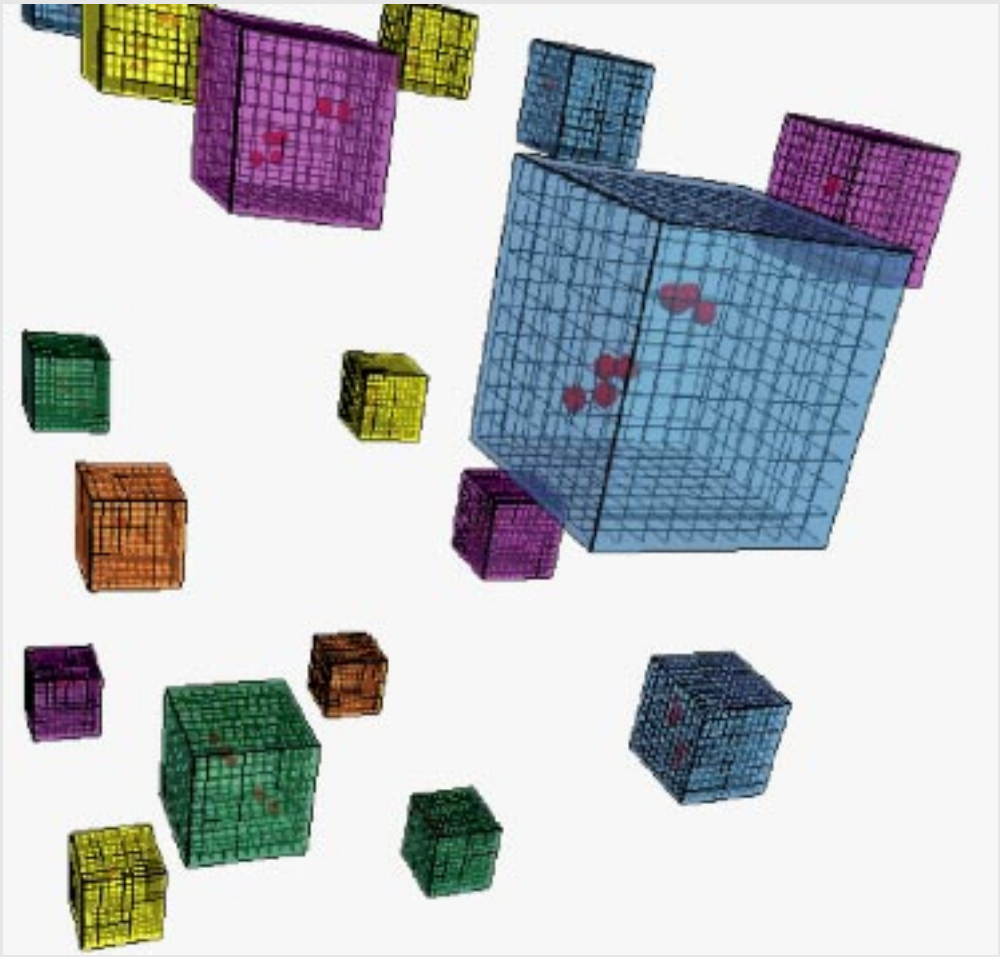


FIG. 2.8. – *Une visualisation expérimentale de Pablo. Les cubes représentent des espaces de métriques et les sphères représentent les processeurs « flottant » dans ces espaces tri-dimensionnels.*

Pablo a voulu être un environnement capable de traiter n'importe quel type de trace et d'effectuer des analyses extrêmement variées. Pour cela ses concepteurs ont défini des mécanismes très généraux afin d'être à même de traiter tous les cas qui pourraient se présenter. Si cette intention est louable, le résultat actuel est que Pablo est très difficile à utiliser sans apprentissage : la configuration du schéma d'analyse de la programmation visuelle est une tâche extrêmement lourde et qui

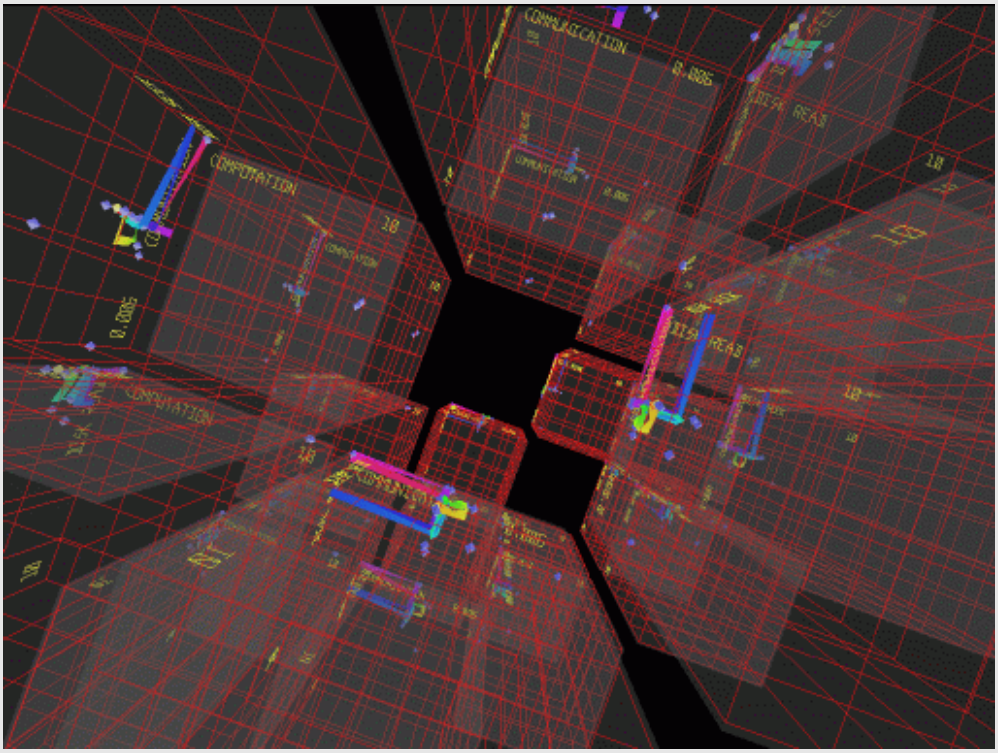


FIG. 2.9. – *Immersion dans un espace de données (ici l'ensemble des projections tri-dimensionnelles d'un jeu de données N-dimensionnelles) par utilisation de la réalité visuelle.*

ne donne pas envie de recommencer[2].

À notre connaissance l'utilisation de Pablo hors des milieux universitaires américains est toujours confidentielle trois ans après sa mise à disposition publique. On notera cependant que Pablo Par contre le projet continue toujours et sert de cadre à un certain nombre d'essais de visualisations expérimentales, notamment multi-dimensionnelles (figure 2.8 page précédente) ou faisant appel à des accessoires de réalité virtuelle (figure 2.9) ; mais ces vues sont encore difficilement exploitables. D'autres expériences en cours concernent par exemple l'intégration de Pablo avec l'environnement Fortran D de la *Rice University* (HIRANANDANI

2. L'exemple le plus simple d'utilisation de Pablo dans son guide de l'utilisateur comprend huit pages de configuration du schéma d'analyse !

et al. 1992), la conduite d'une session d'évaluation de performance en temps réel ou encore l'étude de méthodes statistiques de réduction de la taille des données à manipuler lors de l'évaluation d'applications importantes.

2.2.5 Paradyn

L'environnement Paradyn (MILLER, CALLAGHAN, CARGILLE, HOLLINGSWORTH, IRVIN, KARAVANIC, KUNCHITHAPADAM et NEWHALL 1994, MILLER, HOLLINGSWORTH et CALLAGHAN 1994) est récent et plutôt novateur, mettant en pratique un certain nombre d'idées inédites qui méritent de s'y intéresser. Il donne une bonne idée d'une des manières dont l'évaluation de performance des applications parallèles pourrait être conduite dans un futur proche.

La prise de traces dans Paradyn se fait en temps réel. Mais à la différence de XPVM, où cette prise de trace est très classique, Paradyn effectue celle-ci par une instrumentation dynamique du code de l'application au cours de son exécution. Cette instrumentation est dirigée par l'utilisateur, et l'application n'est effectivement tracée que lorsque ce dernier en a besoin. La mise en place des points de trace est une opération délicate et qui est toujours susceptible de perturber l'exécution de l'application puisqu'il faut suspendre ses tâches puis les relancer après insertion des instructions de prise de traces. Par contre, le problème de la taille des traces est résolu puisqu'il n'y a plus de stockage des informations avant leur exploitation.

Les traces prises par Paradyn ne sont pas non plus des traces classiques. Alors que jusqu'à présent tous les outils de prise de traces pour l'évaluation de performance généraient des événements correspondant aux changements d'états de l'application tracée, Paradyn échantillonne à intervalles réguliers un jeu d'indicateurs de performance spécifiés par l'utilisateur. Les traces sont plus petites que des traces classiques mais ne peuvent pas du tout servir à la reconstruction des états de l'application, par exemple.

Une session d'évaluation de performance sous Paradyn est contrôlée depuis une fenêtre contenant une représentation des différents modules de l'application évaluée, cette représentation indiquant quelle partie de l'application est en cours d'exécution (figure 2.10 page suivante).

La recherche d'un problème de performance avec Paradyn n'est pas du tout

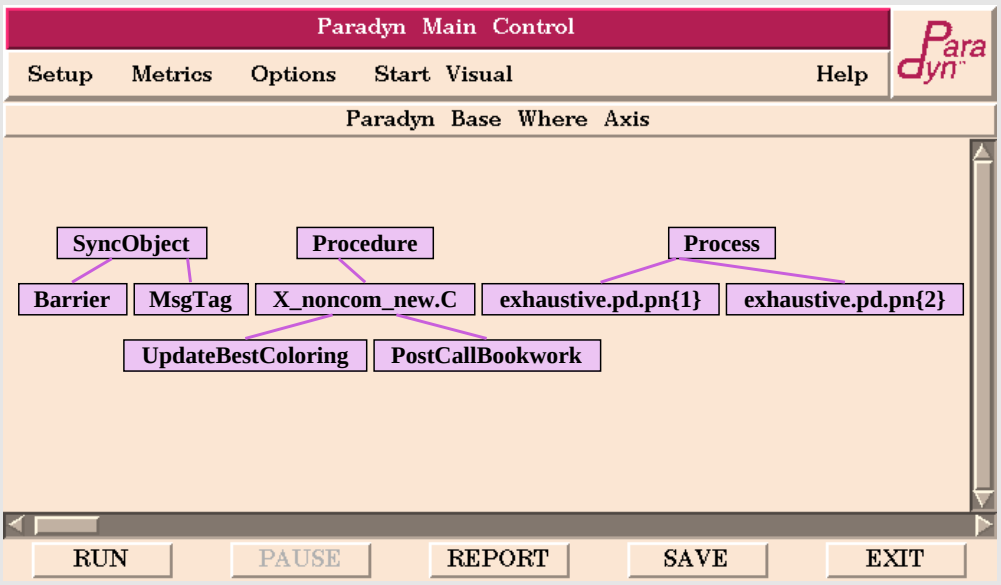


FIG. 2.10. – *Vue de la fenêtre de contrôle de l'environnement d'évaluation de performance Paradyn, présentant différents modules d'une application et indiquant où l'on se trouve dans l'exécution de celle-ci.*

la même qu'avec les autres environnements de visualisation. Alors que dans ces derniers on procède en observant l'exécution et en regardant si l'on peut voir une anomalie dans celle-ci, Paradyn cherche un problème de performance puis montre ce problème une fois qu'il l'a détecté (figures 2.11 page suivante et 2.12 page 47).

Cette recherche de problèmes est automatisée par ce que les auteurs de Paradyn appellent un « consultant en performance ». Ce consultant utilise un modèle de caractérisation des problèmes de performances suivant trois axes, W^3 (pour *Where, Why, When*). Ces axes correspondent à la localisation du problème dans l'application (axe *Where*), dans la séquence d'actions de celle-ci (axe *Why*) et dans le temps (axe *When*). Lorsqu'un indicateur de performance prend une valeur anormale, le consultant cherche le point correspondant au problème dans l'espace de recherche du modèle W^3 (en simplifiant, la date à laquelle l'anomalie est détectée fixe une coordonée sur l'axe *When*, l'instruction en cours d'exécution est liée à l'axe *Where* et l'axe *Why* correspond aux conditions ayant provoqué l'anomalie). Une fois le problème identifié Paradyn peut fournir une représentation graphique

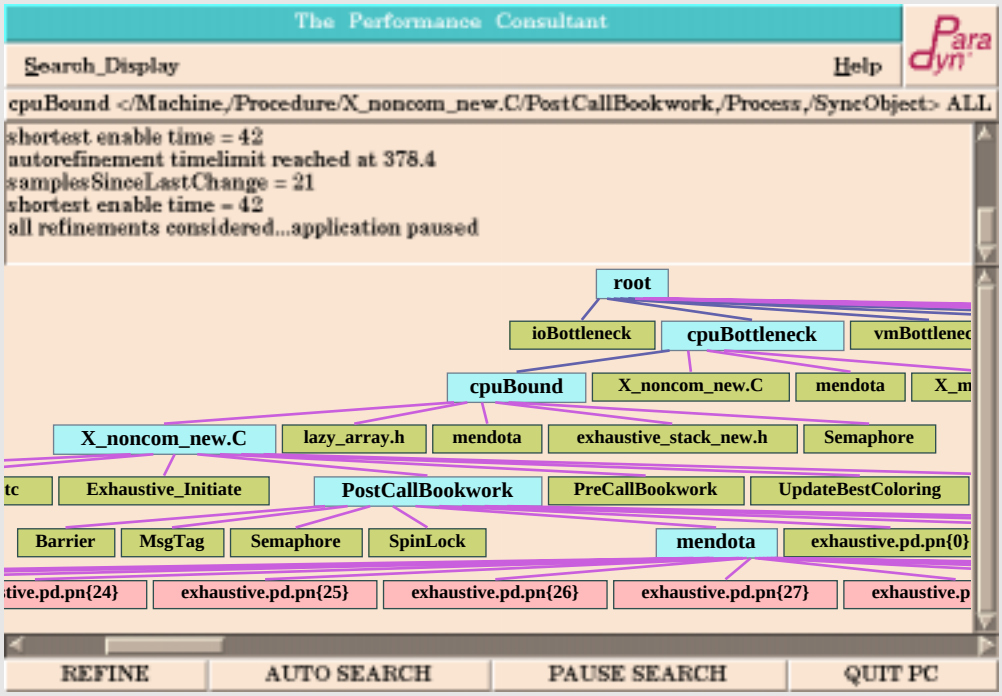


FIG. 2.11. – *Détection automatisée d'un problème avec Paradyn : le « consultant en performance » intégré à l'environnement indique le type de problème rencontré et sa localisation dans l'application. De haut en bas on trouve successivement les différentes classes de problèmes de performance identifiées par le consultant puis les endroits considérés comme pouvant être la cause du problème détecté. Le chemin dessiné en cyan définit le problème et sa localisation dans l'application..*

de l'évolution de l'indicateur concerné au cours du temps pour mettre en évidence l'anomalie.

Paradyn n'offre que deux visualisations non interactives, un tracé d'histogrammes et une visualisation d'une matrice d'indicateurs ; ces visualisations supportent un nombre d'informations à priori quelconque. L'environnement est toutefois doté d'une interface de programmation qui devrait lui permettre d'utiliser des visualisations extérieures en leur communiquant les indicateurs qu'il obtient au cours de l'exécution de l'application.

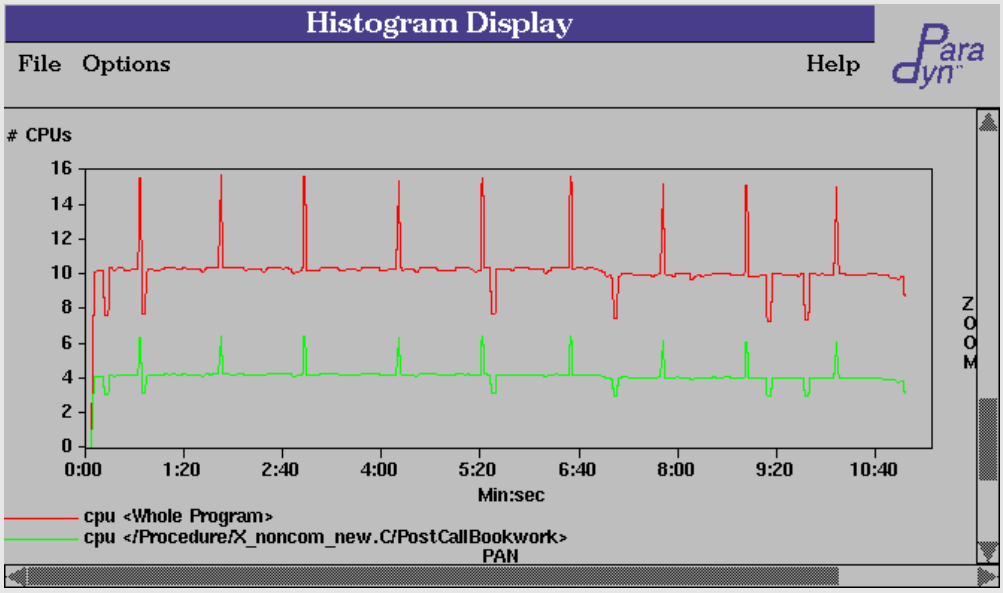


FIG. 2.12. – *Illustration d'un problème par Paradyn après sa détection : ici la courbe du haut montre la situation globale de l'application et celle du bas donne les informations relatives au problème de performance détecté.*

Les principaux inconvénients de Paradyn sont son absence de visualisations classiques (ou des données permettant de les produire) et surtout son manque de portabilité actuel. Il ne peut pour l'instant fonctionner qu'avec des applications écrites en PVM et pour une seule machine parallèle. Ceci est dû à l'instrumentation dynamique du code qui, outre le fait qu'elle a demandé des modifications à PVM, nécessite un accès à l'espace d'exécution de l'application sur la machine. Un tel accès n'est pas possible sur tous les systèmes ou peut exiger la réalisation de fonctions spécifiques de très bas niveau.

Nous pensons que le consultant en performance de Paradyn est représentatif des environnements d'évaluation de performance des prochaines années, car l'aide à la détection de problèmes de performance devrait prendre une grande part dans ceux-ci. Cette recherche de problèmes est le principal souci de l'utilisateur et, les applications parallèles devenant de plus en plus complexes, il est utopique de penser que l'utilisateur va pouvoir continuer à voir des indications de contre-performance perdues au milieu d'un gros volume d'information. La disponibilité

de ces aides déchargera également l'utilisateur du (long) travail d'observation et d'attente qui précède la détection d'un problème, et devrait l'aider à ne pas passer à côté de ce dernier.

Reste à savoir si l'instrumentation dynamique est effectivement une bonne solution et ne perturbe pas trop les applications. Et dans tous les cas Paradyn ne remplace pas les environnements de visualisation traditionnels qui sont capables de fournir des informations liées à la succession d'états de l'application, informations que Paradyn ne peut donner faute de traces adéquates.

2.3 Comparaison des environnements

Le tableau 2.1 page ci-contre donne une appréciation de la qualité des environnements que nous avons décrits. Cette appréciation est faite en fonction des objectifs présentés au chapitre 1 page 11.

On notera que les environnements les plus complexes ne sont pas forcément les mieux évalués : mieux vaut un environnement simple mais offrant des capacités telles que la visualisation de différents types de traces ou une interaction utile avec les visulisations qu'un environnement que sa difficulté de manipulation rend inadapté à un usage courant.

2.4 Conclusion

Nous avons présenté les principes de la prise de traces pour l'évaluation de performance d'applications parallèles ainsi que le fonctionnement des environnements de visualisation exploitant ces traces. Ces principes ont été illustrés par la présentation de quelques environnements représentatifs de l'état actuel de la recherche dans le domaine de la visualisation pour les performances.

À notre connaisance, aucun des environnements disponibles actuellement ne satisfait les objectifs que nous avons présenté au début de cette partie. C'est ce constat, ainsi que la volonté de se doter d'un environnement susceptible d'accueilir des expérimentations dans le domaine de l'évaluation de performance d'applications parallèles à partir de traces d'exécution, qui ont motivé la conception et la

Environnement	Ext.	Gén.	Pers.	Int.	Puis.	Conv.
ParaGraph	②	✪	②	✪	⑤	③
XPVM	✪	✪	✪	③	⑤	⑤
Upshot	✪	⑨	⑧	⑦	④	⑤
Pablo	⑦	⑨	✪	✪	⑦	①
Paradyn	⑤	⑤	✪	✪	⑦	

TAB. 2.1. – *Comparaison des environnements présentés par rapport à nos objectifs (extensibilité, généricité, personnalisation, interactivité, puissance et convivialité). L'échelle de notation va de 0 (représenté ici par ✪) à 10. L'évaluation de Paradyn est faite d'après la littérature s'y rapportant alors que les autres environnements ont été utilisés. Le critère de convivialité de Paradyn est absent car difficile à évaluer d'après la littérature..*

réalisation de Scope, l'environnement que nous présentons dans ce mémoire.

Bibliographie

ACM, *ACM/ONR Workshop on Parallel and Distributed Debugging, Proceedings*, t. XXVI de « ACM Sigplan Notices », 1991.

ARROUYE (Yves), *The* TAPE *Reference Manual*, LMC–IMAG, 46, av. Félix Viallet, 38031 Grenoble CEDEX 9, France, 1992.

BEGUELIN (Adam), DONGARRA (Jack J.), GEIST (G. A.), MANCHEK (Robert) et SUNDERAM (Vaidy S.), « Graphical Development Tools for Network-Based Concurrent Supercomputing », rapport technique, Oak Ridge National Laboratory, P.O. Box 2009, Bldg. 9207-A, Oak Ridge, TN 37831-808, 1991.

BEGUELIN (Adam), DONGARRA (Jack J.), GEIST (G. A.), MANCHEK (Robert), SUNDERAM (Vaidy S.), MOORE (Keith), WADE (Reed) et PLANCK (Jim), *HeNCE: A Users' Guide, Version 1.2*, Oak Ridge National Laboratory, P.O. Box 2009, Bldg. 9207-A, Oak Ridge, TN 37831-808, 1991.

BEMMERL (T.), BODE (Arndt), BRAUN (Peter), HANSEN (O.), LUKSCH (P.) et WISSMÜLLER (R.), « TOPSYS—Tools for Parallel Systems (User's Overview and User's Manual », SFB-Bericht 342/25/90 A, Lehrstuhl für Rechnertechnik und Rechnerorganisation, Institut für Informatik, Technische Universität, München, Arcisstr. 21, 8000 München 2, FRG, 1990.

DONGARRA (Jack) et TOURANCHEAU (Bernard), *Environments and Tools for Parallel Scientific Computing*. SIAM Press, 1994.

ERLANGEN-NÜRNBERG UNIVERSITÄT, *Simple User's Guide v5.3 — Part A: TDL Reference Guide*, Erlangen-Nürnberg Universität, IMDD VII, Martenstrasse 3, D-8520 Erlangen, Deutschland, 1992.

ERLANGEN-NÜRNBERG UNIVERSITÄT, *Simple User's Guide v5.3 — Part B: POET Reference Manual*, Erlangen-Nürnberg Universität, IMDD VII, Martenstrasse 3, D-8520 Erlangen, Deutschland, 1992.

ERLANGEN-NÜRNBERG UNIVERSITÄT, *Simple User's Guide v5.3 — Part D: FDL/VARUS Reference Guide*, Erlangen-Nürnberg Universität, IMDD VII, Martenstrasse 3, D-8520 Erlangen, Deutschland, 1992.

FRANCIONI (Joan M.), ALBRIGHT (Larry) et JACKSON (Jay Alan), « Debugging Parallel Programs Using Sound», dans *ACM/ONR Workshop on Parallel and Distributed Debugging, Proceedings*, p. 68–75, 1991.

FRANCIONI (Joan M.), ALBRIGHT (Larry) et JACKSON (Jay Alan), « The Sounds of Parallel Programs », dans *Sixth Distributed Memory Computing Conference Proceedings*, sous la direction de STOUT (Quentin) et WOLFE (Michael), p. 570–577, 1991.

GEIST (G. A.), HEATH (Michael T.), PEYTON (B. W.) et WORLEY (Paul H.), *PICL, A Portable Instrumented Communication Library, C Reference Manual*, Mathematical Sciences Section, Oak Ridge National Laboratory, P.O. Box 2009, Bldg. 9207-A, Oak Ridge, TN 37831-808, 1991.

GLENDINNING (Ian), GETOV (Vladimir), HELLBERG (Stephen), HOCKNEY (Roger) et PRITCHARD (David), « Performance Visualization in a Portable Parallel Programming Environment », dans *Workshop on Performance Measurement and Visualization of Parallel Systems*, sous la direction de HARING (Günter), 1992.

HACKSTADT (Steven T.) et MALONY (Allen D.), « Next-Generation Parallel Performance Visualisation: A Prototyping Environment for Visualization Development », rapport technique CIS-TR-93-21, Department of Computer and Information Science, University of Oregon, Eugene, OR 97403, 1993.

HARING (Günter), « Main Issues for Parallel Monitoring and Visualization Systems in the Future—General Discussion », dans *Workshop on Performance Measurement and Visualization of Parallel Systems*, sous la direction de HARING (Günter), 1992.

HARING (Günter), *Workshop on Performance Measurement and Visualization of Parallel Systems*, Austrian Center for Parallel Computation & Slovak Academy of Sciences, Springer-Verlag, Moravany, Czecho-Slovakia, 1992.

HEATH (Michael T.), « Visual Animation of Parallel Algorithms for Matrix Computation », dans *Fifth Distributed Memory Computing Conference Proceedings*, sous la direction de STOUT (Quentin) et WOLFE (Michael), p. 1213–125. IEEE Computer Society Press, 1990.

HEATH (Michael T.), « Recent Developments and Case Studies in Performance Visualization Using ParaGraph », dans *Workshop on Performance Measurement and Visualization of Parallel Systems*, sous la direction de HARING (Günter), 1992.

HEATH (Michael T.) et ETHERIDGE (Jennifer A.), *ParaGraph: A Tool for Visualizing Performance of Parallel Programs*, Oak Ridge National Laboratory, P.O. Box 2009, Bldg. 9207-A, Oak Ridge, TN 37831-808, 1991.

HEATH (Michael T.) et ETHERIDGE (Jennifer A.), « Visualizing the Performances of Parallel Programs », *IEEE Software*, 1991, t. V, n° 8, p. 29–39.

HERRARTE (Virginia) et LUSK (Ewing), *Studying Parallel Program Behavior with Upshot*, 1992.

HIRANANDANI (Seema), KENNEDY (Ken), KOELBEL (Chuck), KREMER (Ulrich) et TSENG (Chau-Wen), « An Overview of the Fortran D Programming System », rapport technique, Rice University, 1992.

HON (Robert W.), « Standard Trace Interchange Format », dans *Supercomputing '90 BOF Session on Standardizing Parallel Trace Formats*. Apple, Inc., 1990.

JÉZÉQUEL (Jean-Marc), « Building a Global Time on Parallel Machines », rapport technique 513, Irisa, 1990.

KILPATRICK (Carol) et SCHWAN (Karsten), « ChaosMON—Application-Specific Monitoring and Display of Performance Information for Parallel and Distributed System », dans *ACM/ONR Workshop on Parallel and Distributed Debugging, Proceedings*, p. 57–67, 1991.

KLAR (Rainer), « Event-Driven Monitoring of Parallel Systems », dans *Workshop on Performance Measurement and Visualization of Parallel Systems*, sous la direction de HARING (Günter), 1992.

KRAEMER (Eileen) et STASKO (John T.), « The Visualization of Parallel Systems: An OverView », *Journal of Parallel and Distributed Computing*, 1993, t. XVIII, n° 2, p. 105–117.

LEBLANC (T. J.), MELLOR-CRUMMEY (J. M.) et FOWLER (R. J.), « Analysing Parallel Programs Executions Using Multiple Views », *Journal of Parallel and Distributed Computing*, 1990, t. VIII, n° 6, p. 203–217.

MAILLET (Éric), TAPE*/PVM an efficient performance monitor for PVM applications — User guide*, LMC–IMAG, 46, av. Félix Viallet, 38031 Grenoble CEDEX 9, France, 1995.

MAILLET (Éric) et TRON (Cécile), « On Efficiently Implementing Global Time for Performance Evaluation on Multiprocessor Systems », *Journal of Parallel and Distributed Computing*, 1995, t. XXVIII, n° 1, p. 84–93.

MALONY (Allen D.), HAMMERSLAG (David H.) et JABLONOWSKI (David J.), « Traceview: A Trace Visualization Tool », *IEEE Software*, 1991, t. VIII, n° 5, p. 19–28.

MALONY (Allen D.) et NICHOLS (Kathleen M.), « Standards in Performance Instrumentation and Visualization for Parallel Computer Systems: Working Group Summary », 1991.

MALONY (Allen D.) et REED (Daniel A.), « Performance Analysis, Visualization with Hyperview », dans *IEEE Software* (NICHOLS 1990), 1990, p. 21–30.

MILLER (Barton P.), « What to Draw? When to Draw? An Essay on Parallel Program Visualization », *Journal of Parallel and Distributed Computing*, 1993, t. XVIII, n° 2, p. 265–269.

MILLER (Barton P.), CALLAGHAN (Mark D.), CARGILLE (Jonathan M.), HOLLINGSWORTH (Jeffrey K.), IRVIN (R. Bruce), KARAVANIC (Karen L.), KUNCHITHAPADAM (Krishna) et NEWHALL (Tia), « The Paradyn Parallel Performance Measurement Tools », rapport technique, Computer Sciences Department, University of Wisconsin-Madison, 1210 W. Dayton Street, Madison, WI 53706, 1994.

MILLER (Barton P.), HOLLINGSWORTH (Jeffrey K.) et CALLAGHAN (Mark D.), *The Paradyn Parallel Performance Tools and PVM*, dans DONGARRA et TOURANCHEAU (1994), chap. 1, 1994.

NICHOLS (Kathleen M.), « Performance Tools », *IEEE Software*, 1990, p. 21–30.

OBELÖER (Wolfgang), WILLEKE (Harald) et MAEHLE (Erik), « Performance Measurement and Visualization of Multi-Transputer Systems with DELTA-T », dans *Workshop on Performance Measurement and Visualization of Parallel Systems*, sous la direction de HARING (Günter), 1992.

REED (Daniel A.), « Scalable Performance Environments for Parallel Systems », dans *Sixth Distributed Memory Computing Conference Proceedings*, sous la direction de STOUT (Quentin) et WOLFE (Michael), p. 562–569, 1991.

REED (Daniel A.), AYDT (Ruth A.), MADHYASTHA (Tara M.), NOE (Roger J.), SHIELDS (Keith A.) et SCHWARTZ (Bradley W.), « An Overview of the Pablo Performance Analysis Environment », rapport technique, Department of Computer Science, University of Illinois, Urbana, Illinois 61801, 1992.

ROMAN (Gruia-Catalin) et COX (Kenneth C.), « A Taxonomy of Program Visualization Systems », *IEEE Software*, 1993, t. XXVI, n° 12, p. 11–24.

SARAIYA (Nakul P.), « Simple/Care concurrent-application analyzer », dans *IEEE Software* (NICHOLS 1990), 1990, p. 21–30.

SARAIYA (Nakul P.), NISHIMURA (Sayuri) et DELAGI (Bruce A.), « SIMPLE/CARE—An Instrumented Simulator for Multiprocessor Architectures », rapport technique, Knowledge Systems Laboratory, Stanford University, Stanford CA 94305, 1990.

SCHÄFERS (Lorenz) et SCHEIDLER (Christian), « A Graphical Programming Environment for Parallel Embedded Systems », 1992.

STOUT (Quentin) et WOLFE (Michael), *The Sixth Distributed Memory Computing Conference Proceedings*, IEEE Computer Society Press, Portland, Oregon, 1991.

TOURANCHEAU (Bernard) et VAN RIEK (Maurice G.), « GPMS, General Parallel Monitoring System », rapport technique, Laboratoire de l'Informatique du Parallélisme, École National Supérieure de Lyon, 46, allée d'Italie, 69364 Lyon CEDEX 7, France, 1992.

VAN RIEK (Maurice G.), « A General Approach to the Monitoring of Distributed Memory Machines — A survey », rapport technique, Laboratoire

de l'Informatique du Parallélisme, École National Supérieure de Lyon, 46, allée d'Italie, 69364 Lyon CEDEX 7, France, 1991.

VAN RIEK (Maurice G.) et TOURANCHEAU (Bernard), « A brief presentation of TMS », rapport technique, Laboratoire de l'Informatique du Parallélisme, École National Supérieure de Lyon, 46, allée d'Italie, 69364 Lyon CEDEX 7, France, 1991.

WORLEY (Paul H.), « A new PICL trace file format », rapport technique ORNL-12 125, Mathematical Sciences Section, Oak Ridge National Laboratory, P.O. Box 2009, Bldg. 9207-A, Oak Ridge, TN 37831-808, 1992.

II

L’environnement Scope

1

Architecture de Scope

UN ENVIRONNEMENT de visualisation pour l'évaluation de performance est un logiciel dont le coût de développement est très important. Il importe donc de rentabiliser l'investissement effectué en s'attachant à concevoir un tel environnement de manière à ce que celui-ci perdure et soit capable de supporter des changements dans la façon dont est faite l'évaluation des performances d'une application.

Nous avons vu au § 1.1 page 12 que de nombreux facteurs peuvent provoquer ce besoin de changement. Un environnement d'évaluation de performance doit être à même d'intégrer ces changements sans pour autant que cette intégration se traduise par la nécessité de modifier l'environnement de manière importante. Idéalement l'environnement ne sera même pas changé : seule son interface avec l'utilisateur changera de telle sorte que les outils qui lui sont proposés sont effectivement les mieux adaptés au modèle de programmation qu'il utilise et à la tâche qu'il désire effectuer.

Une idée maîtresse devant présider à la conception de l'environnement est son extensibilité : il faut que l'on puisse accroître ses capacités non seulement à moindre coût mais également de manière à ce qu'il intègre des techniques dont nous n'avons pas forcément l'idée actuellement. Le pendant de cette flexibilité est la généricité des outils disponibles dans l'environnement : ceux-ci doivent être conçus afin d'être réutilisables dans le plus large contexte possible afin qu'ils

puissent servir pour différents modèles de programmation par exemple.

Ce chapitre présente les choix de conception que nous avons faits afin que Scope apporte une réponse satisfaisante à ces problèmes d'extensibilité et de généricité. Nous présentons tout d'abord l'organisation générale de l'environnement puis les caractéristiques de ses différents types de composants. Nous finissons par la présentation des mécanismes de contrôle des sessions d'évaluation de performance de Scope.

1.1 Environnement de développement

Le choix d'un environnement de développement conditionne non seulement la qualité et la puissance des outils disponibles mais aussi les possibilités des applications réalisées grâce à cet environnement. Les environnements de développement traditionnels dans le monde de la visualisation pour les performances sont basés sur X et Motif. Bien que ces derniers permettent effectivement la réalisation d'interfaces complexes et puissantes, leur temps d'apprentissage est très important.

Comme nous cherchions non pas à nous conformer à un « standard » mais plutôt à réaliser un environnement souple et puissant tout en essayant de minimiser le coût de son développement, nous avons choisi d'utiliser NEXTSTEP (NEXT COMPUTER, INC. 1994).

NEXTSTEP comprend un système d'exploitation basé sur Mach mais surtout une interface graphique très évoluée et conviviale. Celle-ci permet facilement la manipulation directe et le glisser-jeter (*drag and drop*) généralisé (*i.e.* au sein d'une application mais aussi entre différentes applications) ; elle offre de plus une interface de programmation orientée objet de très haut niveau et d'excellente facture réalisée en Objective C (WIENER 1991, NEXT COMPUTER, INC. 1992). Les applications peuvent dessiner en deux dimensions grâce à Display PostScript (ADOBE SYSTEMS INCORPORATED 1992), une extension de PostScript adaptée aux supports de faible résolution tels que les écrans et qui gère un accès multiprogrammé aux ressources de l'écran, et en trois dimensions grâce à RenderMan (UPSTILL 1992). Ces deux types d'interfaces de dessin sont encapsulées dans des classes destinées à en faciliter l'usage.

NEXTSTEP offre également des outils de prototypage très puissants permettant par exemple de concevoir et tester une interface utilisateur sans avoir à programmer. Les applications réalisées sous NEXTSTEP peuvent facilement être rendues modulaires et extensibles grâce aux mécanismes d'édition de liens dynamiques utilisés en standard par l'environnement.

Il existe également un grand nombre de bibliothèques de classes d'interface de très haute qualité dans le domaine public. Leur disponibilité tend à réduire le temps de programmation et de mise au point et facilite l'interaction des applications nouvellement écrites avec celles qui existent déjà.

Un autre point qui nous importait est que l'interface utilisateur de NEXTSTEP est très précisément définie, et que les outils de développement supportent les règles de programmation d'interfaces en vigueur. Ceci garantit que le temps d'apprentissage de l'interface d'une nouvelle application par l'utilisateur est faible et que les applications fonctionnent bien ensemble.

Enfin NEXTSTEP fonctionne actuellement sur stations NeXT, machines compatibles PC à base de processeurs Intel i386, systèmes Sparc (tels que ceux de Sun) et stations HP PA-RISC. Les spécifications de la partie interface utilisateur de NEXTSTEP ont été publiées sous le nom d'OpenStep. Les applications OpenStep fonctionnent sur tous les systèmes NEXTSTEP ainsi que sur Solaris, Windows NT et Windows 95 ; un portage sur X est également en cours et sera disponible gratuitement. Les applications NEXTSTEP et OpenStep sont portables et le parc de machines susceptibles de les faire fonctionner est déjà important et croît régulièrement.

1.2 Organisation générale

1.2.1 Découpage logique

Si nous essayons de découper un environnement de visualisation générique en parties nous pouvons distinguer d'une part une partie produisant les visualisations et d'autre part une partie permettant de contrôler l'environnement. La production des visualisations est le fruit de la transformation des données représentant une exécution : il faut d'abord obtenir ces données qui sont ensuite manipulées puis présentées sous une forme assimilable par l'utilisateur. Nous distinguons donc

non pas deux mais trois parties principales: la première correspond à l'acquisition et à la production des données correspondant à l'exécution de l'application à évaluer, la deuxième englobe tout ce qui concerne l'évaluation, *i.e.* l'analyse et la visualisation d'indices de performances et la dernière permet de contrôler l'environnement et la session d'évaluation de performances (choix de la trace, des outils d'évaluation, déplacement dans l'exécution de l'application, etc.).

Dans le cas d'une visualisation dirigée par les traces la première partie consiste à extraire les événements de la trace décrivant l'exécution d'une application et à se servir de ces événements pour reconstituer les états successifs de l'application : c'est le rôle de la lecture de traces et de la simulation qui sont décrites au chapitre 2 page 87. Cette partie est dépendante du modèle de programmation utilisé et de la syntaxe des traces ce qui signifie qu'il faut des outils spécifiques à chaque format de traces et à chaque modèle de programmation.

En revanche la seconde partie a un caractère générique : un calcul de moyenne n'a pas besoin de connaître la signification des données sur lesquelles il calcule ; un histogramme ou une courbe peuvent être utilisés indifféremment pour représenter des tailles de messages ou des pourcentages d'utilisation des ressources de calcul ; une représentation de graphe peut aussi bien montrer la topologie d'une application qu'un graphe d'appel de procédures à distances ; etc. Afin qu'ils soient réutilisables les outils ne connaissent absolument pas la sémantique des données qu'ils manipulent : c'est l'utilisateur qui donne une sémantique en choisissant quelles données sont fournies à quels outils.

Quant à la dernière partie elle fournit l'interface entre l'utilisateur et l'environnement en lui donnant les outils nécessaires à la conduite d'une session d'évaluation de performances. Cette interface permet non seulement d'effectuer des actions telles que le choix d'une trace et des outils qui seront utilisés pour son analyse mais aussi de contrôler l'action de ces outils ainsi que la façon dont la trace est déroulée durant l'évaluation de performance. Elle fournit également un cadre dans lequel les différents outils disponibles peuvent s'intégrer pour présenter une interface de manipulation cohérente à l'utilisateur. Enfin elle se charge de présenter une interface générique permettant de manipuler les outils de la partie spécifique afin que l'utilisateur n'ait pas à changer sa manière d'utiliser l'environnement lorsque ces outils changent.

On comprend alors quel est l'intérêt de ce découpage logique en trois parties (figure 1.1 page 64) : il permet d'isoler soigneusement les outils qui devront être

réécrits pour chaque modèle de programmation et encourage l'environnement à se doter de moyens de cacher leur spécificité à l'utilisateur. Mais il garantit aussi que les outils d'analyse de données et de visualisation réalisés seront réellement génériques puisqu'ils n'auront pas un accès direct à la trace, un tel accès risquant d'induire une dépendance de ces outils par rapport aux formats de traces ou d'événements exploités pour l'évaluation de performance.

1.2.2 Principes directeurs

Nous distinguons trois niveaux de facilité d'extension à fournir aux utilisateurs de Scope. Le plus simple permet à l'utilisateur d'ajouter le calcul de nouveaux indices de performances ou de changer les données visualisées ou les données elles-mêmes ; ce niveau ne doit pas requérir de connaissance du fonctionnement des parties de l'environnement ni du langage d'implantation de l'environnement. Le plus complexe autorise l'ajout de nouveaux objets complexes dans l'environnement (un lecteur de traces, un simulateur, une visualisation, etc.), ces objets pouvant avoir une interaction complexe avec l'environnement ; une telle modification requiert la compréhension des mécanismes internes de l'environnement et de la programmation. Enfin le niveau intermédiaire autorise l'ajout de nouveaux objets simples dans l'environnement (analyse de données, statistiques, filtrage, etc.) ; il nécessite la programmation de ces objets et celle-ci doit être simplifiée au maximum.

Pour permettre l'extension facile de l'environnement par les utilisateurs sans leur demander une grande expertise en programmation nous avons doté Scope d'un langage de programmation visuelle, Eve (pour « Environnement visuel extensible »), qui permet de construire graphiquement des « programmes » sous forme d'un graphe de composants de manipulation de donnés.

Ce langage, que nous décrivons au § 1.3 page 65, est la base de Scope : l'environnement est formé par un ensemble d'objets ou composants qui transforment des données et se les transmettent. Ces composants sont incorporés dynamiquement dans l'environnement en fonction des besoins et des choix de l'utilisateur, offrant ainsi une très grande souplesse puisque l'ensemble de l'environnement peut être à tout instant modifié selon la volonté de l'utilisateur. Cette modification de l'environnement correspond au niveau d'extension simple. Les autres niveaux d'extension demandent simplement l'écriture de composants Eve, certains

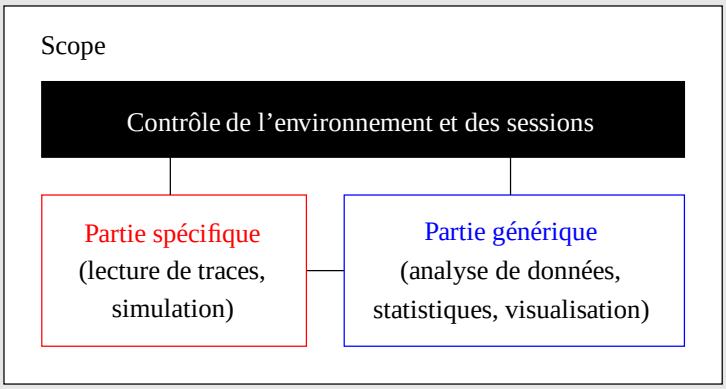


FIG. 1.1. – *Les différentes parties qui composent Scope sont interconnectées mais c'est le contrôle de l'environnement et des sessions qui contrôle l'interaction principale avec l'utilisateur.*

étant simplement plus complexes que d'autres puisqu'ils doivent obéir à plus de contraintes (par exemple s'intégrer avec le contrôle de session de Scope pour les lecteurs de traces).

Cette architecture a de nombreux intérêts. Les différents outils de l'environnement étant isolés sous forme de composants Eve il est facile de prototyper un nouvel outil sans influencer le comportement ou la stabilité d'outils déjà disponibles. L'intégration dynamique de composants dans l'environnement signifie que l'utilisateur peut ajouter de nouveaux outils pour l'aider au moment où il le souhaite, *i.e.* après avoir découvert un problème de performance par exemple. Cela veut donc dire que l'environnement n'utilise de la mémoire et du temps machine que pour ce qui est réellement nécessaire : il est susceptible d'être non seulement plus rapide que si tous les outils étaient toujours disponibles mais aussi capable de traiter de plus grandes traces. Cela signifie également qu'il n'est pas nécessaire de modifier l'environnement lui-même pour l'étendre : il suffit d'écrire un nouveau composant et de l'y incorporer. Outre le fait d'isoler le composant cette capacité permet un échange facile des extensions de l'environnement au sein d'un groupe d'utilisateurs. Enfin le fait de se servir d'un langage de programmation visuelle fournit une interface de développement d'outils bien spécifiée et permet de faire bénéficier tous les composants développés des améliorations apportées au langage. Par exemple si celui-ci comporte la possibilité d'exécuter un compo-

sant sur une machine distante, ce qui est très facile à réaliser, tous les composants en bénéficieront et il sera possible de placer les composants les plus coûteux en temps d'exécution ou en mémoire sur des machines puissantes, construisant ainsi un environnement distribué d'évaluation de performance.

Les programmes visuels qui sont réalisés constituent ce que nous appelons des schémas d'évaluation car ils permettent l'analyse et la visualisation du système à évaluer[1]. Ces schémas peuvent être sauvés et réutilisés ce qui signifie qu'il est possible de se constituer une bibliothèque de schémas spécialisés dans certaines tâches. En utilisant les schémas adaptés, en les combinant et éventuellement en les modifiant l'utilisateur peut facilement obtenir les informations exactes dont il a besoin. Ces schémas peuvent également être échangés et il est possible d'envisager qu'un utilisateur expert diffuse des schémas dédiés à des tâches d'évaluation très précises.

Scope contrôle Eve au sens où il est capable de modifier les programmes visuels, de contrôler leur création, leur chargement et leur exécution. Ceci est fait à travers le mécanisme de contrôle de session d'évaluation de performances que nous décrivons au § 1.6 page 80. À l'inverse Eve contrôle Scope car ce sont les composants des programmes visuels qui déterminent les actions possibles de l'utilisateur et l'état de l'environnement.

Ce contrôle mutuel (ou cette coopération si l'on préfère) permet d'avoir un environnement intégré et cohérent. Hormis sa base même, tout le développement de Scope est fait simplement en écrivant des composants pour la programmation visuelle. Il n'y a pas de traitement spécifique des composants suivant l'usage auquel ils sont destinés et tous peuvent être indifféremment remplacés sans que leur changement ne nécessite une altération de Scope.

1.3 Programmation visuelle

La programmation visuelle (CHANG 1990, TANIMOTO 1990, CHABERT 1991, SHU 1988) est une technique de construction graphique de programmes. Elle a été appliquée dans un premier temps à la construction de programmes Lisp et Smalltalk puis ses applications se sont diversifiés quand sont apparus des langages de

1 . Nous emploierons également les appellations de schémas d'analyse et de schémas d'analyse et de visualisation.

programmation visuelle. Ceux-ci ont d'abord été utilisés dans des outils de traitement d'images mais leur usage se généralise et ils sont de plus en plus disponibles comme un moyen de personnaliser et d'étendre un environnement de manipulation de données. On parle alors d'environnement de programmation visuelle pour désigner et le langage et l'interface qui permet de composer et d'exécuter des programmes.

1.3.1 Principes

Les langages de programmation visuelle qui nous intéressent sont des langages où les programmes sont représentés sous forme de *graphes de flot de données* (figure 1.2 page suivante). Ces graphes ont pour sommets des objets ou *composants* transformant les données qu'ils reçoivent et leurs arêtes représentent les chemins que peuvent prendre les données. (Dans ce qui suit nous parlerons de programmation visuelle pour parler de ce type de langages uniquement.)

Un composant est formé par un certain nombre de *ports* d'entrée et de sortie et d'une fonction transformant les entrées reçues en sorties. Le système de programmation visuelle s'occupe de propager les données entre les composants : il sélectionne un composant prêt à se *déclencher* (*i.e.* ayant reçu les données dont il a besoin), le déclenche et transmet les données disponibles sur les ports de sorties de ce composant à tous les composants qui y sont connectés, puis recommence. Il est possible de connecter plusieurs entrées à un même port de sortie. L'utilisateur écrit ses programmes en posant des composants sur un plan de travail puis en les connectant. La plupart du temps le programme s'exécute dès que des données sont disponibles et s'arrête dès qu'il est impossible de propager des données : c'est ce qu'on appelle un programme *vivant*.

Les langages de programmation visuelle disponibles diffèrent principalement suivant leurs possibilités : certains ont des ports typés alors que d'autres non, le déclenchement des composants peut être synchrone (toutes les entrées doivent être disponibles) ou asynchrone (il suffit qu'une entrée soit disponible), les possibilité de connexion entre composants sont plus ou moins restreintes, etc.

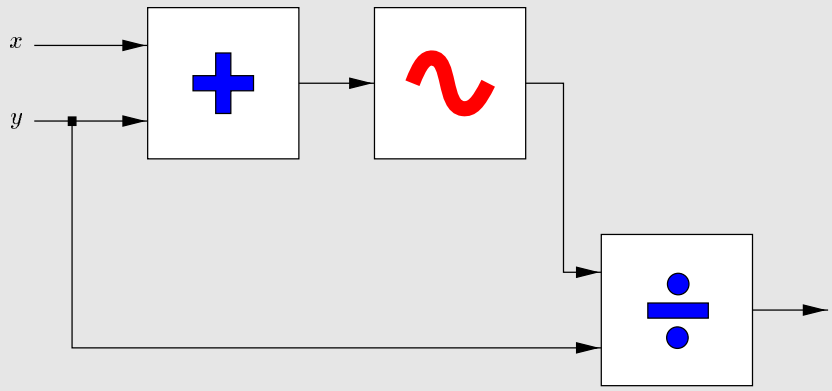


FIG. 1.2. – *Un programme visuel simple pour calculer* $\sin(x+y)/y$. *Les données sont transportées le long des liens et lorsqu'un composant a reçu les données dont il a besoin il se déclenche, transformant les données et en produisant éventuellement de nouvelles qui sont alors propagées dans le graphe.*.

1.3.2 Eve (environnement visuel extensible)

Il existe quelques environnements de programmation visuelles disponibles publiquement, les plus connus étant ceux qui sont fournis avec l'environnement de traitement d'images Khoros (RASURE et ARGIRO n.d.) et avec l'environnement de visualisation Pablo (REED *et al.* 1992). Malheureusement la programmation visuelle dans ces environnements est extrêmement liée au code de l'environnement lui-même et est très difficile à extraire. De plus ces environnements sont conçus pour X et le portage de leurs interfaces de programmation visuelle est une tâche importante ; étant écrits en C ou C++, ils demanderaient en plus la réalisation d'une couche d'interface pour leur manipulation sous forme de classes Objective C. Enfin aucun de ces langages de programmation visuelle n'est à la fois assez simple d'utilisation et assez évolué pour supporter la réalisation de programmes complexes. VIVA (TANIMOTO 1990) est un langage de programmation visuelle pour NEXTSTEP. Il n'est malheureusement pas disponible, ses sources ayant apparemment été perdues.

Eve (ARROUYE 1995) est l'environnement de programmation visuelle que nous avons développé dans le cadre de notre travail sur Scope. Il peut être intégré dans n'importe quelle application pour permettre sa personnalisation et son exten-

sion. Scope est simplement un exemple d'utilisation d'Eve dans une application : Eve en soi est totalement indépendant de Scope.

L'environnement se présente sous la forme de deux bibliothèques d'objets. La première comporte les objets décrivant les composants et leurs ports ainsi que les programmes ; elle conditionne également la manière dont l'exécution d'un programme visuel est régie. La seconde contient les objets nécessaires à l'interface de programmation visuelle elle-même : apparence et interaction des composants à l'écran, gestion du plan de travail (placement et connexion des composants), sauvegarde et restauration de programmes, etc. Cette séparation entre l'interface et la mécanique de programmation visuelle elle-même permet d'envisager le portage d'Eve sur des environnements autres que NEXTSTEP et facilitera l'exécution distribuée en milieu hétérogène de tout ou partie d'un programme visuel. Elle permet également de changer l'interface de la programmation visuelle sans avoir à modifier cette dernière.

Données manipulées

Les données circulant entre les composants sont des objets quelconques. Lorsqu'une tentative de connexion entre deux ports est réalisée un port peut refuser la connexion si la classe des objets de l'autre port ne lui convient pas. Ce mécanisme permet de typer les ports des composants et de garantir que seules les données qu'un composant sait manipuler lui seront transmises.

Par défaut les composants travaillent sur les données d'origine et non sur des copies de celles-ci. Cela autorise une plus grande vitesse d'exécution mais signifie qu'un effet de bord inattendu modifiant une des données manipulées par un composant sera immédiatement perceptible dans tout le programme, y compris en amont du composant ayant produit l'effet de bord. Mais il est évidemment possible à n'importe quel composant de générer des copies des données qu'il manipule s'il en a besoin ou s'il souhaite transmettre des données sans qu'elles puissent être modifiées.

Composants

Les composants d'Eve disposent évidemment de ports d'entrée et de sortie. Mais contrairement à ce qui existe dans la plupart des langages de programmation

visuelle les ports d'entrée (et donc le comportement des composants) peuvent indifféremment être synchrones ou asynchrones.

Lorsqu'un port asynchrone reçoit une donnée, il la transmet immédiatement au composant qui la traite alors avec une méthode spécifique aux entrées asynchrones. Lorsqu'un port synchrone reçoit une donnée, en revanche, le composant vérifie tout d'abord si cette donnée était la dernière attendue pour pouvoir déclencher le composant. Si c'est le cas, le composant est déclenché et utilise les données disponibles sur ses ports d'entrée ; dans le cas contraire, il ne se passe rien.

Traditionnellement un composant n'accepte des données en entrée que s'il a déjà consommé les données reçues précédemment et un composant n'a fini de se déclencher que lorsqu'il a pu transmettre toutes ses sorties. Cela qui signifie que si un composant a un problème (par exemple s'il ne reçoit pas une donnée dont il a besoin ou s'il est bogué) il bloque l'ensemble du système mais également que le composant le plus lent conditionne la vitesse de propagation des données dans le graphe (figure 1.3 page suivante). Dans Eve les ports d'entrée et de sortie gèrent un tampon de données afin d'éviter le blocage des composants ou la perte de données qui résulterait d'un non-blocage en cas d'absence de tampons. Tant qu'un port d'entrée accepte des données en entrée celles-ci lui sont transmises, et ensuite elles sont gardées dans le tampon du port de sortie émetteur si besoin est. La taille maximale des tampons est choisie par le composant.

Un des point originaux d'Eve est la possibilité de multiplexer les ports d'entrée, c'est-à-dire de connecter les ports de sortie de plusieurs composants sur un même port d'entrée. Les données sont alors consommées par le composant dans l'ordre où elles sont arrivées. Cela se révèle être très utile lorsque des composants ont des comportement récursifs, par exemple (figure 1.4 page 71).

Les composants ont enfin la possibilitée de gérer un certain nombre de leurs entrées sous forme de ports de contrôle. Cette séparation est entièrement logique, mais se traduit dans l'interface graphique de construction des programmes par un placement différent des ports suivants qu'ils sont dits d'entrée ou de contrôle, cette différence permettant à l'utilisateur de distinguer facilement les différents types de ports offerts par les composants d'un graphe de programmation visuelle. Une utilisation courante des ports de contrôle est de permettre la réception asynchrone de données influant sur le comportement du composant (par exemple pour lui indiquer de changer de manière de traiter ses données, ou pour provoquer une réinitialisation d'une partie de son état).

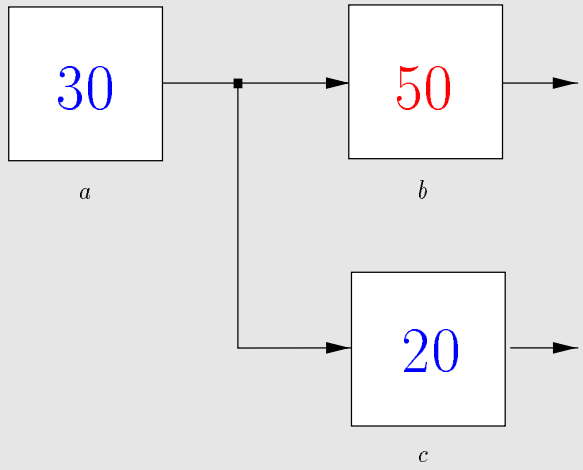


FIG. 1.3. – *Si l'on considère d'une part que les nombres portés sur les composants donnent le temps nécessaire au calcul qu'ils effectuent et d'autre part que le composant a produit des données toutes les 30 unités de temps, alors c ne peut recevoir des données que toutes les 80 unités de temps s'il n'y a pas de tampons de données. Après s'être déclenché une première fois, a peut transmettre toutes ses données à l'instant 30 mais lorsqu'il veut transmettre les données suivantes à l'instant 60 il est bloqué car b n'a toujours pas libéré son port d'entrée ; étant bloqué il ne pourra pas se déclencher à nouveau avant l'instant 80, retardant ainsi la délivrance de données à c. En revanche si les données sont tamponnées c les reçoit bien toutes les 50 unités de temps..*

La figure 1.5 page ci-contre représente un graphe Eve qui montre les différents types de ports que peuvent avoir les composants, ainsi que la façon dont ces ports peuvent être connectés. Elle met en évidence des points importants comme la possibilité de mélanger ports synchrones et asynchrones dans un même composant ou encore le multiplexage des données en entrée d'un composant. L'interface graphique que l'on voit est celle que nous avons développée pour la manipulation des programmes visuels sous NEXTSTEP (cf. 1.3.2 page 74).

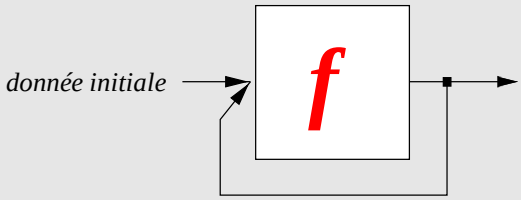


FIG. 1.4. – *Exemple de multiplexage des entrées d'un composant. Le composant présenté utilise une fonction récursive pour produire une suite de données : une fois la donnée initiale arrivée, le déclenchement du composant produit une donnée qui est propagée non seulement vers d'autres composants mais également vers sa propre entrée, ce qui provoque un nouveau déclenchement..*

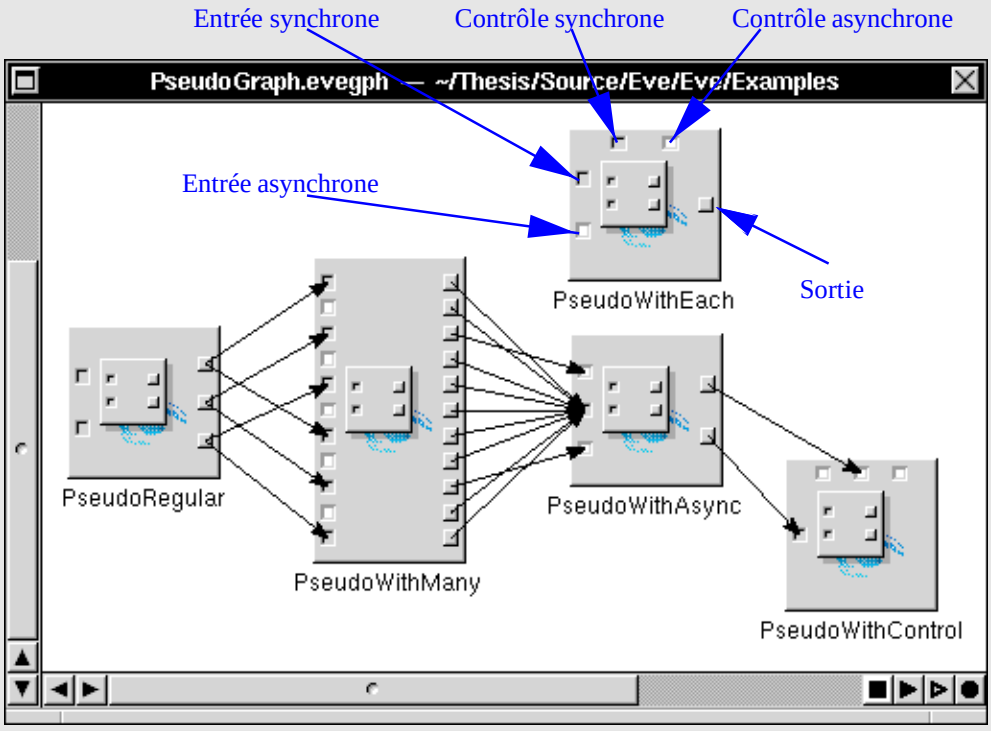


FIG. 1.5. – *Types de ports et connexions en Eve..*

Mode de propagation des données

Nous avons vu précédemment que le principe général de la propagation des données dans le graphe est le suivant : un composant est prêt à se déclencher (tous ses ports d'entrée sont remplis), il se déclenche en remplissant ses ports de sortie, et le système propage ensuite les données de ses ports de sortie vers les ports d'entrée qui y sont rattachés.

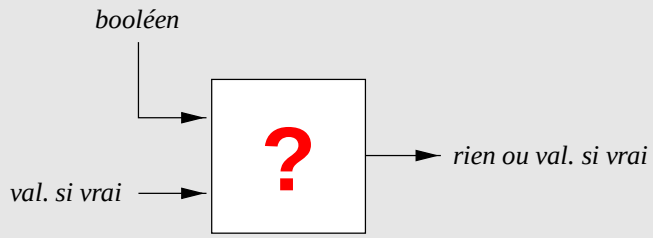
Dans Eve c'est le remplissage d'un port de sortie qui déclenche la propagation éventuelle des données. Si le composant était en train de se déclencher alors la propagation se fera normalement une fois le déclenchement fini. Mais dans tous les autres cas, si le port de sortie était le dernier port à remplir il y a propagation des données. Ceci signifie qu'un composant n'est pas obligé de remplir tous ses ports de sortie, ce qui permet par exemple de réaliser des composants réalisant des structures de contrôles conditionnelles (comme le « si ... alors ... » ou le « si ... alors ... sinon ... » présentés figure 1.6 page ci-contre). De même un composant peut émettre des données à tout moment, relançant la propagation de données (c'est ainsi que fonctionnent les composants sans ports d'entrée, par exemple : ils ne sont jamais prêts à être déclenchés car ils ne reçoivent pas de données, mais ils peuvent en produire à leur rythme).

Graphe de programmation et exécution

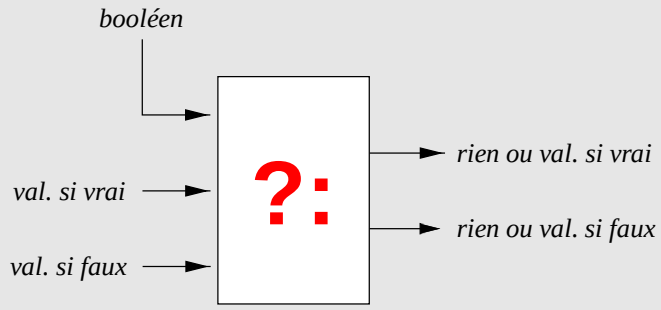
Le graphe de programmation — c'est-à-dire le programme visuel — gère un certain nombre de tâches indispensables à sa bonne exécution en plus des opérations simples consistant à autoriser l'exécution ou à la suspendre, à réinitialiser l'ensemble des composants et à gérer l'ajout et la suppression de composants dans le programme.

Un de ses rôles est de considérer les composants pouvant s'exécuter dans l'ordre induit par un tri topologique[2] de lui-même, afin de garantir le respect des précédences liées aux connections entre composants. Il est essentiel d'exécuter les composants dans cet ordre sous peine de s'exposer à des erreurs d'exécution telles que le blocage de certains composants ou l'impossibilité d'exécuter une partie du

2. Si le graphe comporte des circuits, le tri topologique ne modifie pas ces circuits, mais garantit que tous les composants en amont du circuit seront effectivement exécutés avant l'entrée dans le circuit.



(a) Si ... alors ...



(b) Si ... alors ... sinon ...

FIG. 1.6. – *Exemple de composants réalisant des structures de contrôles. On notera que la forme présentée en (b) permet non seulement d'interdire le déclenchement de la partie du programme qui est connectée au port de sortie ne propageant pas de données mais aussi de choisir une valeur parmi deux et de la propager, simplement en reliant les deux sorties du composant à un même port d'entrée de données..*

graphe de programmation, provoquant ainsi la perte de données ou la famine de sous-graphes.

Il se charge également d'éliminer toute récursion terminale dans les déclenchements de composants même si le graphe d'exécution comporte des boucles. Cette tâche est indispensable pour permettre l'exécution de tels graphes sans saturation de la mémoire du système due aux imbrications d'appels aux fonctions de manipulation des données des composants.

Enfin il gère la substitution de composants. En effet en Eve un composant peut décider de déléguer sa tâche à un autre composant. Cette substitution est faite au moment où l'on considère la possibilité de déclencher un composant : la méthode des composants qui teste cette possibilité renvoie en cas de succès le composant qui sera déclenché, ce composant n'étant pas forcément le même que celui dont la méthode a été appelée. Deux exemples où cette facilité est utile sont : la résistance aux pannes, où un composant incapable d'effectuer une tâche peut décider de se faire remplacer par un autre composant ; et le « courtage » de composants, où un composant reçoit les données et délègue sa tâche à un autre composant.

Interface de construction des programmes

La construction des programmes se fait graphiquement en déposant des « paquets » (*bundles*) sur un plan de travail représentant l'espace du programme puis en connectant les composants présents de manière à construire le graphe de flot de données. Toute partie d'un programme (c'est-à-dire un ensemble de composants et les connexions les reliant) peut être copiée puis collée dans un plan de travail.

Les paquets peuvent être de plusieurs types : soit ils contiennent des classes dont le programme visuel a besoin, soit ils contiennent exactement un composant mais aussi si besoin est son interface et même d'autres classes dont le composant se sert. Si un paquet ne contient pas de composant alors rien n'apparaît sur le plan de travail ; sinon, l'interface crée une représentation graphique du composant et la place sur le plan de travail sur lequel elle pourra être manipulée.

Les composants eux-mêmes sont écrits en Objective C avec la bibliothèque de programmation Eve. Lorsqu'Eve est intégré dans un environnement, comme c'est le cas pour Scope, cet environnement fournit une bibliothèque de composants prêts à l'emploi.

Lorsqu'un paquet est déposé sur le plan de travail, Eve s'occupe de charger dynamiquement toutes les classes (que ce soient des classes de composants ou d'objets dont ils se servent) dont celui-ci dépend, sans intervention de l'utilisateur, facilitant ainsi le découpage des outils en objets coopérant et évitant la production d'un ensemble de composants de taille importante comportant plusieurs classes communes [3].

3 . Dans les environnements que nous connaissons c'est la seule possibilité si l'on souhaite éviter de forcer l'utilisateur à déposer au préalable des paquets contenant toutes les classes dont un com-

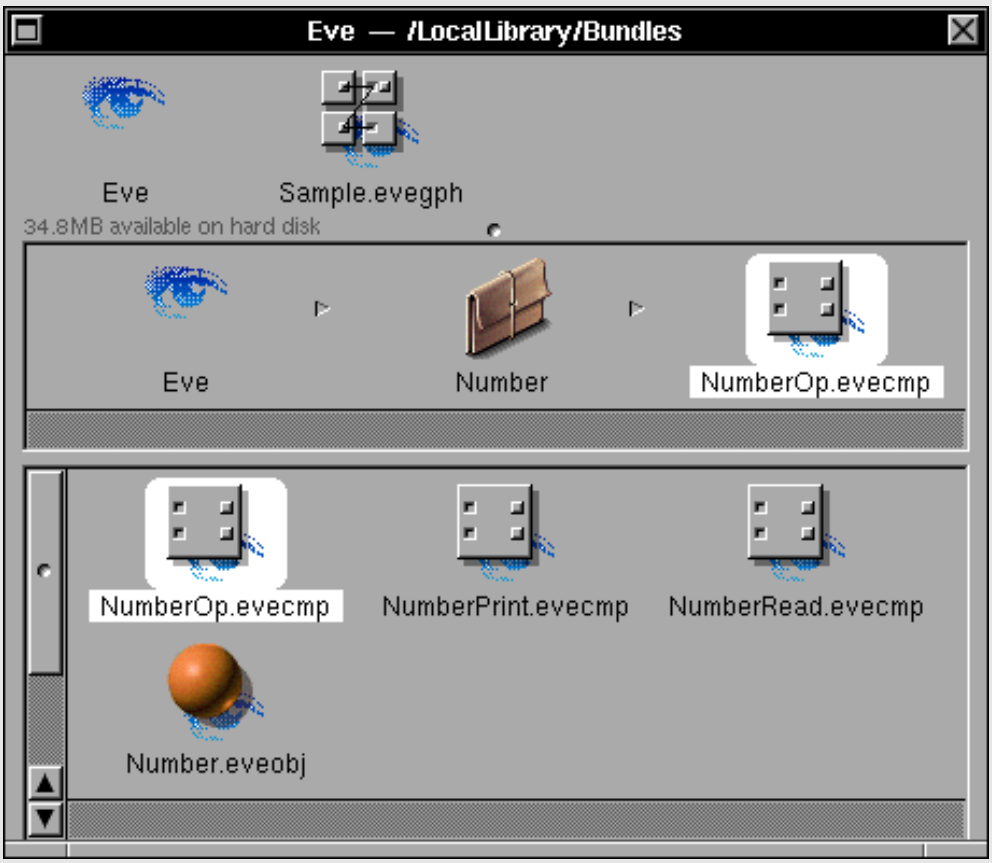


FIG. 1.7. – *On voit sur cette figure différents objets manipulés par Eve. Sur l'étagère (tout en haut) on trouve un programme visuel complet. Dans le répertoire on distingue des paquets de composants et un « objet Eve » qui est en fait un ensemble de classes ; il est automatiquement chargé lorsqu'Eve charge un composant qui en a besoin..*

posant a besoin, à moins que ces classes ne soient déjà liées à l'environnement (mais alors celui-ci n'est plus généraliste). Outre la perte d'espace disque et la complexité de la maintenance de ces paquets due à la nécessité de regénérer chaque paquet contenant une classe lors de la modification de celle-ci, cette méthode donne lieu à des messages d'erreur lors de l'édition de liens dynamique ; si ces messages sont donnés à l'utilisateur, il peut avoir du mal à en identifier la cause, ce qui est gênant, mais si les messages d'erreur lui sont toujours cachés alors il ne pourra pas diagnostiquer les problèmes des composants qu'il développera lui-même, ce qui est encore plus déplorable.

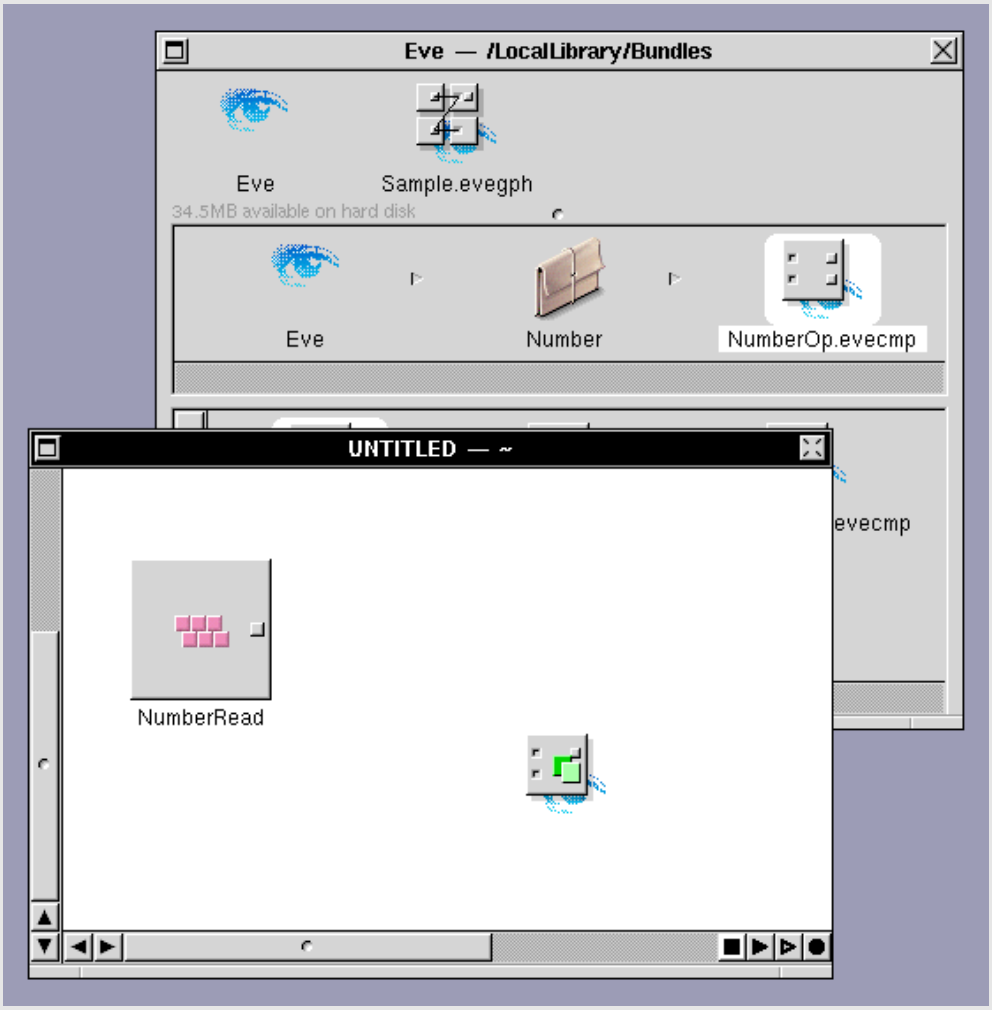


FIG. 1.8. – *On ajoute un composant dans un programme visuel en le faisant glisser sur le plan de travail représentant le programme et en l'y lâchant..*

Les composants qui le supportent peuvent être inspectés pour modifier leur configuration (c'est-à-dire les paramètres de la fonction qu'ils réalisent, par exemple). Les inspecteurs sont fournis par les composants[4] et sont également chargés

4. C'est pour cela que l'on manipule des paquets. Ceux-ci peuvent contenir autant de classes que voulu et seule la première de celles-ci — qui est le composant — est liée à l'application lors du chargement du paquet. Elle peut ensuite demander que d'autres classes, comme les inspecteurs

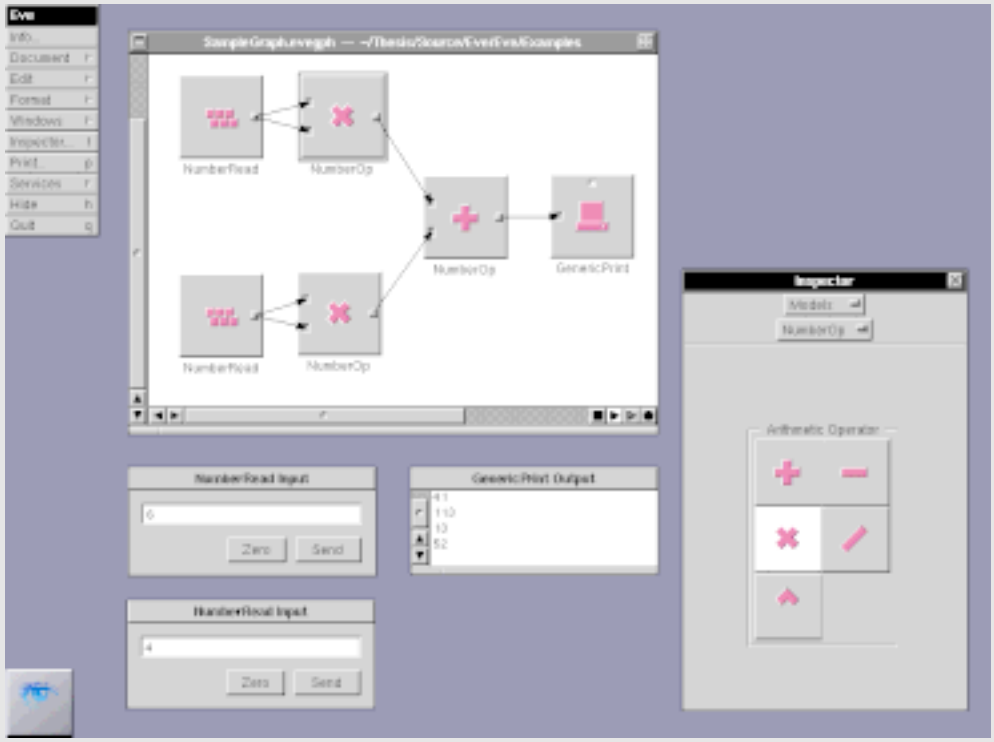


FIG. 1.9. – *Exemple de session de travail avec Eve. Les composants sont sur le plan de travail nommé « UNTITLED » et le graphe est complètement construit (tous les composants sont connectés). Les petits panneaux constituent l'interface fournie par les composants lors de leur exécution. Le composant NumberOp du haut est en train d'être inspecté, son inspecteur (à droite) permettant de choisir un opérateur arithméthique binaire. On remarque à droite de l'ascenseur horizontal les boutons de contrôle d'exécution du programme visuel, permettant l'arrêt, l'exécution et la réinitialisation de ce dernier.*

dynamiquement au fur et à mesure des besoins de l'utilisateur. L'environnement gère l'interface nécessaire à la sélection de l'inspecteur approprié au(x) composant(s) à inspecter et à sa présentation à l'utilisateur, l'inspecteur se chargeant lui-même de répercuter les effets de l'inspection, c'est-à-dire les modifications éventuelles de sa configuration, sur le composant (figure 1.9).

par exemple, soient liées lorsqu'elle en a besoin.

Les programmes peuvent être sauvegardés puis rechargés sur différents types de machines sans se soucier de leur architecture. Les programmes sauvegardés représentent uniquement l'état des divers objets composant le graphe ainsi que les connexions entre ces objets. Plutôt que les composants complets l'environnement sauve une identification (nom et chemin éventuel) de ces derniers. Les programmes sauvés sont ainsi bien plus compact mais en plus il est possible d'améliorer un composant sans avoir à toucher aux programmes qui l'utilisent : lorsqu'ils seront rechargés ils utiliseront automatiquement le composant amélioré.

L'interface graphique de programmation visuelle est développée au-dessus d'une bibliothèque de manipulation de diagrammes développée à l'*University of Houston*, DiagramKit (GLOVER 1994).

1.4 Composants spécifiques

Nous appelons composants spécifiques les objets qui ont besoin de connaître la syntaxe ou la sémantique de la trace. Les composants lisant la trace pour en extraire les événements ou ceux simulant les changements d'états à partir de ces événements sont des exemples évidents (et nécessaires) de tels composants. Ils sont décrits au chapitre 2 page 87.

Il bien entendu possible d'écrire d'autres composants spécifiques exploitant par exemple directement les événements produits par le lecteur de trace. Mais à moins de ne pouvoir agir autrement il faut éviter de le faire car ces composants ne seront pas réutilisables pour d'autres types de traces ou d'autres modèles de programmation.

1.5 Composants génériques

Les composants génériques, par opposition aux composants spécifiques, sont ceux qui n'ont pas besoin de manipuler directement la trace en en connaissant la syntaxe et la sémantique. Au lieu de cela ils sont spécialisés dans la réalisation d'un travail sur des données dont ils fixent le type mais pas la sémantique, celle-ci étant connue uniquement de l'utilisateur.

Nous présentons ici les deux grandes catégories de composants qui sont utilisées dans Scope et leurs particularités.

1.5.1 Analyse des données

Les composants d'analyse de données et de statistiques n'ont pas d'interaction avec l'environnement Scope ou avec l'utilisateur (à part lorsqu'ils sont inspectés, mais l'interaction n'est pas à proprement parler de leur fait). Seule importe la manière dont ils manipulent les données qui leur sont fournies et l'efficacité avec laquelle ils font ces manipulations.

Ils n'ont donc pas de contraintes particulières à respecter du fait de leur intégration dans Scope. Ils sont de plus totalement indépendant de NEXTSTEP et sont donc potentiellement exécutable sur une machine quelconque.

1.5.2 Visualisation

Les composants de visualisation sont plus complexes et permettent une interaction avec l'utilisateur. Afin d'homonogénéiser cette interaction les visualisations de Scope doivent être capables d'être à même de réaliser un certain nombre d'actions, et surtout deux composants réalisant une même action doivent obligatoirement le faire de façon identique, c'est-à-dire que l'utilisateur n'a pas à apprendre plusieurs méthodes pour réaliser une action donnée.

Les actions que doivent supporter les visualisations concernent la manipulation des informations présentées et spécifient la manière dont une visualisation doit conduire un type d'interaction donné avec l'utilisateur.

Toutes les interactions avec l'utilisateur se font par *manipulation directe* c'est-à-dire par l'application d'un *outil d'interaction* sur la visualisation présentée. Lorsqu'un outil est appliqué sur une visualisation, celle-ci détermine la portée de l'interaction, *i.e.* les objets concernés par celle-ci, puis mène l'interaction jusqu'à ce que l'outil soit « reposé » par l'utilisateur. La réponse de la visualisation à la manipulation dépend de l'outil utilisé : certains outils ont un effet immédiat alors que d'autres doivent être reposés pour qu'une action soit effectuée. Des exemples d'outils classiques sont ceux permettant l'inspection d'une visualisation (obten-

tion d'informations textuelles sur les objets ou les données représentées) ou la modification de la présentation des données (remaniement spatial de la visualisation).

Dans la mesure du possible, les visualisations se partagent des outils afin de minimiser l'apprentissage de leur manipulation (et incidemment de leur coût de développement et de leur occupation mémoire). Dans la « boîte à outils Scope » se trouve par exemple une caméra virtuelle permettant d'agir sur les visualisations tri-dimensionnelles (zoom, rotation et translation du point de vue de l'utilisateur, choix de la qualité de rendu, etc.).

Par principe toutes les visualisations proposées doivent supporter une forme de manipulation spatiale ainsi que l'inspection des données présentées. Elles sont libres d'ajouter tout outil de manipulation directe supplémentaire qui peut aider à leur exploitation.

1.6 Contrôle des sessions

Nous appelons *contrôle de session* l'ensemble des mécanismes s'occupant de l'interaction entre Scope lui-même (et non pas les visualisations de Scope) et l'utilisateur, *i.e.* le contrôle d'actions telles que le choix d'un format de trace, le déplacement dans le temps de l'exécution simulée, l'extraction des informations d'une trace d'exécution et l'annotation des traces.

1.6.1 Choix du modèle de programmation

Scope est un environnement générique avec lequel il est possible d'analyser différents types de traces, chacun pouvant correspondre à un modèle de programmation différent. Le choix d'un format de trace conditionne ainsi le choix des composants spécifiques à ce format. De même qu'il traite les traces de manière uniforme en utilisant une interface déterminée, Scope possède un certain nombre de conventions permettant d'installer les composants nécessaires au traitement d'une trace donnée.

Afin de pouvoir charger une trace, Scope demande à ce que celle-ci se présente sous forme de *trace Scope, i.e.* comme un répertoire nommé `nom.trace`

FIG. 1.10. – *Une trace Scope est un répertoire contenant d'une part la trace à utiliser mais aussi d'autres informations comme la documentation de cette trace ou encore des données à destination des composants l'utilisant, comme par exemple un placement par défaut des nœuds du système pour une représentation graphique.*

et contenant diverses informations (figure 1.10). La première de ces informations correspond à la trace qui se présente également sous la forme d'un répertoire. Celui-ci est nommé *nom.format* où *format* est un nom indiquant quel est le format de trace utilisé. Ce répertoire contient les données de la trace ; ces données ne sont pas manipulées par Scope mais laissées au lecteur de traces approprié. On peut éventuellement trouver dans la trace Scope une documentation multimédia qui peut être consultée depuis l'environnement. Enfin il peut y avoir diverses informations destinées à des composants Scope : un exemple classique est la description du placement initial dans l'espace des objets graphiques représentant les tâches d'une application dans une visualisation tri-dimensionnelle.

Scope associe un programme visuel à chaque type de trace, basé sur le nom de

format présent dans la trace Scope. Ainsi lorsque l'utilisateur demande à ouvrir une trace, Scope charge simplement ce programme visuel, lequel est adapté à la manipulation de la trace choisie.

Par défaut le programme visuel correspondant à un type de trace donné comprend simplement un panneau de contrôle du déroulement de la session d'évaluation ainsi qu'un lecteur de traces adapté contrôlé depuis de ce panneau et connecté à un simulateur dédié au modèle de programmation correspondant à la trace (figure 1.11 page ci-contre). De ces trois composants, deux sont nécessaires (le panneau de contrôle et le lecteur de trace) et bien que le troisième (le simulateur) ne le soit pas il est utilisé dans la majorité des sessions d'évaluation de performance.

Mais il est également possible de disposer de plusieurs programmes de complexité arbitraire : Scope demande alors à l'utilisateur de choisir quel programme il souhaite utiliser. Il est ainsi possible de mettre à la disposition de l'utilisateur des schémas d'évaluation de performance adaptés à différentes analyses.

1.6.2 Déroulement de la session d'évaluation

Une session d'évaluation de performance consiste à lire une trace et à analyser les informations obtenues à partir de celle-ci. Le contrôle de cette session consiste donc à contrôler l'extraction des événements de la trace et leur traitement par le programme visuel décrivant le schéma d'analyse employé pour l'évaluation de performance.

Le panneau de contrôle de Scope (figure 1.12 page 84) permet de gérer d'une part le défilement du temps au cours de l'analyse et d'autre part le déplacement vers une date arbitraire de l'exécution du système évalué.

Défilement du temps

Le défilement du temps peut se faire de plusieurs manières : en continu ou pas à pas. Lorsque le défilement est continu, Scope ne s'arrête de lire les événements qu'à la fin de la trace. Quand il se fait en pas à pas, il ne lit que quelques événements avant de s'arrêter.

Nous distinguons deux types de pas à pas : le pas à pas par événement consiste

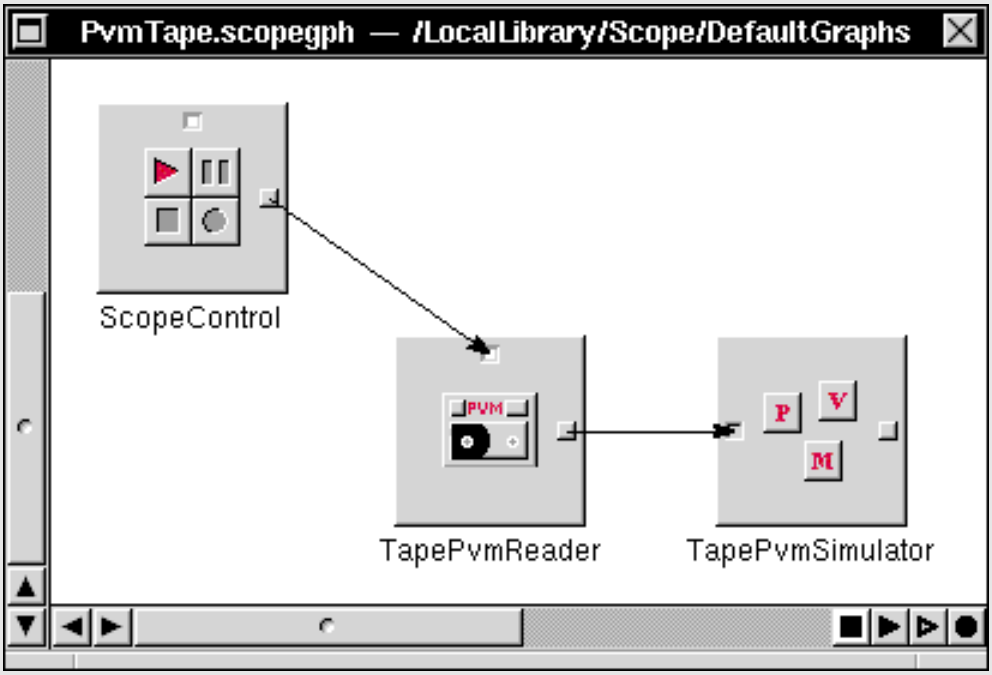


FIG. 1.11. – *Programme visuel associé à un format de trace (ici les traces produites par le traceur logiciel* TAPE/PVM*). On notera la manière dont les composants sont connectés : le panneau de contrôle pilote le lecteur de traces en lui envoyant des messages lui demandant de se déplacer dans la trace, et le lecteur transmet les événements lus au simulateur.*

à lire un seul événement puis à s'arrêter alors que le pas à pas par date lit des événements tant que leurs dates sont identiques. Le premier permet de s'intéresser très précisément à chaque action de l'application tracée alors que le second pemet de voir les changements en fonction du temps, les actions simultanées étant effectivement traitées en même temps par l'environnement.

Déplacement dans le temps

Scope est conçu de manière à permettre le déplacement vers un instant quelconque de l'éxécution de l'application tracée. Ce mécanisme est basé sur une sauvegarde de l'état de l'environnement à certains instants et est expliqué au cha-

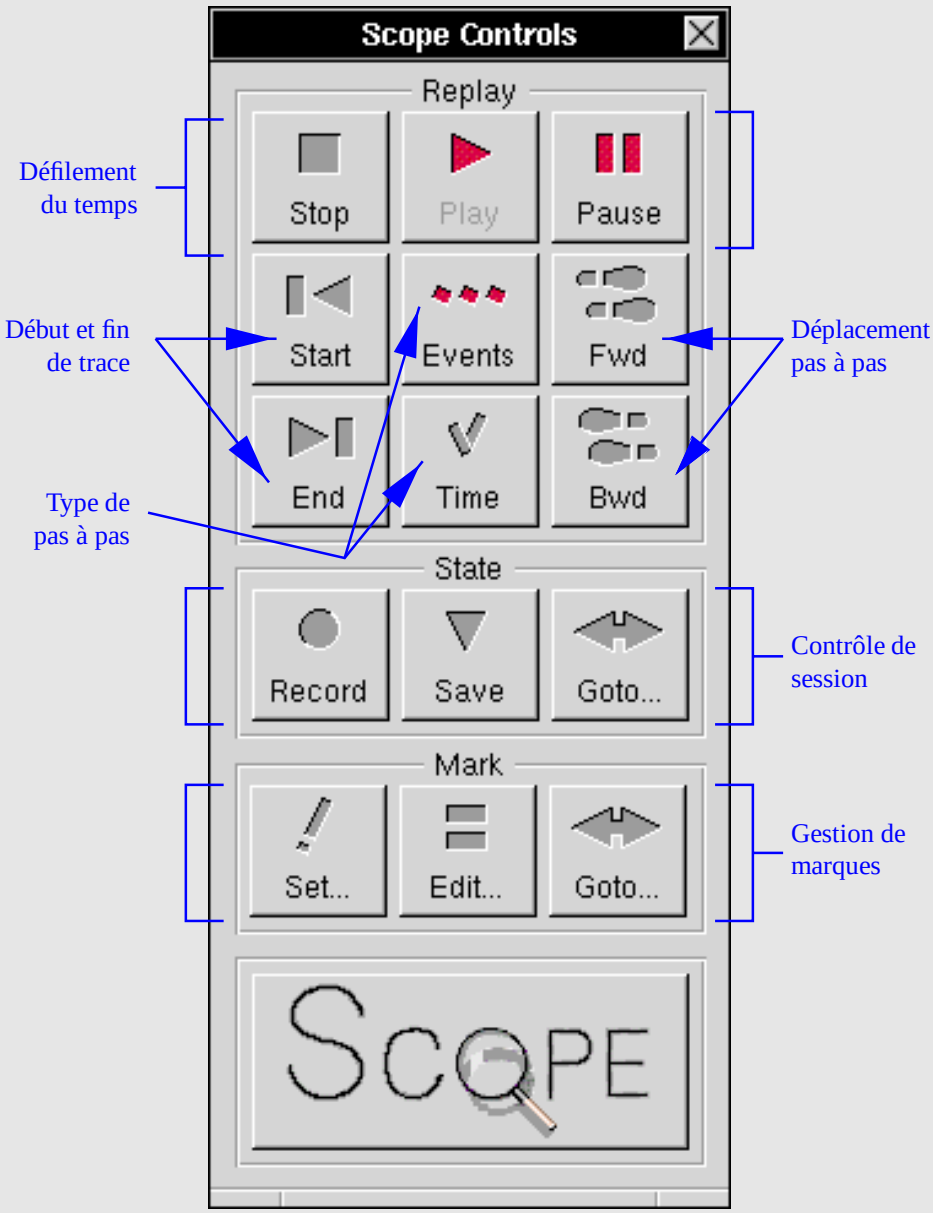


FIG. 1.12. – *Le panneau de contrôle de session de Scope permet de contrôler le défilement du temps (*i.e. *l'extraction des événements de la trace) ainsi que le mécanisme de voyage de temps intégrés à l'environnement.*

pitre 3 page 131.

Le contrôle du déplacement dans le temps permet de déterminer quels sont les instants auxquels l'état de l'environnement est sauvé, de forcer cette sauvegarde et de choisir un de ces instants pour s'y déplacer. Scope gère également des marques qui permettent d'associer un nom à un instant (que celui-ci corresponde à une sauvegarde ou non) afin de pouvoir y revenir facilement.

Autres capacités

Le composant de contrôle de session est doté d'un port de contrôle qui lui permet de recevoir des messages de commande de la part des autres composants du programme visuel. Par ce biais, les composants sont capables de contrôler le déroulement de la trace et le déplacement dans le temps. Il est alors envisageable d'écrire des composants « intelligents » dont le rôle est d'attendre l'apparition d'un problème de performance et de stopper l'exécution à l'instant voulu afin de laisser à l'utilisateur le loisir d'examiner le problème.

1.7 Conclusion

Scope est organisé suivant une architecture orientée objet. Les différents types d'objets nécessaires à la conduite d'une session d'évaluation de performances ont été identifiés et classés en outils génériques ou spécifiques à un modèle de programmation. Les premiers sont réutilisables et n'entraînent aucun besoin de réécriture lors de l'ajout du support d'un nouveau modèle de programmation ; la spécificité des seconds est cachée à l'utilisateur et l'environnement sait choisir et mettre en place les outils spécifiques nécessaires au traitement d'un modèle de programmation donnée. Scope comprend un environnement de programmation visuelle intégré, Eve, qui permet de construire dynamiquement des schémas d'analyse et de visualisation. Eve est le cœur de Scope en ce sens que tous les outils présents dans l'environnement sont des composants Eve. Il fournit également un puissant mécanisme de personnalisation et d'extension de l'environnement puisqu'il est possible de modifier les schémas d'analyse de Scope et d'ajouter des outils sans changer l'environnement lui-même. Scope est manipulé à travers un mécanisme de contrôle de session qui permet d'obtenir automatiquement un pro-

gramme d'analyse adapté à un modèle de programmation donné lors du chargement d'une trace, puis de gérer le déplacement dans cette trace. Le composant de contrôle de session de Scope peut recevoir des commandes des composants du programme visuel, cette capacité autorise la création de composants « intelligents » prenant le contrôle de l'environnement pour l'utilisateur.

2

Lecture de traces et simulation

AVANT de pouvoir visualiser tout ou partie d'une exécution pour évaluer ses performances, il faut être capable d'obtenir suffisamment d'informations sur cette exécution. L'information de plus bas niveau disponible dans le contexte qui nous intéresse est constituée par la succession des événements produits par une application durant son exécution ; la récupération de ces événements est le rôle de la lecture de trace. Les événements décrivent les changements d'états successifs dus aux actions de l'application (par exemple, création d'une nouvelle tâche ou échange de données entre différentes tâches). Afin de reconstruire les états dans lesquels s'est trouvé le système, il faut simuler les effets de ces événements, en connaissant l'incidence de la production d'un événement alors que le système est dans un état donné : c'est le rôle des simulateurs.

À partir des événements de la trace et des états de l'application il est possible de construire des visualisations adaptées à la représentation des performances de l'application au cours du temps [1].

La lecture de traces et la simulation représentent deux parties très importantes d'un environnement d'évaluation de performances : sans la première il est impossible d'obtenir quelque information que ce soit sur l'exécution d'une application et

1. Il faut noter qu'il est parfaitement possible de construire des visualisations n'utilisant pas les états de l'application mais seulement les événements de la trace, comme par exemple une représentation du volume de données transmis par chaque tâche au cours de son exécution.

la seconde est indispensable à la reconstitution des états de l'application au cours de l'exécution, seule cette reconstruction permettant de répondre à des questions du type « Combien de tâches restent bloquées plus de tel laps de temps ? » ou « Quelle est l'utilisation moyenne des processeurs de la machine ? ».

Ce chapitre présente tout d'abord les principes de la lecture de traces et de la simulation tels qu'ils apparaissent dans Scope (*i.e.* avec les choix de conception qui ont été faits). Une fois cette présentation effectuée nous décrivons l'application de ces principes à deux bibliothèques de programmation, l'une pour machines à base de Transputers et l'autre pour réseaux de stations de travail ou machines parallèles utilisant PVM (*Parallel Virtual Machine*, GEIST *et al.* 1994a, GEIST *et al.* 1994b). Pour chaque bibliothèque nous présentons les problèmes et les choix liés à la réalisation d'un lecteur de traces dédié et d'un simulateur adapté au modèle de programmation utilisé par la bibliothèque, Les composants de traitement du modèle de programmation de PVM font l'objet d'une évaluation de performance permettant de connaître leur efficacité et d'évaluer l'impact de nos choix de conception sur la rapidité de fonctionnement de l'environnement.

Nous conclurons en évoquant les problèmes de réutilisation et d'adaptation des composants développés lors du support de nouveaux formats de trace et de différents modèles de programmation.

2.1 Lecture de traces

La lecture de traces a pour objet de fournir les événements successifs d'une trace sous une forme utilisable pour l'évaluation des performances.

Les événements doivent donc pouvoir être manipulables par le simulateur dédié au modèle de programmation des applications tracées afin de permettre la reconstitution des états successifs de l'application. Il doit également être possible de les traiter afin d'en extraire des informations sur les actions de l'application.

2.1.1 Lecteurs de traces

Rappelons que Scope, de par sa conception, n'a aucune connaissance de la syntaxe des traces manipulées ni de leur sémantique. C'est ce qui permet à l'envi-

ronnement d'être indépendant du format de traces et donc facilement extensible de façon à supporter de nouveaux formats. Scope a néanmoins besoin de manipuler la trace afin de « cadencer » le déroulement de la session d'évaluation de performances, laquelle est basée sur l'exploitation des informations contenues dans la trace.

Pour réaliser cette tâche Scope impose un certain nombre de contraintes aux objets liés à la lecture de traces (lecteurs de traces et événements). Le respect de ces contraintes garantit d'une part que les informations nécessaires au cadencement peuvent être obtenues par Scope et d'autre part que celui-ci peut agir sur la lecture de traces. Les contraintes imposées par l'environnement sont les suivantes :

- les lecteurs de traces se chargent d'ouvrir une trace et d'en extraire les événements à la demande de l'environnement ;
- les événements comportent tous une horloge, ou date, représentant l'« instant » de leur production lors de l'exécution de l'application ;
- les horloges sont comparables entre elles de façon à établir un ordre total entre les événements [2].

Grâce à ces contraintes Scope est capable d'avancer dans une trace sans pour autant connaître la syntaxe ou la sémantique des événements manipulés : une fois une trace ouverte il est possible de se déplacer dans celle-ci vers une date donnée en en extrayant les événements et en comparant leur date à la date voulue jusqu'à ce que l'instant choisi ait été atteint.

Dans un modèle de programmation orientée objet comme celui utilisé pour Scope il existe deux manières principales de garantir qu'un objet est capable de respecter certaines contraintes : la première consiste à s'assurer que l'objet appartient à une classe héritant d'une classe abstraite déclarant les opérations nécessaires au respect de ces contraintes et la seconde réclame l'adoption de protocoles recensant un certain nombre d'opérations que l'objet est capable de réaliser.

Nous souhaitons que le coût de l'ajout de nouveaux composants dans Scope soit le plus faibe possible afin non seulement de faciliter cet ajout mais également

2. La façon dont cet ordre est établi ne concerne pas l'environnement. Si les dates sont vectorielles (LAMPORT 1978), par exemple, il est possible de définir un site de référence pour établir cet ordre total et de laisser l'utilisateur changer cette référence afin de voir différents ordres d'événements.

d'inciter à sa réalisation. Or s'il existe des bibliothèques de lecture pour les différents formats de traces existant il est irréaliste de penser que ces bibliothèques sont organisées de façon à respecter les contraintes de Scope. Il est donc nécessaire d'adopter une solution non intrusive au sens où la mise en conformité avec les contraintes de Scope est une opération minime.

Des deux possibilités présentées ci-dessus, nous avons donc choisi l'utilisation des protocoles qui offrent une grande souplesse. En effet si une bibliothèque de lecture de traces existe sous forme de classes il est aisé de leur ajouter le respect de protocoles alors que réorganiser ces classes de façon à les faire hériter de classes imposées est impensable : non seulement c'est une tâche qui peut prendre un certain temps ou même se révéler impossible mais il peut aussi arriver que la nouvelle hiérarchie ne soit plus utilisable dans les applications qui utilisaient la bibliothèque originale.

2.1.2 Protocoles de la lecture de trace

Nous présentons ici les différents protocoles utilisés par Scope pour manipuler les objets relatifs à la lecture de traces c'est-à-dire les méthodes que ces objets doivent implémenter afin d'être utilisables par Scope[3].

Protocole des lecteurs de traces

Le tableau 2.1 page 92 décrit les diverses méthodes composant le protocole TraceReader auquel adhérent les lecteurs de traces. Les méthodes de ce protole permettent d'une part l'ouverture d'une trace et d'autre part l'accès séquentiel aux événements qui la composent.

L'accès aux événements dans un sens de lecture donné n'est possible que si le lecteur annonce qu'il en est capable. Il est évidemment entendu qu'un lecteur doit au moins être capable de lire une trace vers l'avant si l'on veut pouvoir exploiter les traces... La lecture de traces vers l'arrière n'est exploitée par Scope qu'à des

3 . La présentation des méthodes a été simplifiée en supprimant les types des objets, ceux-ci étant facilement déterminables d'après les noms des méthodes. Les méthodes précédées par un signe plus sont des méthodes auxquelles répondent les classes et celles précédées par un signe moins sont celles auxquelles répondent les objets de ces classes.

fins d'optimisation et n'est donc absolument pas nécessaire à la bonne marche de l'environnement.

Protocole des événements

Le tableau 2.2 page 93 décrit les diverses méthodes du protocole TraceEvent auquel adhérent les événements manipulés par Scope. Ce protocole est extrêmement réduit puisqu'il ne comporte qu'une seule méthode destinée à fournir la date d'un événement, seule partie d'un événement qui est directement manipulée par l'environnement lui-même.

Protocole des horloges

Le tableau 2.3 page 93 décrit les diverses méthodes du protocole TraceClock auquel adhérent les horloges des événements manipulés par Scope. Ces méthodes autorisent la manipulation d'horloges et leur comparaison, sans pour autant que la syntaxe des horloges et la sémantique de leur comparaison soient connues de l'environnement.

2.1.3 Production des événements

Les traces pouvant être organisées arbitrairement il est impossible de prédire l'ordre des événements lus séquentiellement dans une trace. Le classement des événements afin de les exploiter par ordre de dates croissantes est classiquement le travail de composants tels que les simulateurs, qui gèrent un échéancier d'événements à venir.

Mais contrairement au cas de la simulation traditionnelle, dans la simulation dirigée par les traces la gestion de cet échéancier n'est pas une lourde tâche : tous les événements étant connus la gestion de l'échéancier consiste simplement en un interclassement des événements d'après leurs dates. Il est de plus souvent souhaitable de pouvoir travailler directement sur les événements et les informations qu'ils contiennent afin de manipuler ces dernières. Ceci est particulièrement facilité si les événements sont classés par dates croissantes.

Méthode	Effet
- openTraceFile:*trace*	Ouvre la trace *trace* et renvoie l'état du lecteur après ouverture sous forme de chaîne de caractères (*trace* est un répertoire dans lequel se trouvent les informations voulues)
- traceReaderStatus	Renvoie l'état du lecteur sous forme de chaîne de caractères ; si cette chaîne est non-nulle, elle correspond à un message d'erreur
- currentTraceEvent	Renvoie le dernier événement lu dans la trace ; cet événement adhère au protocole TraceEvent
- nextTraceEvent	Avance d'un événement dans la trace et renvoie l'événement courant ; cet événement adhère au protocole TraceEvent
- previousTraceEvent	Recule d'un événement dans la trace (si cela est possible) et renvoie alors l'événement courant ; cet événement adhère au protocole TraceEvent
- canStepForward	Renvoie YES si il est possible d'avancer dans la trace (*i.e.* si il reste des événements)
- canStepBackward	Renvoie YES si il est possible de reculer dans la trace (*i.e.* si le lecteur en est capable et s'il n'est pas au début de la trace)

TAB. 2.1. – *Méthodes du protocole TraceReader auquel adhérent les lecteurs de traces de Scope. Ces méthodes permettent de manipuler la trace elle-même ainsi que les événements qui la composent.*

Méthode	Effet
- clock	Renvoie l'horloge (ou date) associée à l'événement ; cette horloge adhère au protocole TraceClock
- printOnStream:*aStream*	Affiche le récepteur sur le flux *aStream*

TAB. 2.2. – *Méthodes du protocole TraceEvent auquel adhérent les événements manipulés par Scope.*

Méthode	Effet
- initWithClock:*aClock*	Initialise le récepteur avec la date contenue dans l'horloge *aClock* adhérant au protocole TraceClock
- compare:*aClock*	Renvoie le résultat de la comparaison du récepteur avec l'horloge *aClock* adhérant au protocole TraceClock, *i.e.* une valeur nulle si les dates sont identiques, une valeur strictement positive si la date du récepteur est postérieure à celle de l'horloge *aClock* et une valeur strictement négative sinon
- stringRepresentation	Renvoie une chaîne de caractères représentant le récepteur
- printOnStream:*aStream*	Affiche le récepteur sur le flux *aStream*
+ clockEditor	Renvoie un objet permettant l'édition d'une horloge (cet objet sera utilisé par l'environnement)

TAB. 2.3. – *Méthodes du protocole TraceClock auquel adhérent les horloges des événements manipulés par Scope. Ces méthodes permettent l'initialisation des horloges, leur comparaison afin d'établir un ordre total sur les dates et leur édition.*

Scope ajoute donc une contrainte aux lecteurs de traces : ceux-ci doivent fournir les événements classés. Cette contrainte a deux intérêts : elle évite aux autres composants du système d'avoir à gérer un échéancier d'interclassement, même simple, et elle permet un gain de temps lors de l'exécution de Scope puisque l'interclassement n'a lieu que lorsqu'il est vraiment nécessaire, le lecteur de trace étant le composant le mieux placé pour connaître ce besoin et y répondre.

2.1.4 Présentation des événements

Comme nous venons de le voir il peut être intéressant de travailler directement sur les informations contenues par les événements, pour calculer la moyenne des volumes de données échangées entre les différentes entités d'une application, par exemple.

Les composants de l'environnement ayant à travailler sur ces données sont des composants génériques d'analyse de données et de statistiques. En tant que tels ils ne connaissent pas la syntaxe des événements et ne peuvent donc en extraire les informations voulues. Ce serait de plus une erreur de vouloir procéder ainsi car cela signifie qu'il faudrait éventuellement une version spécialisée de chaque composant pour chaque format de trace susceptible d'être traité et que des changements de syntaxe dans un format de trace donné nécessiteraient la modification des composants d'analyse de données.

La solution proposée pour répondre à ce problème est l'utilisation de *présentateurs d'événements*. Un présentateur d'événement est un objet qui offre une liste d'attributs correspondant aux différentes informations contenues dans un événement ; lorsqu'un événement lui est donné ces attributs reflètent immédiatement ces informations. En connectant des présentateurs d'événements à un lecteur de trace il est alors possible d'obtenir les informations souhaitées et de passer ces informations à des composants d'analyse de données.

2.2 Simulation

Même s'il est possible de tirer directement parti des événements décrivant les actions d'une application, il est souvent nécessaire de connaître la succession des

états d'une application pour pouvoir mieux la comprendre et déterminer ses performances. La simulation dirigée par les traces — qui est une forme de simulation à événements discrets (LEROUDIER 1979) — est la technique permettant de reconstruire l'état du système (qui peut être l'application ou celle-ci et une partie de son environnement, comme les processeurs sur lesquels elle s'exécute) à un instant donné.

2.2.1 Principes

Le fonctionnement d'un simulateur est très simple : l'état du système simulé est représenté sous forme d'objets modélisant les entités du système réel et le simulateur se charge de faire évoluer cet état en fonction des événements représentant les actions du système. Pour chaque événement, le simulateur exécute une fonction de transition permettant de passer de l'état antérieur à cet événement à l'état postérieur à celui-ci (figure 2.1).

L'état du système est la composition des états de tous les objets représentant le système. L'état d'un objet est défini par les valeurs d'un certain nombre de variables d'état modélisant l'objet réel.

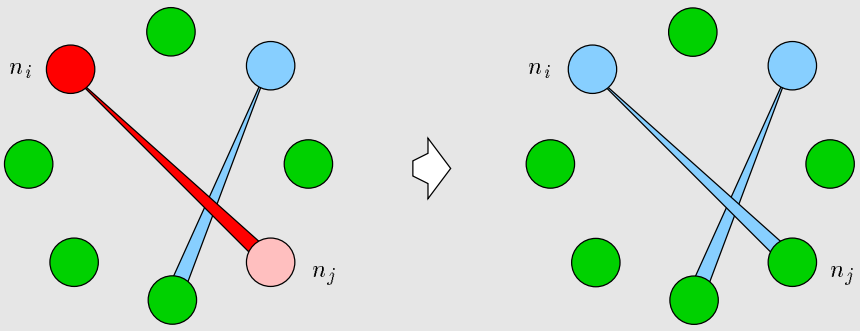


FIG. 2.1. – *Principe de la simulation : chaque événement provoque la mise à jour d'un certain nombre d'objets représentant l'état du système. Dans cet exemple, la simulation d'un événement du style « n_j* finit un envoi vers *n_i » provoque la mise à jour des objets représentant n_i, n_j et le lien de communication les reliant.*

Le problème le plus important lors de la réalisation d'un simulateur pour un modèle de programmation n'est pas tant de programmer la fonction de transition, aussi complexe soit-elle, que d'identifier correctement les différents objets à simuler ainsi que les états dans lesquels chacun d'entre eux peut se trouver. C'est de la qualité de cette analyse que dépend l'exactitude des informations fournies par le simulateur et donc la précision des indices de performances et des visualisations qui peuvent en être tirés.

Simulateurs

Les simulateurs disponibles dans Scope associent des attributs à chacun des objets du système. Ces attributs représentent non seulement les variables d'états servant à la modélisation des divers objets mais également d'autres informations utiles telles que la date de la dernière mise à jour d'un objet ou encore des indices simples tels que le volume total de données ayant circulé sur un lien de communication ou des indications des différents pourcentages de temps passé par l'objet dans un état donné[4]. L'idée est que les simulateurs doivent pouvoir répondre à tout instant à des questions de performances sur les objets qu'ils simulent.

Nous avons dit précédemment que les objets manipulés par les simulateurs représentent les objets réels. Dans le cas des applications parallèles ces objets peuvent être des tâches, des fils d'exécution, des messages, des sémaphores, mais aussi des processeurs, des liens de communication, etc. On remarque que les relations entre les différents objets sont toujours des relations de dépendances qui peuvent être représentées sous forme de graphe. Deux exemples mettant en jeu des entités différentes en sont l'échange de données (dépendance entre l'entité recevant les données et celle qui les fournit) et la prise d'un sémaphore (dépendance de l'entité par rapport au sémaphore). Dans ce qui suit nous appellerons indifféremment *nœud* une tâche, un fil d'exécution ou un processeur.

Les sessions d'évaluation des performances d'une application nécessitant la plupart du temps l'utilisation d'un simulateur, il est évident que les performances de ce dernier conditionnent l'agrément — si ce n'est la faisabilité — de ces sessions. Il est donc extrêmement important que les simulateurs soient efficaces pour

4. Le simulateur maintient en fait des cumuls de temps mais les présente sous formes de pourcentages car c'est la forme la plus parlante pour l'utilisateur. Ces calculs pourraient être faits par d'autres composants du système mais le simulateur est bien placé pour les effectuer et ces opérations n'influent pas sur ses performances, aussi n'est-ce pas gênant de les calculer.

que l'environnement soit utilisable.

Les performances d'un simulateur dépendent d'une part de la rapidité de sa fonction de transition et d'autre part de la vitesse à laquelle celui-ci est à même d'effectuer une mise à jour de l'état du système. La mise à jour de l'état du système est faite en modifiant les attributs des objets concernés par un événement donné ; si cette mise à jour a la plupart du temps un coût constant, l'accès à l'objet à mettre à jour, lui, est directement lié au nombre d'objets du système simulé. Il faut donc optimiser le temps d'accès à un objet simulé afin de garantir de bonnes performances du simulateur même lorsque le système simulé comporte un grand nombre d'objets (ce qui peut être le cas pour de grandes applications parallèles). Étant donné qu'un système parallèle est représenté par un graphe dont les sommets sont les nœuds et les arêtes les liens de communication, les simulateurs ont besoin de gérer efficacement des graphes en étant capable de les parcourir très rapidement.

Pour traiter ce problème, nous avons choisi de nous servir d'une bibliothèque de manipulation de dictionnaires et de graphes développée par des chercheurs des laboratoires d'AT&T (NORTH et VO 1993). L'intérêt immédiat de ce choix est qu'il n'a pas été nécessaire d'écrire des fonctions de manipulation de graphes. Mais ces bibliothèques sont également très performantes de par leur utilisation de structures de données adaptatives ; leur usage n'entraîne donc pas de perte de performances par rapport à un développement *ad hoc*, et est même certainement plus efficace étant donné le soin apporté à leur réalisation. De plus ces bibliothèques sont assez simples d'emploi ce qui signifie qu'il est facile de manipuler les structures de données d'un simulateur afin de l'étendre ou de l'adapter à certains besoins.

Cette bibliothèque de manipulation de graphes se révèle de plus être particulièrement adaptée à la façon dont fonctionnent les simulateurs de Scope puisqu'elle permet d'associer aux différents objets d'un graphe — c'est-à-dire le graphe lui-même ainsi que les sommets et arêtes le composant — des attributs sous forme de chaînes de caractères associées à un nom. Les simulateurs utilisent ces attributs pour exporter l'état du système sous une forme facile à manipuler : si l'on connaît le nom d'un attribut et sa sémantique il est possible de l'utiliser pour construire une visualisation ayant un sens pour l'utilisateur sans pour autant qu'il soit nécessaire de donner à l'environnement une connaissance quelconque de ce sens. En outre, les versions futures de cette bibliothèque supporteront la spécification dynamique de sous-graphes, ceux-ci pouvant alors être utilisés par les simulateurs

pour représenter des états composés (par exemple l'état d'un processeur comme la composition des états des tâches y étant exécutées). Une telle possibilité permet d'envisager la construction dynamique de visualisations présentant une hiérarchie d'informations et supportant l'affichage de ces informations sous forme exhaustive (mise à plat du graphe) ou agrégée.

Enfin, deux autres points confirment l'intérêt d'utiliser cette bibliothèque de manipulation de graphes. En premier lieu il est également possible d'augmenter les structures de données associées au graphe de façon dynamique, possibilité qui est utilisée par les simulateurs pour accéder à moindre coût aux variables d'états des objets simulés, mais qui permet aussi à des objets manipulant un graphe de maintenir des informations sur ces composants (c'est ainsi par exemple que l'on peut associer une position dans l'espace aux sommets d'un graphe, position qui sera ensuite utilisée pour représenter ce graphe). En second lieu il faut savoir qu'il existe un certain nombre d'outils de manipulation des graphes gérés par cette bibliothèque[5] (KOUTSOFIOS et NORTH 1992) et qu'il est donc possible de travailler sur la représentation de l'état du système simulé en dehors de l'environnement d'évaluation de performances.

2.3 Support des applications sur Transputer

Notre première réalisation en matière de lecteur de traces et de simulateur dédié concerne le support d'applications écrites pour la machine parallèle Meganode de Telmat (ARROUYE *et al.* 1993), en C avec l'aide d'OUF/TRITA (ARROUYE 1993), une bibliothèque de programmation parallèle et de communication développée localement.

Le Meganode est une machine à base de Transputer. Ces processeurs peuvent être multiprogrammés et disposent chacun de quatre canaux de communication bidirectionnels. La topologie physique du Meganode peut être reconfigurée tant qu'elle correspond à un graphe de degré quatre au plus.

Les applications OUF/TRITA peuvent être de type MIMD ou SPMD et sont constituées par des tâches elles-mêmes éventuellement composées de multiples fils d'exécution. Le modèle d'envoi de messages est celui d'Occam, *i.e.* celui

5. Il est possible de sauver un graphe sous forme textuelle, avec tous ses attributs, et de relire ensuite une telle description pour reconstituer un graphe.

de CSP (messages bloquants synchrones, cf. § A.1.1 page 232). Pour pouvoir communiquer les tâches doivent déclarer vers qui elles veulent envoyer des données (liens monodirectionnels) et ces déclarations détermineront quels canaux de communication seront utilisés entre les processeurs. Outre l'envoi de message, OUF/TRITA supporte la synchronisation de fils par des sémaphores.

Les sections qui suivent donnent les caractéristiques des traces disponibles et la description de l'architecture du lecteur de traces et du simulateur développés en appliquant les principes présentés précédemment.

2.3.1 Prise et format de trace

Prise de trace

Les applications écrites avec OUF/TRITA peuvent être tracées à l'aide du traceur logiciel TAPE (ARROUYE 1992). Celui-ci est constitué par une version instrumentée de la bibliothèque de programmation OUF/TRITA et par une tâche de collecte de trace à laquelle chaque tâche utilisateur est connectée (figure 2.2 page suivante).

Les fonctions instrumentées produisent des événements qui sont stockés localement dans l'espace d'adressage des tâches afin de minimiser le coût de la prise de trace en évitant toute communication pendant cette prise. Lorsqu'un tampon d'événements est plein, ou en fin d'exécution, les événements sont transmis à la tâche de collecte qui les écrit dans le fichier de trace. La tâche de collecte de trace a également la responsabilité du démarrage et de la terminaison de l'application tracée.

Les horloges des différents processeurs de la machine sur laquelle s'exécute l'application n'étant pas forcément identiques le traceur commence par les synchroniser, *i.e.* par calculer les différences des dates données par chacune d'entre elles. Cette synchronisation associée à des données expérimentales indiquant la dérive de chaque horloge au cours du temps est utilisée pour dater les événements avec suffisamment de précision pour éviter des incohérences dans la trace [6]

6. Un exemple classique d'incohérence est le cas où les événements de la trace indiquent que la date de réception d'un message est antérieure à sa date d'émission ; dans ce cas il est impossible de savoir ce qui s'est passé dans le système et la trace a de fortes chances d'être inutilisable pour

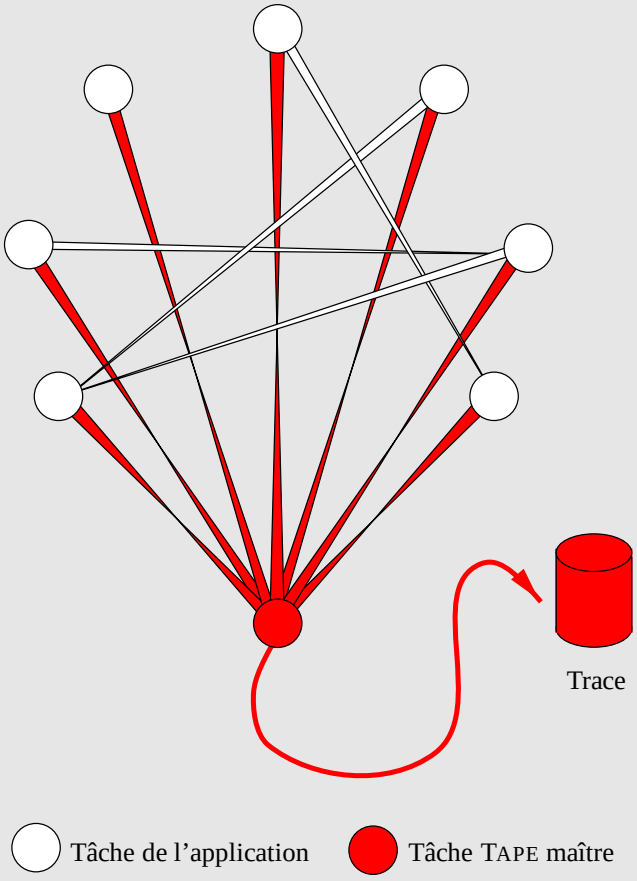


FIG. 2.2. – *Organisation des tâches d'une application tracée avec* TAPE. *Chaque tâche de l'application est connectée à une tâche maître qui récupère ses événements pour les écrire dans le fichier de trace et se charge du démarrage et de la terminaison de l'application.*

(MAILLET 1994, MAILLET et TRON 1995). On obtient alors une référence de temps globale (JÉZÉQUEL 1990) pour tous les processeurs.

Il n'est pas nécessaire de modifier le source d'une application pour produire des traces : lorsqu'on souhaite obtenir une trace, il suffit de refaire l'édition de liens de l'application avec la bibliothèque OUF/TRITA instrumentée pour que

faire de l'évaluation des performances.

l'application produise une trace. L'utilisateur peut également spécifier dynamiquement quels événements doivent être tracés au cours de l'exécution et produire ses propres événements (à charge pour lui ensuite de fournir le code nécessaire à leur interprétation lorsqu'ils sont exploités à des fins d'analyse).

Format de trace

Une trace TAPE est constituée d'un ensemble de fichiers décrivant d'une part la structure de l'application et d'autre part les événements produits au cours de l'exécution ; on appellera cet ensemble « fichier de trace » par abus de langage. Les différents fichiers constituant la trace sont (figure 2.3 page suivante) :

- un fichier de topologie décrivant l'architecture physique de la machine ainsi que l'architecture logique de l'application ;
- un fichier d'index indiquant pour chaque tâche le début d'un bloc d'événements lui appartenant dans le fichier d'événements ;
- un fichier d'événements organisé par blocs : chaque bloc correspond aux événements d'une tâche, les événements d'un bloc donné étant ordonnés par dates croissantes et doublement chaînés afin de permettre une lecture bidirectionnelle de la trace.

Alors que le fichier de topologie est textuel, les fichiers d'index et d'événements sont des fichiers binaires afin de minimiser leur temps d'accès et l'espace nécessaire à la conservation des traces.

Les événements ont un format variable suivant les informations associées à l'action qu'ils représentent. Tout événement comporte cependant un en-tête donnant des informations communes à tous les types d'événements, à savoir le type de l'événement, sa date, l'identification de l'entité qui l'a produit (sous la forme d'un quadruplet donnant les identificateurs de processeurs physique et virtuel [7], de tâche et de fil d'exécution) ou encore la date de production de l'événement.

7 . Le système utilisé comporte un routeur logiciel (DEBBAGE *et al.* 1991) qui autorise la déclaration d'autant de processeurs que souhaité et la spécification du placement de ces processeurs virtuels sur les processeurs physiques effectivement disponibles pour l'exécution.

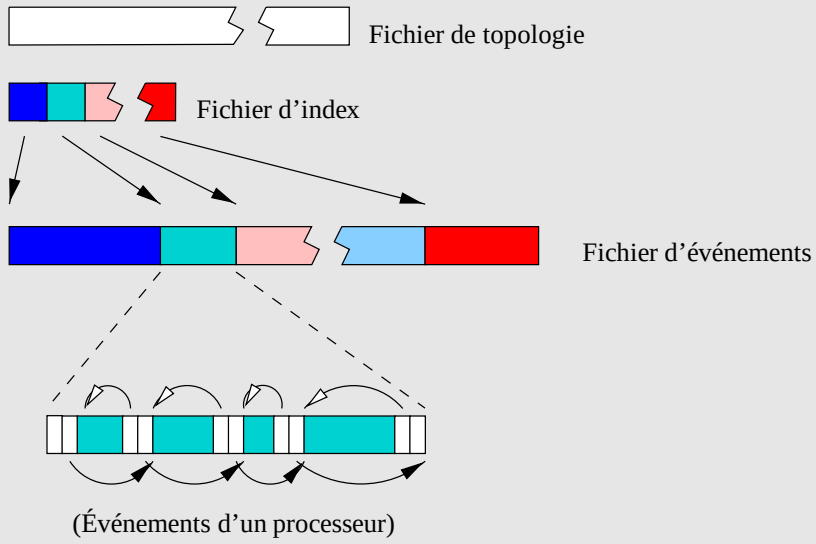


FIG. 2.3. – *Composition d'une trace* TAPE.

Événements

Chaque fonction de la bibliothèque OUF/TRITA génère un ou plusieurs événements décrivant l'action effectuée par cette fonction : alors que la création ou la mort d'un fil provoquent un changement d'état immédiat, l'émission d'un message est une opération bloquante à laquelle correspondent deux événements, le premier lorsque le processus émetteur prend rendez-vous avec un destinataire et le second lorsque ce rendez-vous est fini.

De même que l'on peut classer les différentes fonctions en catégories distinctes, les événements appartiennent à des classes déterminées comme les événements de processus (naissance et mort d'une entité), de communication (envoi et réception d'un message), ou encore de synchronisation (prise et libération d'un sémaphore). Le tableau 2.4 page ci-contre présente les événements générés par TAPE classés par catégorie[8].

8 . Les noms des événements ont été simplifiés afin de ne pas surcharger la présentation. Les noms véritables des événements sont les noms donnés une fois mis en majuscules et précédés de « EVENT_ », *i.e.* l'événement « Send » s'appelle en réalité « EVENT_SEND ».

Catégorie	Nom	Action
Processus	TskBth	Naissance d'une tâche
	TskDth	Mort d'une tâche
Fils	ThdBth	Naissance d'un fil
	ThdDth	Mort d'un fil
	Run	Lancement d'un fil
	Stop	Arrêt d'un fil
Communication	Send	Début d'envoi
	TSend	Début d'envoi avec délai
	EndSend	Fin d'envoi
	Receive	Début de réception
	TReceive	Début de réception avec délai
	EndReceive	Fin de réception
	Timeout	Expiration du délai de communication
Sémaphores	SemInit	Initialisation d'un sémaphore
	SemP	Prise de sémaphore
	SemGOk	Test et prise de sémaphore
	SemG	Fin de prise de sémaphore
	SemV	Libération de sémaphore
Alternatives	Alt	Alternative bloquante
	NAlt	Alternative non bloquante
	EndAlt	Fin d'alternative
Attente	Wait	Attente (sommeil)
	EndWait	Fin d'attente

TAB. 2.4. – *Événements générés par* TAPE.

2.3.2 Lecture de traces

Fonctionnement du lecteur

Étant donnée la structure d'une trace TAPE, la lecture de traces est une opération un peu compliquée. Le principal problème à résoudre est dû à la segmentation du fichier d'événements en blocs, laquelle impose un interclassement des événe-

ments des différents blocs afin de garantir la production d'événements consécutifs lors d'une lecture séquentielle de la trace.

Le lecteur de traces TAPE utilise des lecteurs spécialisés dans le traitement d'un bloc d'événements triés. Il y a un tel lecteur spécialisé pour chaque producteur de blocs d'événements, *i.e.* un par tâche. Les événements fournis par ces lecteurs sont insérés par le lecteur de traces dans des échéanciers qui permettent de délivrer les événements dans l'ordre voulu (figure 2.4 page suivante).

Le lecteur de traces autorisant une lecture bidirectionnelle de la trace, il gère deux échéanciers, l'un pour les événements à venir et l'autre pour les événements passés. Pour produire un événement, le lecteur de traces effectue donc les opérations suivantes :

- insertion de l'événement courant dans l'échéancier opposé au sens de lecture, *i.e.* dans le passé si la lecture se fait vers l'avant et dans le futur dans le cas contraire[9] ;

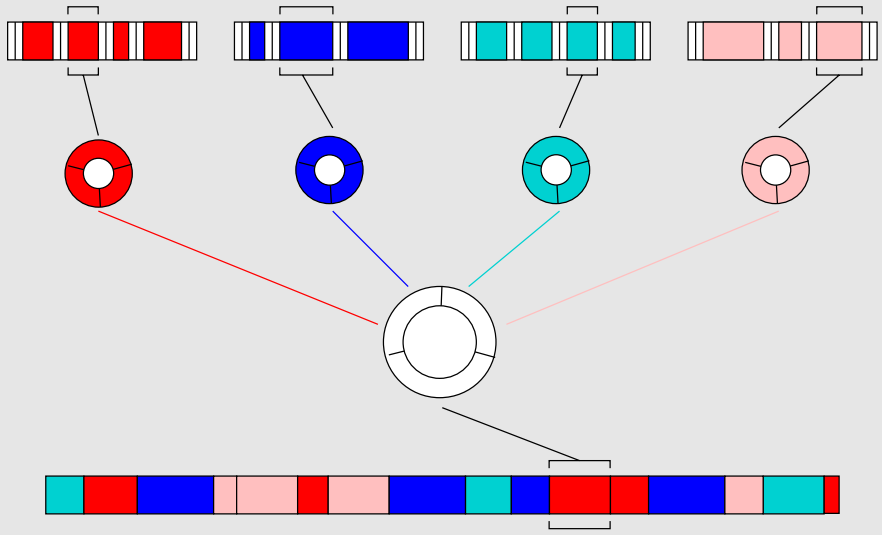
- obtention d'un événement du lecteur spécialisé ayant produit cet événement et insertion de cet événement dans l'échéancier correspondant au sens de lecture ;

- récupération du premier événement de cet échéancier, qui devient l'événement courant.

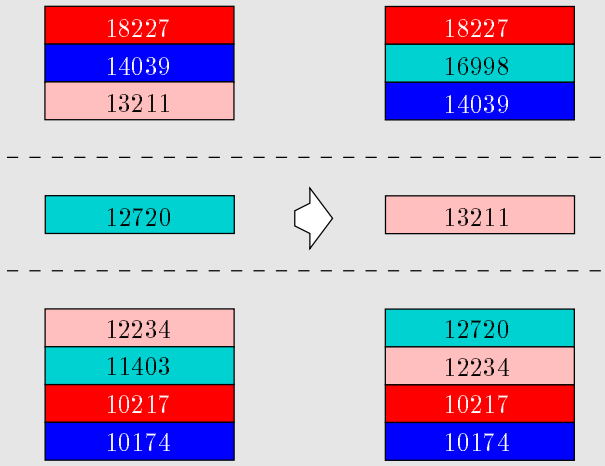
Outre les événements, le lecteur de traces gère des objets représentant les différentes entités d'exécution d'une application, *i.e.* les processeurs, les tâches et les fils d'exécution.

Le lecteur de traces utilise également un dernier objet, le gestionnaire de topologie, qui représente la topologie de l'application obtenue en analysant le fichier de topologie d'une trace TAPE. Le gestionnaire de topologie permet par exemple au lecteur de traces de retrouver la destination d'un lien de communication lors de la lecture d'un événement de communication.

9. Afin de ne pas conserver toute la trace dans l'échéancier, le lecteur de traces détruit d'abord dans l'échéancier tout événement produit par le lecteur ayant produit l'événement à insérer.



(a) Organisation du lecteur de trace.



(b) Manipulation des échéanciers lors d'une lecture en avant.

FIG. 2.4. – *Lecture d'une trace* TAPE. *Le lecteur de traces utilise un lecteur spécialisé par bloc d'événements du fichier d'événements. Les événements fournis par ces lecteurs sont insérés dans des échéanciers pour interclasser les événements afin de les délivrer dans le bon ordre.*

Classes d'événements

Le lecteur de traces transforme les événements contenus dans la trace en objets afin qu'ils soient faciles à manipuler.

La hiérarchie de classes qui est définie correspond à la classification des événements présentée dans le tableau 2.4 page 103. Elle est donnée dans la figure 2.5 page suivante qui indique les différentes classes existantes ainsi que les événements traités par chacune d'entre elles.

2.3.3 Simulation

Le but de la réalisation de simulateur étant d'effectuer une étude de faisabilité de l'application des principes choisis pour la lecture de traces et la simulation, celui-ci ne traite qu'un sous-ensemble du modèle de programmation d'OUF/TRITA.

États des objets manipulés

Comme nous l'avons dit au § 2.2 page 94, le simulateur fait évoluer l'état du système simulé en fonction des événements qu'il reçoit. L'état d'une application OUF/TRITA est un graphe dont les sommets représentent les tâches de l'application et les arêtes les liens de communication entre ces tâches. Les fils d'exécution et les sémaphores ne sont pas simulés.

On notera que même si les processeurs des machines sur lesquelles les traces TAPE sont prises peuvent être multiprogrammés il est impossible au traceur de connaître les instants où se fait l'ordonnancement car la prise de traces se fait au niveau de l'application et non du système. En conséquence les seules informations qu'il puisse donner sur les changements d'état d'une tâche concernent ses communications. On distingue ainsi déjà trois états possibles pour une tâche : elle est dans l'état *actif* lorsqu'elle n'est pas en train de communiquer, dans l'état *communicant* lorsqu'elle est effectivement en train de communiquer avec une autre tâche (c'est-à-dire lorsque les données sont effectivement échangées entre deux tâches ayant établi un rendez-vous) et dans l'état *inactif* lorsqu'elle est bloquée alors qu'elle cherche à communiquer avec une tâche qui n'est pas encore arrivée

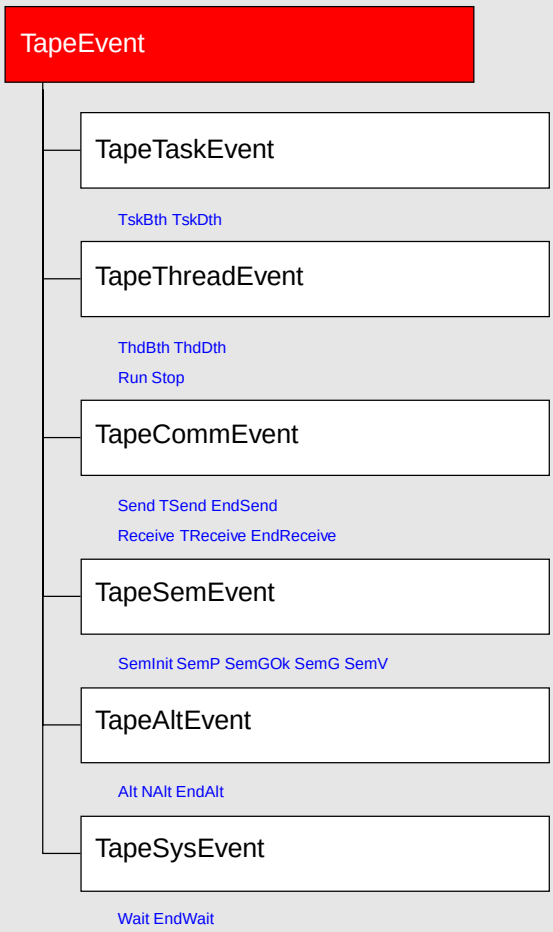


FIG. 2.5. – *Hiérarchie des classes d'événements de* TAPE. *Les classes sur fond rouge sont des classes abstraites.*

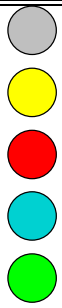
au rendez-vous. Les deux derniers états possibles sont l'état *mort* et l'état *inutilisé* qui indiquent respectivement que la tâche concernée a fini de s'exécuter[10] ou qu'elle n'est pas encore née (mais le simulateur connaît son existence puisqu'il dispose de la topologie de l'application).

Les liens de communication peuvent *a priori* être dans un des deux états sui-

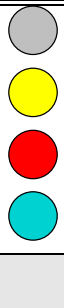
10. Le comportement par défaut du simulateur est de conserver une tâche morte afin qu'il soit possible d'accéder aux statistiques qu'il lui associe, voir plus loin.

vants : l'état *communicant* lorsqu'ils sont en train de transporter des données et l'état *inutilisé* dans le cas contraire. En fait le simulateur génère non seulement ces deux états mais en ajoute également deux autres : un lien est dit dans l'état *inactif* lorsque des données sont attendues sur ce lien (que ce soit en émission ou en réception) et dans l'état *mort* lorsqu'il ne peut plus être utilisé (*i.e.* lorsque la tâche qui peut émettre sur ce lien est morte). L'utilisateur dispose ainsi d'informations plus précises sur l'état de son application à un instant donné. L'utilisation de ces états supplémentaires dans des visualisations permet aussi de donner une vision bien meilleure des communications puisque non seulement la tâche mais également le lien concerné par une communication changent d'état.

Les tableaux 2.5 et 2.6 récapitulent les différents états que peuvent prendre les tâches et les liens de communication.

Nom		Signification
Unused		Inutilisé
Dead		Mort
Idle		Inactif
Talking		Communicant
Busy		Actif

TAB. 2.5. – *États donnés aux tâches par le simulateur d'*OUF/TRITA.

Nom		Signification
Unused		Inutilisé
Dead		Mort
Idle		Inactif
Talking		Communicant

TAB. 2.6. – *États donnés aux liens par le simulateur d'*OUF/TRITA.

Attributs associés aux objets simulés

Chacun de ces objets, et le graphe lui-même, est doté d'attributs permettant de connaître non seulement l'état du système mais également d'autres informations utiles pour l'évaluation des performances ou la compréhension de l'application.

Les tableaux 2.7, 2.8 et 2.9 page suivante listent les attributs maintenus par le simulateur. (Les noms utilisés sont ceux que l'on retrouve au cours de l'utilisation de l'environnement, aussi sont-ils en anglais.)

Attribut	Signification
Clock	Date de la dernière modification.
ClockUnits	Unité dans laquelle sont exprimées les dates

TAB. 2.7. – *Attributs du graphe d'état maintenus par le simulateur d'*OUF/TRITA.

Attribut	Signification
Clock	Date de la dernière modification
State	État du nœud
Name	Identificateur du nœud dans l'application (donne également son type)
Nickname	Surnom du nœud donné par l'utilisateur
Protagonist	Nœud avec qui une communication est en cours
PercentBusy	Pourcentage de temps passé dans l'état Busy
PercentIdle	Pourcentage de temps passé dans l'état Idle
PercentTalking	Pourcentage de temps passé dans l'état Talking

TAB. 2.8. – *Attributs des nœuds maintenus par le simulateur d'*OUF/TRITA.

On remarquera qu'étant donné que la topologie de l'application est connue *a priori*[11] il est possible d'initialiser le simulateur avec un graphe correspondant à cette topologie. Les différents objets de ce graphe sont alors initialisés dans l'état Unused en attendant leur première utilisation.

11 . À l'exception des informations concernant les fils d'exécution dont la création et la destruction sont entièrement dynamiques.

Attribut	Signification
Clock	Date de la dernière modification
State	État du lien de communication
Name	Identificateur du lien dans l'application
Origin	Surnom du nœud d'où part le lien
Destination	Surnom du nœud vers lequel va le lien
Usage	Nombre d'utilisateurs du lien
Traffic	Volume de données en cours de transfert
TotalTraffic	Volume de données global transféré
PercentUnused	Pourcentage de temps passé dans l'état Unused
PercentIdle	Pourcentage de temps passé dans l'état Idle
PercentTalking	Pourcentage de temps passé dans l'état Talking

TAB. 2.9. – *Attributs des liens maintenus par le simulateur d'*OUF/TRITA.

Fonctionnement du simulateur

L'annexe A.1 page 197 décrit précisément le traitement effectué par le simulateur pour chacun des types d'événements qu'il connaît. Elle illustre également les changements d'états produits par ces événements.

2.4 Support des applications PVM

PVM est une bibliothèque de programmation parallèle qui permet de considérer un ensemble quelconque de machines hétérogènes comme une machine parallèle (homogène), permettant de développer et d'exécuter des applications parallèles avec un investissement matériel réellement minimal.

Les applications PVM peuvent être de type MIMD ou SPMD et sont constituées par des tâches communiquant par envoi de messages. Le modèle d'envoi de message supporte l'envoi non-bloquant ainsi que les réceptions bloquante et non-bloquante. Il comporte en outre des opérations globales ou de groupe permettant l'échange complexe d'informations entre un ensemble de tâches. Le modèle PVM

est présenté au § A.1.2 page 235.

2.4.1 Prise et format de trace

Les applications écrites avec PVM peuvent être tracées à l'aide du traceur logiciel TAPE/PVM (MAILLET 1995). Celui-ci étant une adaptation à PVM du traceur TAPE présenté au § 2.3.1 page 99, le lecteur est invité à se rapporter à cette présentation en ce qui concerne l'organisation du traceur et le mécanisme de prise de trace ; de même, seules les différences entre les deux traceurs sont présentées dans la section qui suit.

Prise de trace

La production des traces se fait avec une version spécialisée de l'application, le source de celle-ci étant modifié par un préprocesseur pour supporter la prise de trace. Cette application est ensuite liée avec une bibliothèque contenant les fonctions nécessaires à la mise en place des tâches du traceur ainsi qu'à la prise et à l'enregistrement des traces.

Format de trace

Le traceur TAPE/PVM génère un fichier d'événements non ordonnés, sous forme textuelle. Les événements ne sont pas chaînés, et seule la lecture vers l'avant est possible. Chaque ligne de ce fichier contient un événement dont les informations sont représentées sous forme numérique. Avant que la trace ne soit exploitée, celle-ci est triée avec des outils disponibles sur la machine d'exploitation.

De même que pour les traces TAPE, chaque événement comporte un en-tête donnant des informations communes à tous les types d'événements et une partie variable dépendant du type de l'événement. Les événements sont identifiés par leur date et l'identificateur PVM de la tâche qui les a émis. Contrairement à TAPE, ce traceur ne génère jamais d'événements de début et de fin pour une action donnée. À la place les événements ont une date de début et une durée qui indique le temps pris par l'exécution de la fonction PVM qui est tracée.

La topologie des applications PVM étant dynamique il n'existe pas de fichier de topologie : celle-ci sera déduite des événements de naissances de tâches. Ces événements comportent également des informations sur le processeur sur lequel une tâche s'exécute et permettent donc de reconstruire la topologie physique de la machine virtuelle PVM utilisée par l'application tracée.

Il sera également possible d'associer à la trace un fichier donnant la correspondance entre l'occurrence d'un événement et la fonction générant cet événement (sous la forme du nom du fichier source et de la ligne à laquelle la fonction est appelée) rendant ainsi plus facile l'identification des parties peu performantes d'une application.

Le principal inconvénient de ce format textuel pour l'utilisateur est qu'une trace occupe un espace disque très important. Son principal avantage est qu'il est facile de manipuler une trace avec des outils standards.

Événements

Toutes les fonctions de la bibliothèque PVM sont instrumentées et produisent un événement lors de leur appel, ce qui donne un peu plus de quatre-vingt événements différents.

Mais tous ces événements ne sont pas forcément utilisés pour reconstituer l'état du système au cours de son exécution : certains n'offrent pas d'intérêt (par exemple le fait de demander son numéro de tâche) et d'autres ne sont pas toujours pris en compte (par exemple le temps consacré à l'empaquetage des données peut être considéré soit comme du temps de calcul, soit comme du temps « perdu » pour les communications). Nous nous restreindrons donc pour la suite aux événements les plus intéressants.

Les événements ont été classés par catégories en fonction de leur rôle dans une application PVM. Le tableau 2.10 page ci-contre présente les événements retenus classés par catégorie[12].

12. Les noms des événements ont été simplifiés afin de ne pas surcharger la présentation. Les noms véritables des événements sont les noms donnés une fois mis en minuscules et précédés de « event_ », *i.e.* l'événement « Send » s'appelle en réalité « event_send ».

Catégorie	Nom	Action
Processus	Enroll	Entrée d'une tâche dans le système
	Exit	Sortie d'une tâche du système
Empaquetage	Pk*	Empaquetage (* représente le type qui est empaqueté)
	Upk*	Désempaquetage (* représente le type qui est désempaqueté)
Groupe	JoinGroup	Entrée dans un groupe
	LvGroup	Sortie d'un groupe
Opération dans un groupe	Barrier	Barrière de synchronisation
Communication point à point	Send	Envoi
	PSend	Envoi après empaquetage
	Recv	Réception
	PRecv	Envoi avec désempaquetage
	TRecv	Réception avec délai
	NRecv	Réception non-bloquante
Communication globale	Mcast	Diffusion
Communication dans un groupe	Bcast	Diffusion
	Scatter	Diffusion personnalisée
	Gather	Regroupement
	Reduce	Réduction

TAB. 2.10. – *Événements générés par* TAPE/*PVM.*

2.4.2 Lecture de traces

Fonctionnement du lecteur de traces

Le lecteur de traces TAPE/PVM est bien plus simple que celui de TAPE présenté précédemment, tout simplement parce que la structure du fichier d'événements est extrêmement simple ; il n'y a donc qu'une classe effectuant la lecture

de traces, et celle-ci utilise une version corrigée des fonctions de récupération d'événements fournies avec TAPE/PVM.

Le lecteur ayant obligation de fournir des événements ordonnés il commence par trier la trace si celle-ci en a besoin puis effectue une lecture séquentielle de celle-ci au fur et à mesure des demandes d'événements.

Pour simuler les changements d'états d'une application, il faut qu'un événement ne change qu'une fois l'état d'un objet donné : il est ainsi possible de regarder l'ensemble des états des objets après le traitement de chaque événement pour obtenir un « film » de l'exécution de l'application. Or le traceur ne génère pas d'événements de début et de fin d'action mais plutôt des événements comportant une date de début et une durée, et qui correspondent donc à deux événements logiques. Le lecteur de trace introduit donc les événements logiques manquant lors de la production d'événements. Ces événements de fin, qui sont par convention appelés End* (comme par exemple EndSend pour la fin d'un Send), sont des copies des événements d'origine à ceci près que leur date correspond à la date de l'événement d'origine augmentée de la durée de l'action et qu'ils comportent une indication de leur statut d'événements de terminaison.

L'introduction de ces événements provoque cependant un problème d'ordre des événements puisqu'ils n'appartiennent pas à proprement parler à la trace qui a été triée. Le lecteur de trace utilise donc un échéancier pour interclasser les événements lus dans la trace et ceux qu'il introduit, afin de produire les événements dans un ordre correct.

Pour produire un événement, le lecteur de traces effectue donc les opérations suivantes :

- lecture d'un événement dans le fichier d'événements ;
- si l'événement a une durée, création d'un événement de fin lui correspondant et insertion de ce dernier dans l'échéancier ;
- insertion de l'événement lu dans l'échéancier ;
- récupération du premier événement de l'échéancier, qui devient l'événement courant.

Outre les événements, le lecteur de trace gère des objets représentant les différentes entités d'une application, *i.e.* les processeurs, les tâches et les groupes

auxquelles ces dernières appartiennent.

Classes d'événements

Le lecteur de traces transforme les événements contenus dans la trace en objets afin qu'ils soient faciles à manipuler.

La hiérarchie de classes qui est définie correspond à la classification des événements présentée dans le tableau 2.10 page 113. Elle est donnée dans la figure 2.6 page suivante qui indique les différentes classes existantes ainsi que les événements traités par chacune d'entre elles.

2.4.3 Simulation

États des objets manipulés

L'état d'une application PVM est un graphe dont les sommets représentent les tâches de l'application et les arêtes les liens de communication entre ces tâches.

Rappelons que le traceur TAPE/PVM est un traceur logiciel qui effectue une prise de trace au niveau de l'application et non du système (qu'il s'agisse du système d'exploitation des machines sur lesquelles l'application est exécutée ou simplement de PVM lui-même). Les informations que peut fournir le traceur quant aux changements d'états de l'application sont donc limitées : il est possible de savoir quant une tâche appelle une fonction PVM et quand cet appel de fonction se termine mais il est impossible de savoir exactement ce qui se passe pendant que la fonction est exécutée.

Nous dirons d'une tâche qu'elle est dans l'état *actif* lorsqu'elle calcule et dans l'état *mort* lorsqu'elle a fini de s'exécuter[13]. Elle sera dite dans l'état *système* lorsqu'elle est en train d'effectuer une fonction non bloquante de PVM : le temps passé dans cet état indique le surcoût (« overhead ») lié à l'appel des fonctions

13. Le comportement par défaut du simulateur est de conserver une tâche morte afin qu'il soit possible d'accéder aux statistiques qu'il lui associe, voir plus loin. Il est cependant envisageable de supprimer ces tâches afin de ne pas « encombrer » l'état (si l'état comporte un nombre très important d'objets il se peut que les performances du simulateur se dégradent).

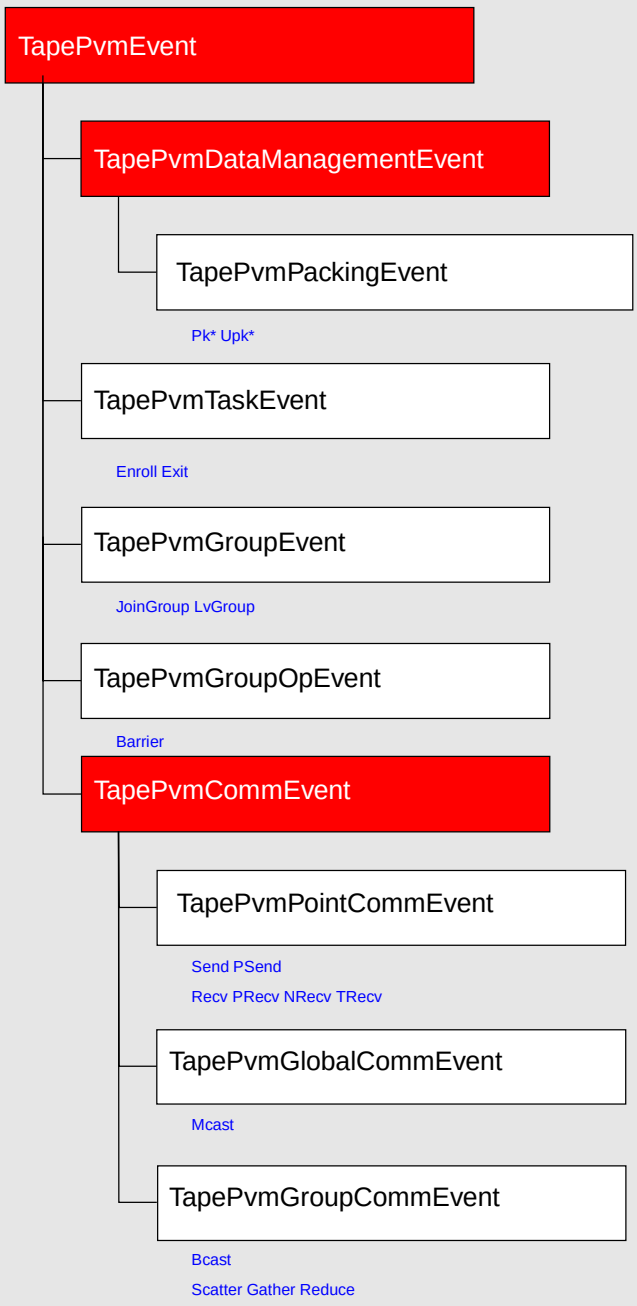


FIG. 2.6. – *Hiérarchie des classes d'événements de* TAPE/PVM. *Les classes sur fond rouge sont des classes abstraites.*

de la bibliothèque de programmation PVM. En ce qui concerne les opérations de communication une tâche peut être dans l'état *inactif* lorsqu'elle est bloquée ou dans l'état *communicant* lorsqu'elle reçoit un message de façon bloquante. Cet état n'a pas de sens pour les émissions de messages puisqu'elles sont non bloquantes.

L'identification des états associés aux liens de communications est nettement plus délicate. En effet l'envoi de message en PVM est non bloquant : la fonction d'émission rend la main à la tâche appelante dès que les données ont été placées dans un tampon système et envoyées au(x) destinataire(s). Si l'on considère également le fait que les fonctions de réception ne donnent qu'une indication de la présence des données au moment de l'appel comme seule information relative à la transmission des données il est impossible de déterminer le temps effectif pris par la communication dans le cas où le message est déjà présent lors de l'appel de la fonction de réception (figure 2.7 page suivante).

Les conventions suivantes sont donc appliquées pour l'état d'un lien. Un lien est dans l'état *inutilisé* lorsqu'il ne véhicule pas de données. Un lien entrant (*i.e.* sur lequel on fait une réception) est dans l'état *bloqué* lorsqu'une tâche attend des données sur ce lien et dans l'état *communicant* lorsque les données attendues circulent effectivement. Nous dirons de plus qu'un lien est dans l'état *système* lorsque des données ont été émises sur ce lien mais pas encore réceptionnées : cela permet de savoir quelles sont les tâches dont les données n'ont pas encore été reçues. Enfin un lien est dans l'état *mort* lorsque la tâche pouvant émettre sur ce lien l'est elle-même.

Les tableaux 2.11 et 2.12 page 119 récapitulent les différents états que peuvent prendre les tâches et les liens de communication.

Attributs associés aux objets simulés

Chacun de ces objets, et le graphe lui-même, est doté d'attributs permettant de connaître non seulement l'état du système mais également d'autres informations utiles pour l'évaluation des performances ou la compréhension de l'application. Les tableaux 2.13 page 120, 2.14 page 121 et 2.15 page 121 listent les attributs maintenus par le simulateur et le tableau 2.11 page 119 donne la liste des états que peuvent prendre une tâche ou un lien de communication ainsi que le code de couleurs qui sera utilisé dans les figures représentant le fonctionnement du

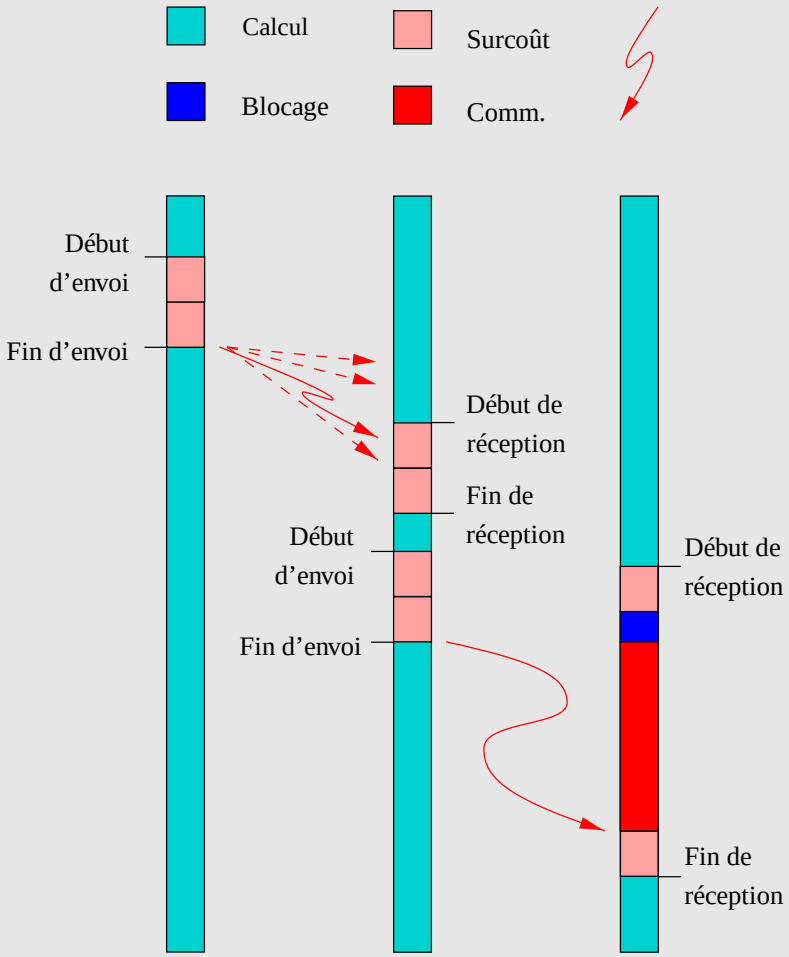
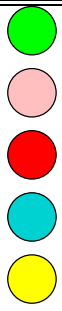


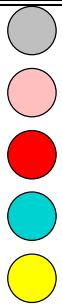
FIG. 2.7. – *Lorsqu'un message est déjà présent quand une tâche demande à le recevoir (à gauche) il est impossible de connaître le moment exact de son arrivée et donc le temps passé à communique effectivement. S'il n'est pas encore présent alors il est possible de reconstruire la communication.*

simulateur. (Les noms utilisés sont ceux que l'on retrouve au cours de l'utilisation de l'environnement, aussi sont-ils en anglais.)

On notera que l'attribut Traffic d'un lien s'interprète différemment suivant l'état de celui-ci : s'il est dans l'état Overhead le trafic est le volume de données qui doit encore être réceptionné alors que s'il est dans les états Idle ou Talking ce

Nom		Signification
Busy		Actif
Overhead		Système
Idle		Inactif
Talking		Communicant
Dead		Mort

TAB. 2.11. – *États donnés aux tâches par le simulateur de PVM.*

Nom		Signification
Unused		Inutilisé
Overhead		Système
Idle		Inactif
Talking		Communicant
Dead		Mort

TAB. 2.12. – *États donnés aux liens par le simulateur de PVM.*

trafic correspond au volume de données de la communication en cours.

Le simulateur est initialisé avec un graphe vide puisqu'aucune information sur la topologie de l'application n'est disponible. Celle-ci apparaîtra au fur et à mesure de l'avancement de la simulation.

Fonctionnement du simulateur

L'annexe A.2 page 200 décrit précisément le traitement effectué par le simulateur pour chacun des types d'événements qu'il connaît. Elle illustre également les changements d'états produits par ces événements.

2.4.4 Performances des composants réalisés

Cette section présente une étude des performances (en termes de rapidité) du lecteur de traces TAPE/PVM ainsi que du simulateur de PVM dont le but est d'évaluer la validité des choix de conception qui ont été faits, à savoir l'utilisation d'une bibliothèque de graphe générale, la transformation des événements en objets afin de pouvoir les manipuler et le maintien systématique d'attributs associés aux objets simulés.

Conditions d'expérience

Les mesures ont été faites sur une machine assez lente (à base de processeur Intel 486DX2/66) sur plusieurs milliers d'opérations pour chaque type d'événement. Cette machine disposait de 16 Mo de mémoire et d'un disque dur lent (temps d'accès d'environ 11 ms). On notera que lors des expériences effectuées nous n'avons observé aucune pagination mémoire et que les mesures n'ont donc pas été perturbées. La machine n'était de plus soumise à aucune charge réseau. Il est cependant raisonnable d'attendre un accroissement des performances important sur des machines modernes (de l'ordre de 2 avec un processeur Pentium, 3 et plus sur une station de travail moderne).

Attribut	Signification
Clock	Date de la dernière modification
ClockUnits	Unité dans laquelle sont exprimées les dates

TAB. 2.13. – *Attributs du graphe d'état maintenus par le simulateur de PVM.*

Attribut	Signification
Clock	Date de la dernière modification
State	État de la tâche
Name	Identificateur de la tâche dans l'application
Nickname	Surnom de la tâche donné par l'utilisateur
Groups	Groupes auxquels la tâche appartient
Protagonists	Tâche(s) avec qui une communication est en cours
PercentBusy	Pourcentage de temps passé dans l'état Busy
PercentOverhead	Pourcentage du temps passé dans l'état Overhead
PercentIdle	Pourcentage de temps passé dans l'état Idle
PercentTalking	Pourcentage de temps passé dans l'état Talking

TAB. 2.14. – *Attributs des tâches maintenus par le simulateur de PVM.*

textbfAttribut	Signification
Clock	Date de la dernière modification
State	État du lien de communication
Name	Identificateur du lien dans l'application
Origin	Surnom de la tâche d'où part le lien
Destination	Surnom de la tâche vers laquelle va le lien
Usage	Nombre d'utilisateurs du lien
Traffic	Volume de données en cours de transfert
TotalTraffic	Volume de données global transféré
PercentUnused	Pourcentage de temps passé dans l'état Unused
PercentOverhead	Pourcentage de temps passé dans l'état Overhead
PercentIdle	Pourcentage de temps passé dans l'état Idle
PercentTalking	Pourcentage de temps passé dans l'état Talking

TAB. 2.15. – *Attributs des liens maintenus par le simulateur de PVM.*

Afin de faciliter la lecture de ce qui suit le nom des classes d'événements est abrégé. Lorsque des mesures différentes ont été faites pour divers événements d'une même classe le nom de l'événement concerné est indiqué entre parenthèses après le nom abrégé de sa classe.

Dans les graphiques le « nombre d'objets » porté en abscisse correspond au nombre de tâches PVM présentes dans le graphe d'état sauf pour les événements de classe GroupOp où il représente le nombre de groupes auquel une tâche appartient.

Performances du lecteur de traces

La figure 2.8 page ci-contre indique les performances du lecteur de traces en termes du nombre d'événements d'une classe donnée produits par seconde. Ce temps comprend non seulement la lecture des événements sur disque mais également la production d'un objet de la classe adaptée à un événement donné après l'avoir lu.

Nous pouvons faire les remarques suivantes sur les mesures de rapidité du lecteur de traces qui ont été obtenues :

- le temps de production d'un événement dépend bien plus du nombre de données à lire sur disque que du coût de la transformation de celles-ci en un objet (par exemple la seule différence entre les événements Enroll et Exit est que le premier comporte un nom de machine, un nom d'architecture et une vitesse de processeur , et il est produit deux fois moins rapidement que le second) ;

- la recherche de l'objet associé à une tâche ou à un groupe est une opération dont le coût est négligeable par rapport à celui de la lecture d'un événement : on n'observe pas de diminution particulière du nombre d'événements produits lorsque le nombre de ces objets augmente (les mesures des événements de classe GroupOp par exemple sont faites avec n tâches appartenant chacune à un groupe différent, n étant le nombre porté en abscisse du graphique) ;

- la diminution rapide de la vitesse de production des événements de classe GlobalComm est due à la taille des données à lire : l'événement lu est une

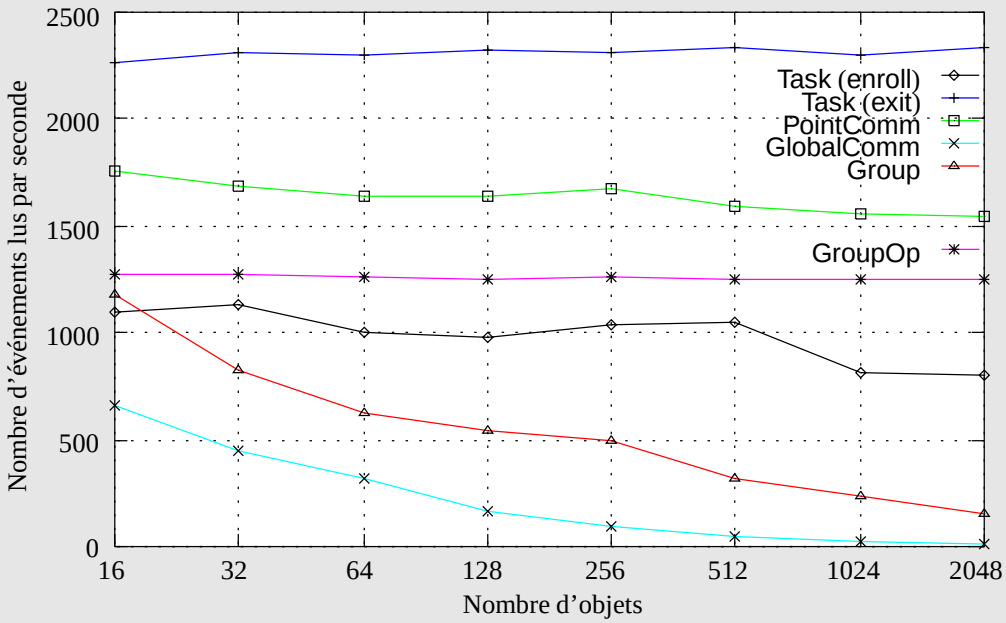


FIG. 2.8. – *Nombre d'événements PVM lus par seconde. Le temps de lecture comprend non seulement le temps d'accès aux données sur le disque mais également la production d'un objet de la classe représentant l'événement.*

diffusion globale qui comporte la liste des tâches concernées par la diffusion, dont le nombre est donné en abscisse sur le graphique[14] ;

– la baisse de performance de la production des événements de classe Group est due au fait que les objets représentant les événements associés à un groupe ont dans leurs informations une copie de la liste des membres du groupe au moment de l'événement : c'est le fait d'effectuer cette copie qui apparaît dans les mesures.

Il faut également noter que la bibliothèque de lecture de traces réalisée utilise elle-même une bibliothèque de lecture en C : outre le fait qu'il y a double transformation des données de la trace le peu de performances de la lecture est

14. Remarquons que le cas $n = 2\,048$ peut être considéré comme un cas d'école (sur une machine IBM SP2 avec 32 processeurs il est impossible de créer autant de tâches PVM, par exemple).

dû à l'utilisation des fonctions d'entrées-sorties de haut niveau de la bibliothèque standard C. La réécriture de la bibliothèque de lecture en utilisant des fonctions spécialement optimisées pour lire le fichier de trace après l'avoir entièrement placé en mémoire virtuelle permettrait d'obtenir un gain de temps appréciable (rendement du lecteur multiplié par un un facteur deux ou trois au moins) ainsi qu'une réduction notable de la charge du système sur lequel le simulateur est placé.

Performances du simulateur

La figure 2.9 indique les performances du simulateur PVM sous la forme du nombre d'événements d'une classe donnée simulés par seconde. Ce temps comprend le traitement complet de l'événement, *i.e.* le fait d'effectuer tous les changements d'états et d'attributs nécessaires à la simulation de l'événement.

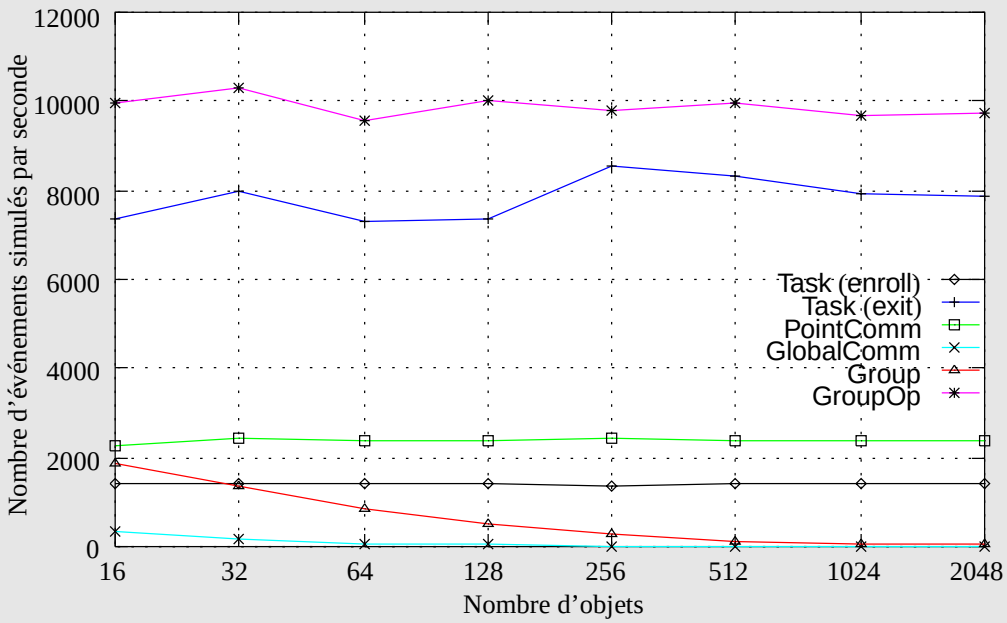


FIG. 2.9. – *Nombre d'événements PVM simulés par seconde.*

Tous les événements ne provoquant pas le même nombre de changements d'états, la figure 2.10 page ci-contre présente les performances du simulateur uen

fois ramenées au nombre de mises à jour de l'état du système effectuées par seconde afin de pouvoir comparer plus facilement ses performances pour chaque classe d'événement.

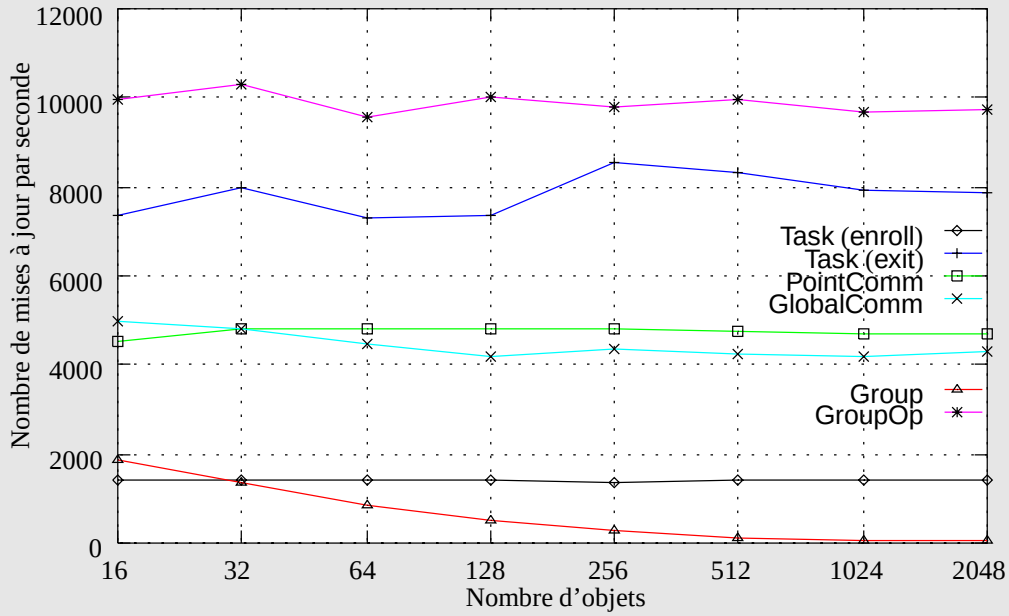


FIG. 2.10. – *Nombre de mises à jour du graphe d'état PVM par seconde.*

Nous pouvons faire les remarques suivantes sur les performances du simulateur de PVM qui ont été obtenues :

– l'apparente aberration des mesures du traitement des événements de classes Group et GlobalComm s'explique par le coût de la mise à jour de l'attribut Groups d'une tâche qui contient la liste des membres du groupe (dans le cas de ces mesures, autant de concaténations de chaînes de 8 caractères que de membres dans le groupe)[15] ;

– l'utilisation d'attributs sous forme de chaînes de caractères n'est pas très

15. Notons ici également que le cas $n = 2\,048$ est un cas d'école plutôt qu'un cas représentatif des applications qui seront simulées.

pénalisante sauf dans le cas d'attributs très longs, cas peu probable[16] ;

– si l'on élimine le cas dégénéré que nous venons d'identifier, on remarque que l'événement le plus plus coûteux à simuler est celui qui indique la naissance d'une tâche (événement Enroll) : la création de l'objet représentant une tâche dans le simulateur s'accompagne en effet de l'initialisation de structures de données propres au simulateur ainsi qu'à la mise en place de tous ses attributs, ce qui explique qu'elle prenne un certain temps ;

– les écarts de performances constatés entre les traitements de différentes classes d'événements viennent du nombre d'objets à mettre à jour : une communication nécessite la recherche (et éventuellement la création) d'un lien de communication alors que la fin d'une tâche non ;

– les performances du simulateur sont très acceptables puisqu'il est capable d'effectuer plusieurs milliers de changements d'états par seconde (plus de 4000 si l'on exclut la naissance d'une tâche et les événements d'entrée et de sortie de groupe, ce qui permet d'attendre de très bonnes performances lors de la simulation d'applications dont la topologie est mise en place une fois pour toute au début de l'application, ce qui est le cas de la très grande majorité des applications de calcul scientifique actuelles).

Validation

Afin de vérifier la correction des mesures de performance données précédemment, nous avons mesuré le temps de traitement de diverses traces. Ce temps, ainsi que les temps prédits d'après nos mesures, sont donnés dans le tableau 2.16 page suivante. Ce tableau confirme les mesures obtenues, lesquelles peuvent donc servir d'estimation du temps de lecture et de simulation nécessaire au traitement d'une trace.

On notera que les traces ont été filtrées pour enlever les événements qui ne sont pas traités par le simulateur ; un tel filtrage est de toute manière possible lors de la première ouverture d'une trace par le lecteur de traces PVM puisqu'il lui faut d'abord la trier pour garantir que les événements seront délivrés par ordre de dates croissantes.

16 . Qui plus est, le seul événement pouvant avoir un attribut très long a peu de chances d'apparaître très souvent dans une application : étant donné le coût de l'utilisation des fonctions de groupes dynamiques de PVM, peu d'applications en font un usage intensif.

Application		Catégorie d'événement						Temps (ms)	
		GC	G	GO	PC	TE	TX	Prédit	Mesuré
Fft1d16p128t65536n	Total	0,00	0,00	0,00	2175,05	215,68	74,93	2465,6706	2536,5563
	Lect.	0,00	0,00	0,00	1586,26	124,95	57,58	1768,8057	
	Simu.	0,00	0,00	0,00	588,78	90,72	17,34	696,8649	
Fft1d16t65536n	Total	0,00	0,00	0,00	121,31	27,39	10,15	158,8661	177,1349
	Lect.	0,00	0,00	0,00	71,67	15,56	7,84	95,0812	
	Simu.	0,00	0,00	0,00	49,63	11,83	2,31	63,7849	
Fft2d16p2048n	Total	0,00	22,20	18,56	1120,14	27,39	10,15	1198,4663	1286,6113
	Lect.	0,00	13,63	16,95	660,89	15,56	7,83	714,8962	
	Simu.	0,00	8,57	1,60	459,24	11,83	2,31	483,5701	
Fft2d16p32t2048n	Total	0,00	52,66	46,54	4034,49	52,92	19,29	4205,9348	4511,4878
	Lect.	0,00	32,57	43,43	2316,81	29,95	14,90	2437,6888	
	Simu.	0,00	20,09	3,10	1717,68	22,96	4,39	1768,2459	

TAB. 2.16. – *Comparaison des temps de traitements réels et prédits (d'après les mesures de performance effectuées). Les catégories d'événements données dans le tableau sont abrégées : GC pour GlobalComm, G pour Group, GO pour GroupOp, PC pour PointComm et TE et TX pour les événements Enroll et Exit de catégorie Task. Seuls les temps de lecture et de simulation prédits sont disponibles par catégorie.*

Conclusion

Les expériences réalisées montrent qu'il existe une différence de performances importante entre la lecture de traces et la simulation. Cette différence est principalement due à la lecture des événements sur disque et ne saurait être éliminée même s'il est certainement possible de la réduire considérablement en optant pour des disques rapides [17] et en optimisant la lecture de traces comme suggéré précédemment. Ce qu'il faut retenir est que le goulot d'étranglement de la génération d'états se situe non pas dans le simulateur mais dans l'accès physique aux données de la trace.

Néanmoins les performances des composants réalisés sont satisfaisantes, le simulateur étant lui particulièrement rapide. Ces composants permettent de traiter un nombre correct d'événements par seconde même sur une machine bas de gamme. Leur temps de traitement est de toute façon très inférieur à ce que sera le temps de mise à jour de n'importe quelle visualisation pour l'évaluation des performances. Les choix généralistes qui ont été faits afin de permettre une grande souplesse d'exploitation des données du simulateur n'entraînent pas de contre-performances du simulateur grâce à l'efficacité des fonctions de manipulation d'objets de ce dernier.

2.5 Réutilisation des composants logiciels

Les composants de lecture de traces permettent à l'environnement de faire abstraction du format de traces utilisé ainsi que de la sémantique des dates associées aux événements (les événements eux-mêmes ne lui sont d'aucune utilité). Les présentateurs d'événements permettent d'obtenir les informations associées aux événements afin de les manipuler (analyse de données, traitements statistiques, etc.). Les simulateurs se chargent de transformer des événements dont ils connaissent la syntaxe et la sémantique en une suite de graphes d'état porteurs d'attributs donnant des informations qualitatives et quantitatives sur les objets du système en cours d'évaluation.

17 . Rappelons que les expériences ont été menées sur une machine disposant d'un disque dur particulièrement lent alors que les stations de travail modernes disposent de disques environ 30 % plus rapides.

Les autres composants de l'environnement peuvent alors être écrits en toute ignorance de la syntaxe et de la sémantique des traces ainsi que du modèle de programmation sous-jacent et sont donc totalement réutilisables (*i.e.* utilisables dans des contextes différents).

Mais les composants que nous avons présenté dans ce chapitre sont extrêmement spécialisés et dépendent à la fois de la syntaxe et de la sémantique des traces manipulées et du modèle de programmation associé à ces traces. En ce sens ils ne sont absolument pas réutilisables dans un autre contexte que celui pour lequel ils ont été dévelopés. Leur but étant d'isoler le reste de l'environnement des particularités d'un format de trace et d'un modèle de programmation donnés, cela n'est pas gênant : il n'y a qu'un contexte dans lequel ils sont utiles, celui de la transformation de la trace en informations manipulables.

On peut cependant se poser la question de savoir si ces composants sont facilement réutilisables au sens objet ou génie logiciel du terme, à savoir s'il est possible de les prendre et de les adapter à de nouvelles contraintes telles qu'un format de trace ou un modèle de programmation différents. La réponse est plutôt négative...

S'il est évidemment possible à moindre coût de changer un lecteur de trace afin qu'il lise les mêmes événements dans un format différent il en va autrement dès qu'il s'agit de lire d'autres types de traces : les événements et la hiérarchie de classes associés étant différents il faut réécrire et la lecture des événements et leur transformation en objets (voire les classes correspondantes elles-mêmes comme ce fut le cas pour les deux lecteurs proposés), soit la totalité du lecteur.

Il en va de même pour le simulateur qui dépend étroitement de ce que lui donne le lecteur de trace et du modèle de programmation utilisé. Nous avons vu que même dans le cas de modèles *a priori* semblables comme CSP et PVM que nous avons traités ici les différences entre les simulateurs sont importantes. Un changement de modèle implique donc l'écriture *ex nihilo* d'un simulateur [18] même s'il est évident qu'il est plus facile de s'inspirer d'un simulateur PVM pour simuler MPI (cf. A.1.3 page 239) qui en est assez proche que pour traiter le modèle de programmation Athapascan (cf. A.4.1 page 246) qui est très différent.

Le manque de généricité de ces composants est le prix à payer pour leur ef-

18 . Même s'il est possible de réutiliser la structure d'un simulateur existant pour ce qui est de la gestion de l'accès aux objets et de la manipulation des attributs, la partie importante du travail à effectuer réside dans l'écriture de la fonction de transition entre états.

ficacité et leur souplesse d'utilisation. Rappelons cependant que l'écriture de ces composants n'est pas une tâche quotidienne et qu'un utilisateur d'un environnement de programmation utilisant Scope devrait le trouver doté des composants nécessaires au support des modèles de programmation qu'il utilise pour ses applications.

L'utilisation d'un lecteur de traces auto-descriptives (PANCAKE *et al.* 1990) permettrait peut-être de n'avoir recours qu'à un seul type de lecteur, à condition toutefois que les traces voulues soient disponibles dans un format auto-descriptif. Un tel choix implique cependant une importante dégradation des performances du lecteur par rapport à un lecteur spécialisé et il est malgré tout nécessaire de disposer de simulateurs spécialisés si l'on veut pouvoir traiter un grand nombre d'événements en un minimum de temps. Et quand bien même un simulateur reconfigurable et efficace serait-il disponible, celui-ci ne pourrait sans doute pas s'accomoder de tous les modèles de programmation que l'utilisateur souhaite exploiter, ou alors sa complexité de configuration le rendrait inapte à la description de modèles complexes.

2.6 Conclusion

Nous avons présenté dans ce chapitre les principes directeurs de la lecture de traces et de la simulation dans Scope. Grâce à un interface abstraite de manipulation des événements, celui-ci est capable de gérer l'avancée dans une trace dont il ne connaît ni la syntaxe ni la sémantique. Nous avons également décrit l'architecture des simulateurs de Scope et le mécanisme des attributs utilisé pour permettre l'accès à l'état des objets simulés ainsi qu'à divers indices de performances associés à ces objets. Les principes présentés ont été mis en application pour la réalisation de deux lecteurs de traces et des simulateurs pour les modèles de programmation associés. Une évaluation des performances des outils dédiés à PVM montre que l'approche générique qui est employée n'entaîne pas de contre-performance et que ces outils sont efficaces. Elle donne également de bonnes indications pour la prédiction du temps de traitement des traces par ces outils.

3

Déplacement dans le temps

LORSQU'UN problème de performance est mis en évidence en analysant l'exécution d'une application, il est important de pouvoir trouver les causes de ce problème. Malheureusement celles-ci peuvent être dues à une action très antérieure à l'instant où le problème est découvert. La question qui se pose alors est de savoir comment revenir à l'instant correspondant à cette action.

Dans les environnements d'évaluation de performances classiques le déplacement dans l'exécution d'une application se fait séquentiellement, par dates croissantes. Il est donc nécessaire de recommencer le déroulement de l'exécution depuis son début pour revenir à la date de l'action qui nous intéresse. Une telle obligation n'incite absolument pas à une exploration interactive de l'exécution de l'application, c'est-à-dire au déplacement à des instants arbitraires pour pouvoir les comparer d'un point de vue des performances de l'application ou encore pour chercher par exemple une corrélation entre une contre-performance notable et un léger « dérapage » l'ayant lointainement précédée.

Nous souhaitons *a contrario* que Scope supporte ce type d'exploration. Pour cela il faut le doter d'un mécanisme de déplacement dans le temps. Ce mécanisme doit en premier lieu être non contraignant afin que l'utilisateur de l'environnement ait envie d'en faire usage et qu'il ne soit pas rebuté par la difficulté de son utilisation. Il doit de plus être simple à mettre en œuvre et conçu de manière à ce qu'il puisse être utilisé quelle que soit la configuration de l'environnement, la

complexité du schéma d'analyse de performance utilisé et la diversité des composants de ce schéma. Il faut enfin qu'il soit suffisamment efficace pour que son coût d'utilisation soit faible par rapport au confort d'exploration qu'il apporte.

3.1 Présentation du problème

Le déplacement dans le temps est le fait de vouloir aller de l'instant présent t_p à un instant destination quelconque t_d. Cet instant destination étant arbitraire, nous distinguons deux cas : l'instant est dans le futur ou il est dans le passé.

Si l'instant est dans le futur, alors le déplacement ne pose pas de problème particulier car il suffit de dérouler la trace d'exécution jusqu'à ce que cet instant (ou l'instant de génération d'un événement qui est le plus proche de cet instant, que nous confondrons désormais avec lui) soit atteint. Nous nous intéresserons donc au cas où l'instant est dans le passé. Lorsqu'on souhaite revenir en arrière dans une exécution, on peut envisager tout d'abord deux approches naïves. La première consiste à repartir du début de l'exécution et à avancer dans celle-ci jusqu'à l'instant voulu. La seconde est de vouloir tout simplement reculer dans l'exécution pour revenir à cet instant antérieur.

La première solution a pour inconvénient immédiat le fait de devoir traiter à nouveau tout le début de la trace. Non seulement cela est pénible mais une telle approche peut également prendre énormément de temps : si l'utilisateur souhaite revenir en arrière d'une dizaine d'événements après en avoir traité plusieurs milliers depuis le début de la trace, le côut de son saut en arrière pour seulement quelques événements se comptera en milliers de traitements et sera certainement prohibitif. Cette approche est donc *a priori* inintéressante : quelle que soit la rapidité du système sur lequel l'environnement fonctionne, il y aura toujours des cas où le déplacement dans le temps avec retraitement complet est inenvisageable.

La seconde approche est encore pire. En effet, même s'il est envisageable de disposer d'un lecteur de traces capable de lire des événements à reculons, il est douteux que cela soit possible pour un simulateur ou même pour des composants d'analyse de données, lesquels transforment souvent leurs données de manière irréversible. Il est donc totalement impensable de chercher à reculer dans une trace : seule la lecture dans le sens croissant des dates est utilisable pour la production d'indices de performances et de visualisations.

Il faut donc trouver une solution intermédiaire. Puisque seule la lecture en avant est possible mais que le traitement de longs segments de trace prend beaucoup de temps nous proposons de mettre en place une solution diminuant le temps de traitement nécessaire pour se déplacer vers une date donnée. En partant du principe que le temps de traitement d'un événement est déjà optimisé et par conséquent incompressible, la seule solution pour parvenir à nos fins est tout simplement de réduire le nombre d'événements à traiter. Pour que cela soit possible il faut que l'environnement puisse repartir non plus du début de la trace mais d'une date située dans le passé de l'instant voulu : plus cette date en sera proche et moins le déplacement prendra de temps.

Rappelons que les composants disponibles dans l'environnement fonctionnent en produisant de nouvelles données à partir de leur état et des entrées qu'ils reçoivent. Pour qu'un tel composant fonctionne correctement à un instant donné il est nécessaire que son état soit cohérent, c'est-à-dire qu'il représente bien l'ensemble des transformations réalisées depuis l'obtention de leur première donnée. Une solution envisageable est de sauvegarder les états des composants et de réaliser le déplacement dans le temps en restaurant les états sauvés. Ces sauvegardes constituent ce que nous appelons un *film d'exécution*.

3.2 Construction d'un film d'exécution

Nous pouvons envisager au premier abord plusieurs façons de construire un film d'exécution. La première est de sauvegarder l'état de l'environnement après chaque événement (figure 1(a) page 135). Cette solution est irréaliste pour au moins deux raisons : elle est très coûteuse en espace disque et, le temps de sauvegarde d'un état étant important, elle ralentit énormément l'avancée dans la trace, décourageant ainsi l'utilisation de l'environnement.

On peut alors penser ne plus stocker un état complet pour chaque événement mais simplement d'en sauvegarder un de temps en temps et ensuite ne sauver que les différences d'états successives pour les événements situés entre ces enregistrements complets (figure 1(b) page 135). Cette solution présente également des problèmes. Tout d'abord, il n'est pas toujours pratique de produire la différence entre deux états successifs d'un composant. Prenons comme exemple un simulateur de Scope qui utilise une bibliothèque de manipulation de graphes dont il ne connaît que l'interface : il lui est possible de sauvegarder un état et de le rechar-

ger, mais il lui est pratiquement impossible d'en sauvegarder simplement quelques parties incomplètes, comme quelques sommets et des arêtes sans destination (ce qui est le cas si la modification de l'état a consisté à mettre à jour l'état d'une tâche et du lien de communication sur lequel elle émet sans que la tâche vers laquelle va le lien soit affectée). D'autre part il faut appliquer plusieurs différences à un état enregistré pour pouvoir reconstituer l'état correspondant à l'instant de destination. Le calcul des différences sur des structures de données complexes, de même ensuite que l'application de ces différences, peuvent prendre en moyenne plus de temps que si l'on omet de calculer les différences et qu'on traite normalement les données.

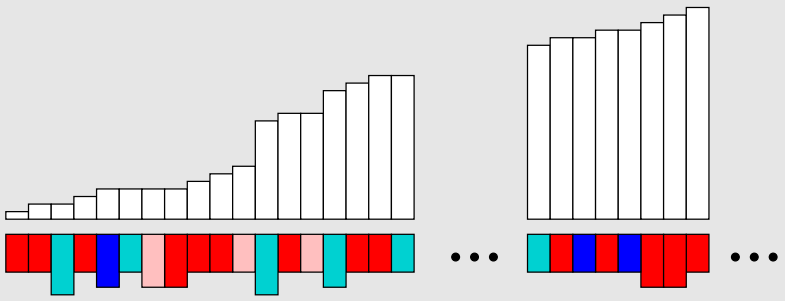
C'est donc cette dernière approche qui a été retenue pour Scope (figure 1(c) page ci-contre). Par rapport aux autres solutions présentées elle a pour avantage de réduire à la fois l'espace nécessaire aux enregistrements d'états pour le déplacement dans le temps, le temps nécessaire à ces sauvegardes et le coût de mise en œuvre du mécanisme pour les composants disponibles dans l'environnement. Nous pensons également qu'elle est valable, par rapport à l'approche de sauvegarde de différences, en ce qui concerne le temps mis pour atteindre un instant à partir de la restauration d'un état complet de l'environnement.

3.3 Organisation du déplacement

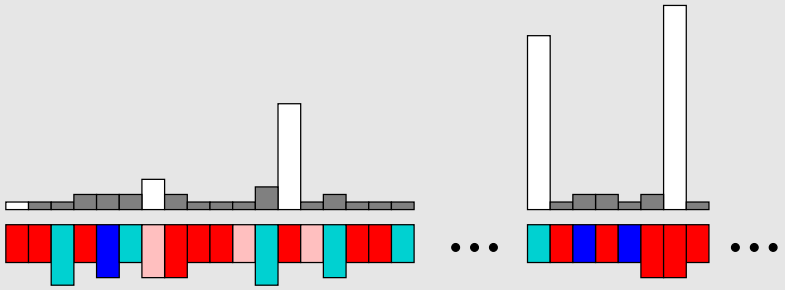
3.3.1 Tâches incombant à l'environnement

Scope se charge de gérer les bases du mécanisme de déplacement dans le temps, c'est-à-dire la sauvegarde de l'état de l'environnement et sa restauration. Il s'occupe également, à travers le composant de contrôle de session, de fournir une interface de manipulation du contrôle de temps à l'utilisateur, interface que nous présentons plus loin. Il gère enfin le choix des instants où l'état de l'environnement est sauvé, choix dont nous discutons plus loin.

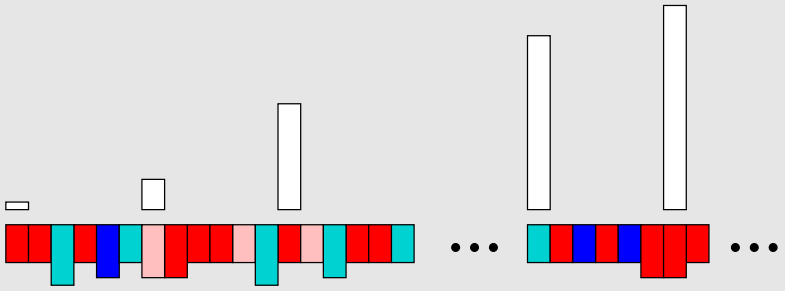
Lorsqu'un déplacement dans le temps est demandé Scope détermine quel est le *point de reprise* correspondant, c'est-à-dire quel est l'instant antérieur le plus proche pour lequel l'état de l'environnement a été sauvegardé. Une fois cet instant connu, il restaure l'état correspondant puis lit autant d'événements que nécessaire pour atteindre l'instant destination, ces événements étant traités normalement par



(a) Sauvegarde après chaque événement.

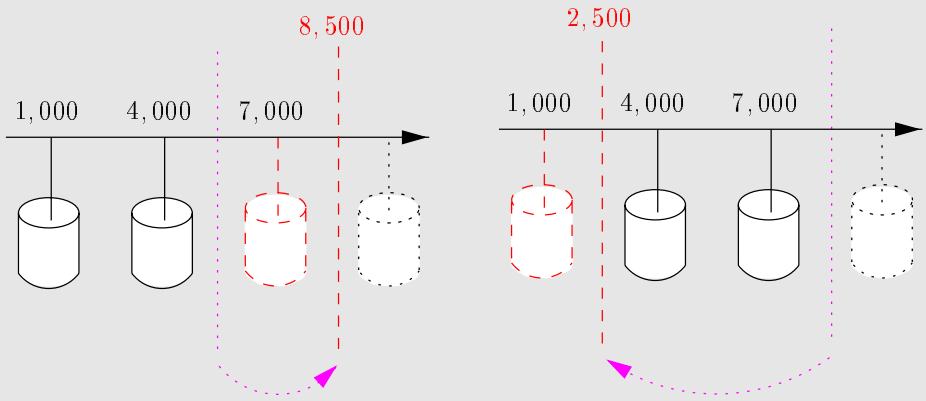


(b) Sauvegarde intermittente avec différences.



(c) Sauvegarde intermittente.

FIG. 3.1. – *Trois possibilités de construction d'un film d'exécution (les barres de couleur représentent les événements et les barres blanches ou grises au-dessus indiquent le volume données à sauver). En (b) on ne sauve l'état que de temps en temps et sinon on stocke la différence entre l'état actuel et l'état précédent ; en (c) on reconstruit normalement les états non sauvés à partir des événements.*



(a) Déplacement en avant.

(b) Déplacement en arrière.

FIG. 3.2. – *Déplacement dans le temps. Les instants où l'état de l'environnement a été sauvé sont figurés par un symbole de fichier. Le dernier état sauvegardé à partir duquel les données sont traitées normalement est dessiné avec des tirets, de même que l'instant destination. On remarque que le sens du saut ne change rien à la manière dont est fait le déplacement dans le temps.*

le graphe d'analyse (figure 3.2).

3.3.2 Mécanismes de sauvegarde et de restauration

Les mécanismes utilisés doivent être les plus simples possibles pour qu'ils ne soient pas eux-mêmes difficiles à mettre en œuvre et pour qu'ils soient aussi efficaces que possibles. Le principe de la sauvegarde d'un état est le suivant : Scope crée un flux d'écriture typé (c'est-à-dire sur lequel on écrit les données dans un format permettant leur relecture sur une machine d'architecture différente) et demande à chacun des composants du programme visuel utilisé de sauvegarder son état dans ce flux. Les composants sont responsables du choix des données qu'ils enregistrent du moment que leur relecture permet la restauration de leur état à l'instant de la sauvegarde. Quant à la restauration d'un état, il s'agit simplement de rouvrir le flux correspondant à l'état à restaurer puis de demander aux composants de relire les données qu'ils y avaient placé lors de la sauvegarde.

En fait les méthodes utilisées sont un peu plus complexes car il faut penser aux problèmes qui peuvent se présenter. Reprenons l'exemple d'un simulateur de Scope : par souci de portabilité et de facilité de manipulation par des outils standards, les fonctions de sauvegarde et de restauration de la bibliothèque de manipulation de graphes utilisée ont été conçues pour utiliser des fichiers de type texte et non des flux de données typés, qui n'existent pas sur tous les systèmes. (Il est certainement possible de tricher un peu et de s'arranger pour écrire le graphe dans un flux de données typé, par exemple en relisant chaque ligne du fichier texte avant de l'écrire dans le flux lors de la sauvegarde et même de recréer un fichier texte d'après ces données lors de la restauration, mais cela va à l'encontre de nos objectifs de simplicité et d'efficacité.) Le flux fourni par Scope est donc ouvert dans un répertoire dédié aux informations composant l'état de l'environnement, et ce répertoire peut être utilisé librement par les composants (sous contrainte de respecter certaines règles de partage de cet espace d'écriture). Dans notre exemple, le simulateur utilise donc un fichier texte comme le demande la bibliothèque de manipulation de graphes dont il se sert, et ne place dans le flux que le nom de ce fichier afin de pouvoir le retrouver pour restaurer son graphe d'état.

Considérons également le fait que le graphe d'analyse est un programme dont la structure n'est pas figée et qui peut être modifié à tout instant par l'utilisateur. Il semble logique en effet que ce dernier puisse ajouter des composants à son programme lorsqu'il souhaite affiner son analyse, par exemple après découverte d'un problème de performance, ou tout simplement afin de changer les visualisations qu'il utilise. La question qui se pose alors est de savoir comment traiter ces graphes changeant lors d'un déplacement dans le temps. La solution que nous avons choisie est de se rappeler la liste des composants présents lors de la sauvegarde d'un état. Seuls ces composants voient leur état restaurés, les autres étant soit réinitialisés afin de ne pas présenter d'état incohérent, soit détruits si l'utilisateur le souhaite. Si un composant dont l'état a été enregistré n'existe plus au moment de la restauration, cet état est inutilisé. Il est ainsi possible de modifier le graphe d'analyse sans pour autant mettre en péril le mécanisme de déplacement dans le temps.

3.3.3 Interaction avec l'utilisateur

L'utilisateur dispose de deux outils pour contrôler le déplacement dans le temps, ces outils étant utilisables à partir du panneau de contrôle de session de

(a) Contrôle de la sauvegarde. (b) Contrôle des marques.

FIG. 3.3. – *Les différents contrôles du déplacement dans le temps disponibles permettent de régler le mécanisme de sauvegarde automatique de l'état de l'environnement mais aussi d'associer des marques à des instants de l'exécution évaluée afin de pouvoir s'y ramener facilement.*

Scope (figure 3.3).

Le premier a trait à la sauvegarde des états et lui permet de décider si et quand une sauvegarde automatique de l'état de l'environnement doit avoir lieu, l'autorise à forcer cette sauvegarde à un instant précis et lui permet de consulter la liste des instants où des sauvegardes ont eu lieu et de se déplacer à ces instants.

S'il est agréable de pouvoir être ramené rapidement aux instants de sauvegarde de l'état de l'environnement, une telle possibilité est assez primitive. Lorsqu'on visualise l'exécution d'une application parallèle on trouve des instants intéressants auxquels on souhaite pouvoir se rapporter ultérieurement. Ces instants ont très peu de chance de coïncider avec les instants de sauvegarde, et même s'ils sont peu nombreux il est souhaitable de pouvoir les retrouver rapidement.

Scope offre donc la possibilité de placer des *marques* dans l'exécution d'une application. Ces marques sont constituées d'un instant associé à un nom et à d'éventuels commentaires multimédia. L'utilisation d'un nom permet de classer facilement les marques et facilite leur gestion, alors que les commentaires autorisent l'ajout d'informations telles que des notes, des questions en suspens ou tout document lié à l'instant marqué. Si l'on annote un instant en faisant référence à la marque associée à un autre instant, par exemple, on dispose d'un moyen efficace de navigation dans l'exécution, similaire dans son principe aux liens hypertextes dans un document classique. La gestion de ces marques est assurée par le second

outil de contrôle du déplacement dans le temps fourni par le panneau de contrôle de session.

3.4 Stratégies et heuristiques

La sauvegarde automatique des états de l'environnement est une bonne chose, mais encore faut-il savoir à quels instants celle-ci doit avoir lieu. C'est le rôle des stratégies et heuristiques de sauvegarde que nous présentons ici.

Nous avons déterminé deux critères de décision que nous utiliserons dans le choix de l'instant de sauvegarde. Le premier est qu'il est possible d'utiliser est le coût de la sauvegarde, tant en espace disque qu'en temps : sachant que plus les sauvegardes sont rapprochées plus leur coût global est elevé mais plus rapide est le déplacement dans le temps, il s'agit d'atteindre un compromis intéressant entre la fréquence des sauvegardes et son coût. Un autre critère est la localité des sauvegardes : une sauvegarde est d'autant plus utile que l'instant auquel elle correspond est proche des instants vers lesquels l'utilisateur se déplace, et si certaines portions de l'exécution ne sont jamais atteintes directement par un déplacement dans le temps alors il est inutilement coûteux de sauvegarder des états dans ces tranches de temps. Alors que le premier critère est facile à déterminer statiquement il n'en va pas de même pour le second puisque le comportement de l'utilisateur est *a priori* totalement inconnu.

3.4.1 Stratégies d'enregistrement

La stratégie par défaut de Scope est extrêmement simple : il n'y a pas de sauvegarde par défaut et c'est l'utilisateur qui force l'enregistrement lorsqu'il le souhaite. Procéder ainsi n'est pas forcément stupide : si l'utilisateur ne souhaite pas se déplacer dans le temps il n'a pas à supporter de perte de temps due aux enregistrements et il lui est toujours possible de sauver l'état de l'environnement lorsque cela lui semble utile (le problème étant évidemment que la détermination de ces instants peut se révéler fort délicate).

Une autre stratégie disponible est d'autoriser l'enregistrement l'état de l'application tous les n événements, n étant choisi par l'utilisateur. C'est une stratégie

très simple à mettre en œuvre mais qui a peu de chances d'être efficace étant donné qu'il est difficile pour l'utilisateur de choisir correctement n (sauf s'il cherche simplement à minimiser le coût des sauvegardes).

Nous pensons qu'un utilisateur désireux de pouvoir se déplacer dans le temps est prêt à consentir le « sacrifice » d'une partie de son temps d'utilisation de l'environnement pour disposer d'une telle possibilité. La troisième stratégie envisagée dans Scope est justement basée sur la mesure de ce sacrifice : si l'utilisateur est prêt à laisser 10 % de son temps pour la sauvegarde, alors Scope essaiera de ne pas y consacrer plus de temps. Cette stratégie est réalisée en mesurant le temps pris par une sauvegarde et en s'interdisant tout enregistrement automatique tant ce temps représente plus que le pourcentage donné du temps que l'environnement passe à traiter la trace et les données en résultant. Ce temps est mesuré chaque fois qu'une sauvegarde est effectuée : plus ce temps sera long (parce que l'état de l'environnement est plus complexe) plus l'enregistrement suivant sera éloigné dans le temps.

Notons qu'il peut être intéressant de combiner différentes stratégies. En se servant par exemple la deuxième et la troisième il est possible d'exprimer une contrainte du type « sauvegarde tous les n événements dans la limite de p % du temps ». Une telle contrainte permet d'obtenir plus de points d'enregistrement (et donc un meilleur temps de déplacement dans le temps) lorsque le temps de sauvegarde est faible.

3.4.2 Heuristiques d'adaptation

Les stratégies que nous avons présentées précédemment ne sont pas vraiment satisfaisantes : elles peuvent permettre d'atteindre en moyenne un compromis correct entre le temps nécessaire à la sauvegarde et celui que prend le déplacement vers un instant donné, mais elles ne sont pas optimisées. Nous avons donc envisagé d'utiliser des heuristiques basées sur le critère de localité afin d'améliorer potentiellement le temps de réponse de l'environnement à une demande de déplacement dans le temps. Ces heuristiques sont basées sur une compréhension plus ou moins fine du comportement de l'utilisateur durant son utilisation de l'environnement.

La première heuristique essaye de faire coïncider les enregistrements avec les placements de marques faits par l'utilisateur, en partant du principe que s'il marque un instant c'est qu'il pense vouloir y revenir ultérieurement. Quand une

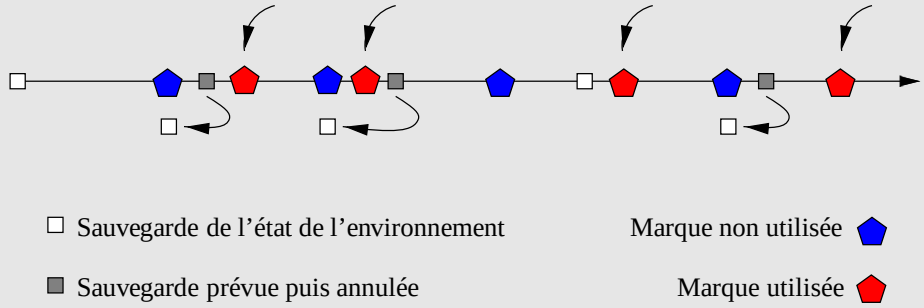


FIG. 3.4. – *Exemple de cas où l'heuristique de sauvegarde anticipée lors du placement d'une marque est inefficace (entraînant même des contre-performances). la plupart des marques placées dans cet exemple ont entraîné un enregistrement anticipé alors que ce sont des marques postérieures à la date de sauvegarde prévue qui sont utilisées pour se déplacer dans le temps : le coût du déplacement est supérieur à ce qu'il serait sans l'heuristique.*

marque est placée Scope considère le moment où la prochaine sauvegarde automatique devrait avoir lieu. Si ce moment est assez proche de l'instant présent alors la sauvegarde a lieu immédiatement[1] : un déplacement vers la marque qui vient d'être placée ne nécessitera donc qu'une simple restauration de l'état de l'environnement.

Le principal inconvénient de cette heuristique est qu'il est facile de rater des marques intéressantes, c'est-à-dire celles qui sont utilisées comme destinations d'un déplacement dans le temps (figure 3.4). Même si elle peut se comporter de manière très satisfaisante si le comportement de l'utilisateur s'y prête (par exemple s'il évite de placer des marques dont il ne servira pas pour le déplacement dans le temps) il est utopique de compter sur une telle situation.

Nous avons donc choisi une seconde heuristique qui complète utilement la précédente ou une des stratégies de sauvegarde présentées précédemment. Le principe est de faire les enregistrements en utilisant une des méthodes précédentes mais de s'autoriser à abandonner certains enregistrements qui semblent inutiles au profit de nouvelles sauvegardes plus fréquemment utilisées. L'abandon de sau-

1 . On notera qu'un cas dégénéré de l'utilisation de cette heuristique permet d'enregistrer systématiquement l'état de l'environnement lors du placement d'une marque.

vegardes permet de ne pas accroitre inutilement l'espace disque occupé par l'ensemble des enregistrements lorsqu'on en crée un nouveau. Notre heuristique est basée sur une analyse des fréquences de déplacement vers les différents instants de l'exécution : si un instant est souvent destination d'un déplacement dans le temps alors la sauvegarde la moins utilisée est abandonnée au profit d'une nouvelle sauvegarde vers cet instant. Le coût de certaines sauvegardes est effectivement augmenté du coût de la suppression d'un enregistrement inutilisé mais nous pensons que celui-ci est presque immédiatement compensé par le gain de temps obtenu en obtenant directement l'état de l'environnement lors du déplacement vers l'instant concerné.

Nous pensons que les différentes possibilités de stratégies et d'heuristiques présentées permettent de rendre tolérable, voire faible, le coût du déplacement dans le temps par rapport aux bénéfices que l'on retire de l'utilisation d'un tel mécanisme.

3.5 Performances

Bien que les mécanismes de déplacement dans le temps présentés ici aient été réalisés, que ce soit au niveau de l'environnement lui-même ou à celui des différents composants disponibles, des problèmes techniques liés à la version de NEXTSTEP utilisée pour le développement nous ont empêché de les utiliser. Il nous a donc été impossible de vérifier pratiquement le coût de ce mécanisme ou le bien-fondé et l'efficacité des stratégies et heuristiques que nous avons choisies. Nous ne pouvons donc pas présenter d'évaluation des performances du déplacement dans le temps.

Nous noterons cependant que la mise en place des méthodes supportant le déplacement dans le temps pour un composant donné est une opération très simple et que notre objectif de minimiser les contraintes liées à cet opération a été atteint.

3.6 Conclusion

Nous avons proposé un mécanisme de gestion du déplacement dans le temps basé sur une sauvegarde et une restauration de l'état complet de l'environnement à

des instants déterminés. Ce mécanisme peut employer différentes stratégies d'enregistrement associées à des heuristiques d'adaptation destinées à les rendre plus efficaces. Même si la sauvegarde et la restauration des états de l'environnement sont des opérations coûteuses, la seconde heuristique proposée fait que sur une session d'évaluation de performance, l'utilisation de ressources pour le déplacement dans le temps se fera au profit des instants qui sont effectivement utilisés comme destination de déplacements dans le temps, évitant autant que possible un gaspillage de ces ressources. Enfin, il suffit qu'un instant de déplacement dans le temps soit un peu éloigné — en nombre d'événements s'entend — du début de la trace pour qu'il soit rentable de supporter une sauvegarde et des restaurations de l'état de l'environnement afin d'éviter de reprendre le traitement de la trace à son début à chaque déplacement.

4

Visualisation

LA VISUALISATION est le moyen par lequel les informations sur le comportement de l'application au cours de son exécution sont présentées à l'utilisateur pour qu'il puisse les analyser. La facilité et la précision avec laquelle cette analyse peut être menée dépend beaucoup de la qualité des visualisations proposées par l'environnement d'évaluation de performance.

Le principal problème de la visualisation pour l'évaluation de performance est de savoir comment présenter les informations afin qu'elles soient facilement compréhensibles (MILLER 1993). Contrairement à la visualisation scientifique pour laquelle cette tâche est facilitée par la possibilité de présenter les informations telles qu'elles se présentent dans la réalité (POST et HIN 1991) il est difficile de trouver une représentation «réelle» d'une application parallèle et de son exécution sur une machine parallèle. Les environnements d'évaluation de performance ont donc recours à des représentations abstraites du comportement des applications, principalement à travers la caractérisation de ce comportement par des indices de performance. Les abstractions utilisées doivent être les plus proches possibles de la manière dont l'utilisateur se représente son application et ses caractéristiques afin de faciliter leur appréhension.

Nous présentons dans ce chapitre les principes de la visualisation en général et dans le cadre de Scope, en insistant particulièrement sur les possibilités de personnalisation et de manipulation des représentations qui sont offertes à l'utilisateur. Les concepts évoqués dans cette présentation sont ensuite mis en évidence

à travers la description d'une visualisation de Scope, cette visualisation permettant la représentation et la manipulation directe d'un graphe représenté sous forme tridimensionnelle.

4.1 Principes

4.1.1 Types de visualisations

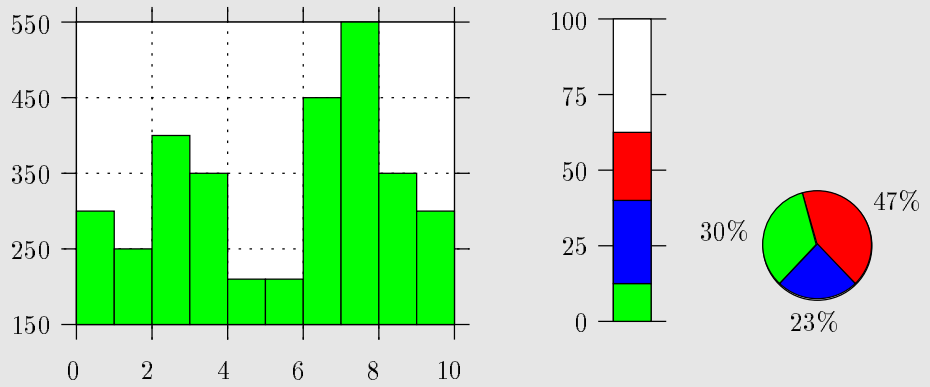
La visualisation pour l'évaluation de performance des applications parallèles est basée sur l'utilisation de deux types de représentations : des *représentations statistiques* de données caractéristiques des performances de l'application et des *représentations spécifiques* au domaine du parallélisme.

Nous distinguons de plus entre les représentations — ou visualisations pour utiliser le terme consacré dans notre domaine — *statiques*, *dynamiques* et instantanées. Nous disons d'une représentation qu'elle est statique lorsqu'elle présente un indicateur global comme un cumul réalisé depuis le début du traitement des données. Nous appelons représentations dynamiques celles qui montrent l'évolution d'un indicateur au cours du temps. Quant aux représentations instantanées elles présentent des informations mises à jour après traitement de chaque événement de la trace et donnent une indication représentative de l'instant courant de l'exécution de l'application.

Visualisations statistiques

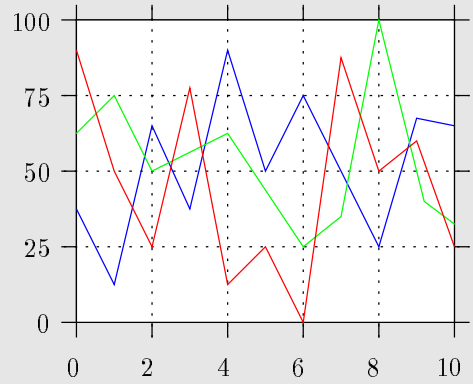
La palette des visualisations statistiques utilisée pour l'évaluation de performance est classique, aussi ne nous y intéresserons nous pas spécialement. Les représentations les plus couramment utilisées sont des histogrammes simples ou cumulés, des jauges, des camemberts, des courbes, des représentations matricielles ou des contours, etc. (figure 4.1 page ci-contre).

Nous plaçons également dans les visualisations statistiques le diagramme de Gantt qui est couramment utilisé dans notre domaine pour présenter l'activité des tâches ou des processeurs : c'est une simple représentation matricielle donnant l'évolution d'un indicateur au cours du temps pour plusieurs entités.

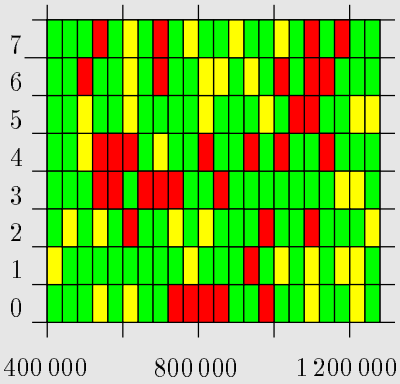


(a) Histogramme.

(b) Jauges et camemberts.



(c) Courbes.

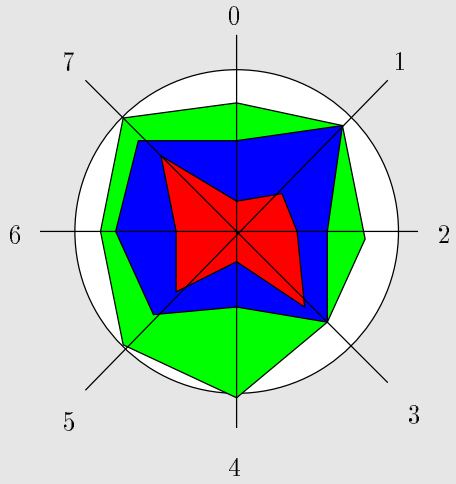


(d) Diagramme de Gantt.

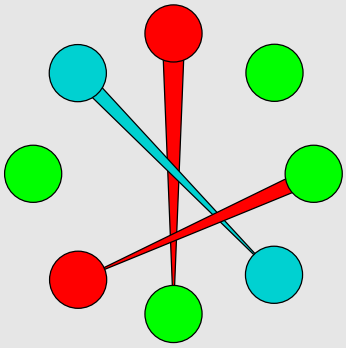
FIG. 4.1. – *Quelques exemples de représentations statistiques couramment utilisées pour la visualisation pour l'évaluation de performance.*

Visualisations spécifiques

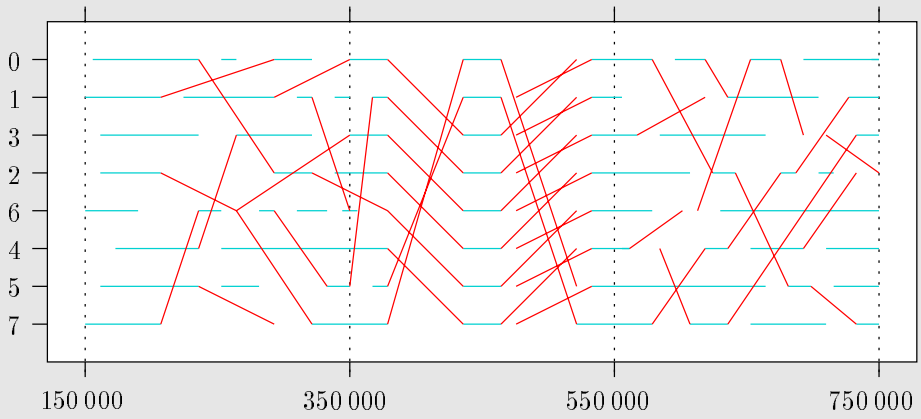
Depuis que la visualisation des applications parallèles existe les concepteurs d'environnements ont cherché à inventer des représentations spécifiques au domaine du parallélisme, ou à « s'approprier » des représentations adaptées à ce domaine, afin que l'utilisateur puisse facilement identifier ces représentations à son application et à ses caractéristiques intrinsèques. Nous présentons ici les plus



(a) Diagramme de Kiviat.



(b) Animation instantanée.



(c) Diagramme espace-temps.

FIG. 4.2. – *Quelques exemples de représentations spécifiques à l'évaluation de performance des applications parallèles.*

courantes (figure 4.2).

Le diagramme de Kiviat représente un certain nombre d'indicateurs pour plusieurs entités. Les valeurs de ces indicateurs sont normalisées et portées sur les

rayons d'un cercle découpé en autant de quadrants que d'entités puis ces points sont reliés, délimitant une surface donnant une indication globale. Une utilisation classique est de porter sur ce diagramme les valeurs minimum, courante et maximum d'un indicateur tel que l'utilisation du temps de calcul par chacune des tâches de l'application. Il est alors possible de voir immédiatement l'efficacité de l'application qui est d'autant plus grande que la surface correspondante est proche de celle du cercle ; les extrema de l'indicateur étant également portés sur le diagramme de Kiviat, les meilleure et plus mauvaise performances sont visibles de la même manière.

L'animation des états de l'application est une visualisation instantanée qui montre l'application parallèle ou la machine sur laquelle celle-ci s'exécute ainsi que les liens de communication entre les tâches ou les processeurs. La représentation utilisée est un graphe dont les sommets sont les tâches ou les processeurs et les arêtes les liens de communication les reliant. Les objets composant le graphe sont habituellement coloriés de manière à indiquer leur état.

Le diagramme espace-temps représente l'évolution au cours du temps de l'activité des tâches de l'application ainsi que leurs communications. À chaque tâche est associée une ligne représentant le temps, cette ligne étant interrompue lorsque la tâche ne calcule pas. Lorsque deux tâches communiquent la communication est portée sur ce diagramme en reliant les points correspondant aux instants de début et fin de la communication sur les lignes des tâches concernées. Cette visualisation est très utile à cause des informations qu'elle fournit sur le comportement de l'application. Outre son temps d'exécution (lorsque les lignes de temps n'avancent plus) l'information la plus utile concerne le schéma de communication de l'application qui apparaît sur le diagramme espace-temps. Il est ainsi très facile d'observer l'influence d'un retard ou d'une mauvaise synchronisation sur le schéma de communication attendu de l'application en le comparant avec celui donné par le diagramme.

4.1.2 Caractéristiques des visualisations

Toutes les visualisations, quel que soit leur type, ont en commun leurs caractéristiques graphiques, c'est-à-dire la manière dont les informations sont finalement affichées. En faisant varier ces paramètres il est possible de représenter plus d'informations dans une même visualisation (BROWN et HERSHBERGER 1992) ou de

modifier la manière dont l'utilisateur perçoit les informations qui lui sont présentées et les relations entre les différents objets affichés.

Paramètres pouvant porter de l'information

Dans une représentation bi-dimensionnelle telle que l'une de celles que nous venons de présenter, les caractéristiques qui peuvent être utilisées pour apporter un surcroît d'information sont au nombre de deux :

- la couleur est la première propriété qu'il est possible de changer et est de loin celle dont les modifications sont les mieux perçues par l'utilisateur : elle est utilisée par exemple pour encoder des états ou pour distinguer différents types de données ;
- il est également possible de changer la taille des objets représentés : nous pouvons par exemple tracer d'un trait plus ou moins larges les communications sur un diagramme espace-temps afin de donner une indication de la taille des messages communiqués entre les tâches.

Les représentations tridimensionnelle utilisent de plus des sources de lumière et des textures. Il serait intéressant de pouvoir associer de l'information à ces paramètres mais nous nous heurtons alors à un certain nombre de problèmes qu'il est difficile de résoudre.

Les sources de lumière ont seulement un effet esthétique : leur rôle est d'éclairer une scène afin de ne pas laisser d'objets dans l'ombre où ils seraient difficilement visibles. Même si chaque objet disposait de sa propre source de lumière ponctuelle, ou si une lumière ambiante était utilisée, la variation d'intensité lumineuse serait trop difficile à percevoir pour que cette dernière puisse être valablement associée à une information. De plus un changement de point de vue de la scène provoque une illumination différente de celle-ci du fait du déplacement des sources de lumière, et il n'est alors plus possible de savoir si un changement d'intensité lumineuse est dû à une modification de l'information associée ou au changement de perspective.

Les textures sont beaucoup plus prometteuses car elles ne sont la plupart du temps pas sensibles aux changements de point de vue. Qui plus est il existe différents types de textures (par exemple pour contrôler la matière d'un objet ou des

variations de sa surface) et chacun d'eux peut être utilisé pour représenter une information différente. Les textures sont malheureusement très difficilement exploitables à cause du coût de leur utilisation : l'ajout de textures aux objets augmente leur temps de rendu dans des proportions telles qu'il est inenvisageable de les utiliser dans une visualisation devant être mise à jour régulièrement [1]. La seule utilité que nous puissions leur trouver est une utilisation ponctuelle après avoir détecté un problème de performance grâce à une visualisation, pour produire une « photographie » de cette visualisation avec quelques informations supplémentaires. Mais même ce type d'opération risque d'avoir un coût prohibitif décourageant son utilisation.

Autres paramètres

Il est également possible de jouer sur des paramètres tels que le placement des objets dans la visualisation ou leur orientation. La plupart du temps ces caractéristiques de la représentation subissent des contraintes dues au type de la visualisation, aussi ne les associe-t-on pas à une information. Dans un diagramme espace-temps, par exemple, il est possible de modifier l'ordre relatif des lignes pour mettre deux tâches côte à côte, mais il est impossible d'aller bouger les segments représentant les communications.

Il est cependant important de pouvoir faire ce type de modification spatiale dans une visualisation afin de permettre à l'utilisateur de voir les informations comme il le désire, en regroupant des objets qu'il souhaite observer simultanément par exemple, ou en les ordonnant suivant un ordre qui lui est plus utile que celui proposé par défaut par la visualisation.

4.2 Visualisations de Scope

Les visualisations de Scope doivent offrir un certain nombre de possibilités bien définies destinées à faciliter la tâche de l'utilisateur ou à accroître ses possi-

1. Ce coût est tel que la version de RenderMan que nous utilisons pour l'affichage interactif, Quick RenderMan (UPSTILL 1992), interdit l'utilisation des textures. Elles ne sont accessibles qu'en mode photo-réaliste (génération d'image de haute qualité) lequel peut prendre plusieurs heures pour une scène complexe.

bilités de manipulation des informations présentées. Nous décrivons ici ces capacités et d'il y a lieu les mécanismes fournis par Scope pour leur utilisation. Dans ce qui suit nous utiliserons les termes « information » et « objet » de manière interchangeable, un objet dans une visualisation étant toujours en correspondance avec des informations.

4.2.1 Personnalisation

Toutes les visualisations de Scope doivent supporter la personnalisation, c'est-à-dire la modification de leurs paramètres par l'utilisateur. Cette modification est faite à travers des gestionnaires de personnalisation qui permettent à l'utilisateur de spécifier la façon dont il veut personnaliser certains paramètres, les visualisations obtenant ensuite la configuration qui leur correspond et changeant leur apparence pour s'y conformer.

Scope fournit à l'utilisateur un mécanisme de personnalisation général que nous appelons *personnalisation par associations*. Ce mécanisme est basé sur l'utilisation d'*attributs* portés par les informations et l'association de certains paramètres des visualisations à des valeurs particulières d'un attribut donné. Ces associations sont faites de manière globale et utilisées par les visualisations au moment de présenter les informations.

Les attributs sont généralement placés sur les informations par les composants qui les produisent. Par exemple les simulateurs de Scope produisent des graphes dont les constituants, sommets et arêtes, ont des attributs décrivant les informations qui leur sont associées (l'état d'un nœud, son nom, des statistiques sur son activité, etc.). De même les présentateurs d'événement des lecteurs de trace fournissent des données avec un attribut qui indique ce qu'elles représentent. Un composant d'analyse de données peut augmenter le jeu d'attributs des données qu'il manipule, ou modifier librement leurs valeurs.

L'association entre les valeurs des attributs et les paramètres des visualisations se fait par l'intermédiaire d'un gestionnaire d'associations. Son rôle est simplement de permettre à l'utilisateur de choisir les associations qu'il veut utiliser et de tenir ces associations à la disposition des visualisations. Lorsque l'une d'elle veut afficher un objet elle demande au gestionnaire approprié, pour chacun des paramètres qu'elle offre pour sa personnalisation, la valeur du paramètre étant donnés les attributs de l'objet à représenter.

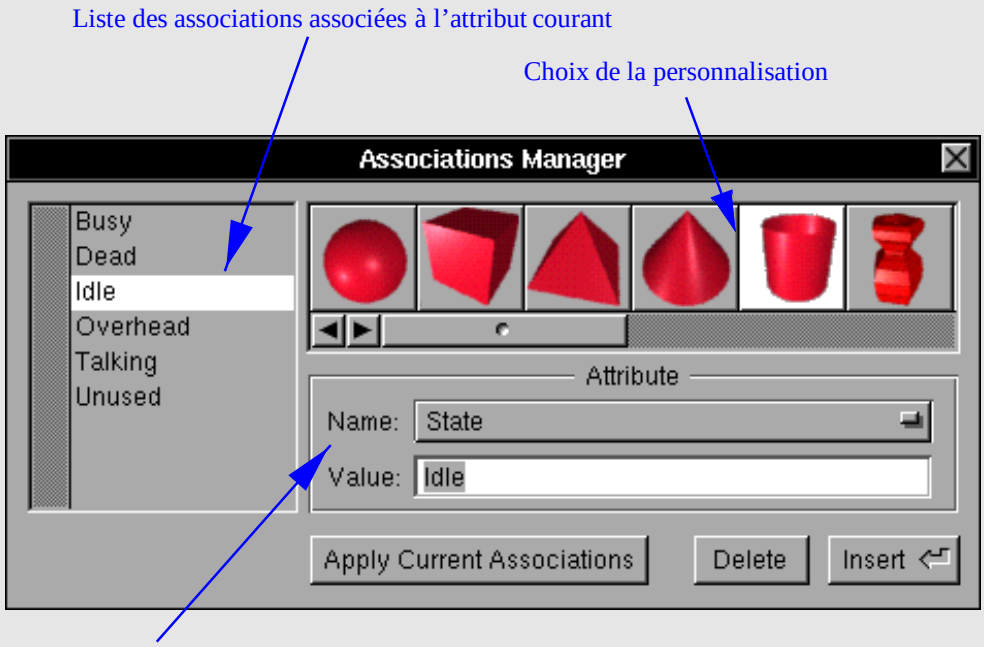


FIG. 4.3. – *Ce gestionnaire d'associations permet à l'utilisateur de spécifier quelle forme doit être utilisée pour représenter un objet en fonction de la valeur d'un de ses attributs..*

La figure 4.3 présente un exemple de gestionnaire d'associations permettant la personnalisation de formes pour des représentations tridimensionnelles. Tout objet avec un attribut ayant fait l'objet d'une telle association sera affiché avec la forme appropriée dans les visualisations se servant de ce gestionnaire. On notera que certaines formes proposées par ce gestionnaire dépendent elles-mêmes d'un autre attribut ce qui permet de générer des instances dissemblables à partir d'une forme générique.

Les autres personnalisations par associations actuellement gérées par Scope concernent la couleur et la taille des objets. La personnalisation de la couleur peut se faire de nombreuses manières, que ce soit en associant une couleur à une valeur donnée, en faisant varier un paramètre d'un modèle de couleur (par exemple une des couleurs primaires ou une de leurs couleurs complémentaires, ou encore la teinte, la saturation et la brillance, etc.) ou en associant des vecteurs de couleurs

(par exemple des dégradés) à des intervalles de valeurs. La gestion de la taille est faite en spécifiant un facteur d'échelle de l'objet, les visualisations définissant les tailles initiales des différents objets qu'elles présentent.

Les différents prototypes de gestionnaires d'associations que nous avons réalisé sont en cours d'intégration dans une interface générique qui permet à l'utilisateur d'avoir directement accès à toutes les associations d'un attribut donné. Une telle interface facilitera l'utilisation de ce mécanisme par rapport à la situation actuelle où il faut ouvrir autant de gestionnaires que de paramètres modifiables pour mettre à jour les associations d'un attribut d'un objet utilisant toutes les personnalisations offertes. Elle offrira également la possibilité de constituer un arbre d'associations, celles-ci étant héritées d'un niveau à l'autre. Il sera alors possible de définir des associations par défaut et de changer certaines d'entre elles en fonction des visualisations les utilisant.

4.2.2 Interactivité

L'interactivité d'une visualisation est définie par la façon dont elle permet la manipulation des informations qu'elle présente. Nous avons choisi dans Scope d'utiliser le paradigme de la manipulation directe, *i.e.* l'application d'un *outil de manipulation* sur les objets des visualisations, le type de l'outil déterminant l'action à effectuer en réponse à la manipulation. Toutes les visualisations de Scope doivent, dans la mesure du possible, supporter ce type d'interaction.

Le seul outil de manipulation que toutes les visualisations doivent supporter est un outil d'inspection qui, utilisé sur un objet, permet d'avoir accès aux informations sous-jacentes. Les visualisations sont libres de supporter autant d'outils de manipulation qu'elles le souhaitent, par exemple pour permettre le réarrangement des objets dans la visualisation ou pour « ouvrir » un objet complexe et présenter son contenu (COUCH 1993).

Les outils sont standardisés, c'est-à-dire que si un outil est utilisable sur plusieurs visualisations son utilisation et son comportement doivent être les mêmes pour toutes : une inspection donnera toujours accès à des informations, le déplacement d'objets se fera toujours de la même manière, etc. Les visualisations sont libres d'ajouter des possibilités à un outil, *i.e.* de faire plus en réponse à son utilisation. Mais elles ne peuvent en aucun cas limiter son utilisation ou dénaturer son usage, ceci dans un souci d'homogénéité des visualisations et pour faciliter

les actions de l'utilisateur.

L'action d'un outil de manipulation, même si son comportement est standardisé, dépend de la visualisation sur laquelle il est appliqué et Scope ne peut donc pas réellement fournir d'aide aux visualisations dans ce domaine. La seule exception à ce fait concerne les représentations tridimensionnelles que nous présentons plus loin et pour lesquelles Scope fournit quelques outils de manipulation spatiale, une telle manipulation n'étant pas dépendante des informations représentées par les visualisations.

4.2.3 Extensibilité

Les visualisations offertes à l'utilisateur doivent supporter l'augmentation du nombre d'informations qu'elles présentent avec un minimum de dégradation de leur qualité. Le risque le plus courant lorsque la quantité de données à afficher augmente est que la densité des informations dans la visualisation soit telle qu'il devienne impossible de distinguer différentes données et d'interpréter les informations de la visusalisation. Le but de la réalisation de visualisations extensibles est donc de concevoir des représentations qui sont lisibles et informatives même lorsque le nombre de données à présenter est important.

Il existe plusieurs moyens de réaliser des visualisations extensibles, plus ou moins faciles à mettre en œuvre et à exploiter :

- la méthode la plus évidente consiste en fait à utiliser un *filtrage d'informations* en amont de la visualisation pour réduire la quantité d'informations fournies cette dernière, lui permettant ainsi de présenter effectivement plus d'informations au prix d'une réduction de leur précision ;
- le *filtrage graphique* est une technique de base pour réduire la quantité d'informations présentées en ne considérant que les objets auxquels l'utilisateur s'intéresse vraiment, les autres objets étant tout simplement ignorés ;
- le *zoom structurel* ou *regroupement* d'informations (COUCH 1993, HACKSTADT et MALONY 1993, HACKSTADT *et al.* 1994) consiste à présenter plusieurs informations avec un seul objet, réduisant ainsi le nombre d'objets à afficher (toute l'information est cependant utilisée, un objet présentant une synthèse du groupe d'informations qu'il représente) ;

– les *représentations tridimensionnelles* ont une meilleure densité de représentation des informations puisque le placement des objets se fait en jouant sur trois coordonnées et non deux et parce que l'utilisateur peut changer de point de vue, autorisant la visualisation à représenter des objets cachés par d'autres du moment qu'il est possible de voir l'ensemble des informations en changeant — éventuellement plusieurs fois — la perspective.

Le filtrage d'informations est une technique efficace mais qui n'est pas à proprement parler du domaine des visualisations. C'est de plus une technique réductrice qui appauvrit les informations présentées à l'utilisateur. Si l'on souhaite que celui-ci puisse accéder à des informations complètes il faut prendre soin de conserver les données brutes associées aux données filtrées. La visualisation présentant ces dernières doit fournir un moyen d'afficher les informations d'origine lorsque l'utilisateur manipule les informations filtrées (COUCH 1993).

Si le filtrage graphique permet effectivement de réduire la quantité d'information que la visualisation doit finalement présenter ce n'est qu'un artifice visuel et cette technique, bien que parfois très utile, ne garantit pas l'extensibilité de la visualisation puisque l'utilisatuer ne peut accéder simultanément à toutes les informations s'il le souhaite.

Le regroupement est bien adapté à des visualisations telles que la représentation des activités de processeurs ou de tâches car la notion de groupe est déjà utilisée en programmation parallèle pour représenter des entités impliquées dans des activitées liées ou similaires. Associé à la possibilité d'éclater un groupe pour présenter ses constituants, cette technique permet de réaliser des visualisations extensibles et hiérarchiques, un groupe contenant des objets qui peuvent eux-mêmes être des groupes.

Les performances des stations de travail sur lesquelles l'évaluation de performance d'une application parallèle est conduite ayant augmentées de manière régulière et contenant encore à croître, la représentation tridimensionnelle des informations, qui il y a peu de temps était réservée à des matériels extrêmement coûteux et à la visualisation scientifique, est aujourd'hui à la portée de n'importe quelle application.

Nous avons choisi de développer dans un premier temps l'utilisation de ce type de représentations dans Scope en fournissant aux visualisations un cadre de manipulation de représentations tridimensionnelles et des outils associés que nous

présentons dans la section suivante.

4.2.4 Représentations tridimensionnelles

Tous les utilisateurs de la visualisation, quel que soit leur domaine, s'accordent à dire que les représentations tridimensionnelles sont intéressantes à cause de la plus grande densité d'information qu'elle sont capables d'afficher par rapport à des représentations bi-dimensionnelles de même taille. Il n'en est cependant pas moins vrai que de telles représentations sont plus difficiles à réaliser et à gérer. Nous avons donc décidé de fournir dans le cadre de Scope une aide à la réalisation des visualisations tridimensionnelles afin de faciliter leur développement et promouvoir leur utilisation.

Toutes les visualisations utilisant la représentation tridimensionnelle ont en commun la charge de la gestion de la scène dans laquelle les informations sont affichées. Dans le modèle de la représentation tridimensionnelle en informatique, nous disons que cette scène est observée par l'utilisateur à travers une caméra et suivant un certain point de vue, celui-ci étant défini par la position de l'œil de l'utilisateur et par un vecteur allant de son œil au centre de la scène et donnant donc la distance et l'angle d'observation. La position de la caméra détermine la manière dont les objets de la scène sont visibles.

L'utilisateur peut se déplacer librement par rapport au centre de la scène : il lui est possible de se rapprocher et de s'éloigner, de tourner autour de la scène ou de se déplacer sur un seul axe, décalant ainsi sa vision de la scène. Toutes ces manipulations permettent de considérer les objets représentés sous divers angles de vues et de choisir ceux qui sont visibles dans différents plans.

Ces manipulations sont essentielles pour exploiter une visualisation tridimensionnelle puisqu'il est possible, en fonction de la position de l'utilisateur, que certains objets soient cachés à sa vue. Il faut donc qu'il puisse se déplacer pour les voir. Scope fournit une *caméra virtuelle* qui permet de se déplacer librement dans la scène présentée (figure 4.4 page suivante). Son utilisation se fait par manipulation directe de la scène : l'utilisateur choisit un mode de déplacement (zoom, rotation, déplacement circulaire et déplacement axial) et agit ensuite sur la visualisation qui reflète le changement de point de vue tant que la manipulation est en cours. Il lui est également possible de combiner plusieurs types de déplacement au cours d'une même interaction.

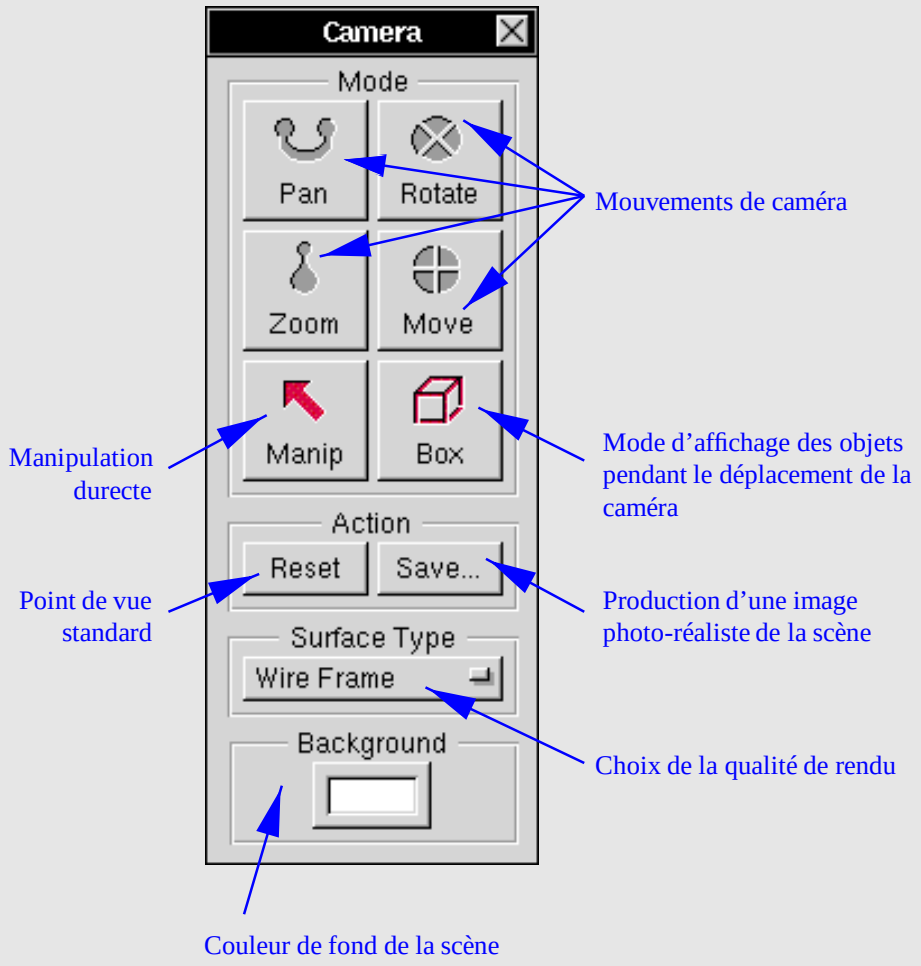
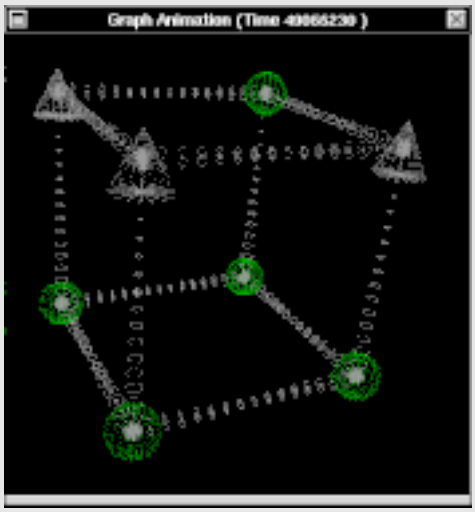


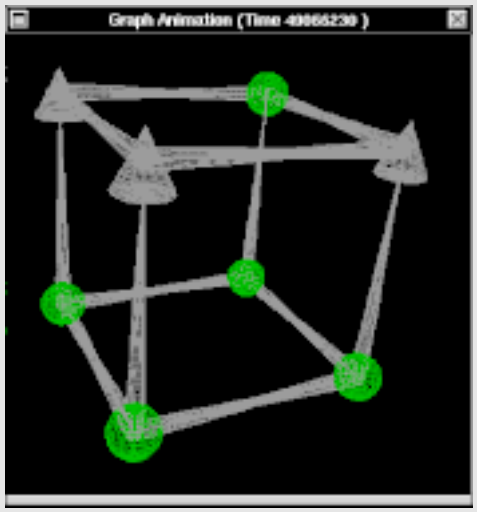
FIG. 4.4. – *La caméra virtuelle de Scope permet la manipulation interactive d'une scène tridimensionnelle ainsi que le contrôle de la qualité de son rendu.*

La caméra virtuelle de Scope est partagée entre toutes les visualisations tridimensionnelles et agit sur celle qui est en cours d'utilisation ; lorsque l'utilisateur choisit une nouvelle visualisation les paramètres de la caméra changent en fonction de celle-ci.

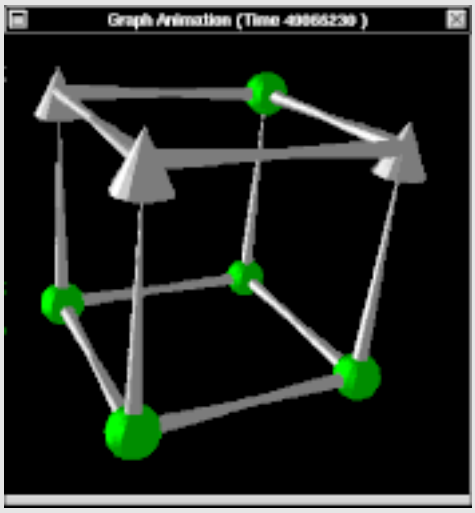
Le rendu de scènes en trois dimensions est une opération complexe et qui peut prendre beaucoup de temps. La caméra virtuelle de Scope propose donc le choix entre plusiurs qualités de rendu, allant de la représentation des seuls points défi-



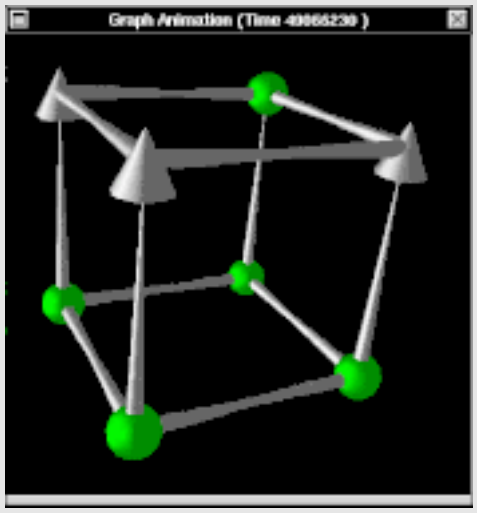
(a) Représentation en nuages de points.



(b) Représentation en fil de fer.



(c) Représentation par facettes.



(d) Représentation lissée.

FIG. 4.5. – *Les représentations tridimensionnelle peuvent être présentées avec différentes qualités de rendu et de réalisme. Sachant que le temps nécessaire au rendu d'une telle représentation est proportionnel à sa qualité il est souvent nécessaire de sacrifier celle-ci à la rapidité d'affichage.*

nissant les formes des objets de la scène au rendu lissé de haute qualité en passant par le dessin en fil de fer et le rendu par facettes (figure 4.5 page précédente). Le choix d'une moindre qualité de rendu entraîne une augmentation conséquente de la rapidité de mise à jour de la visualisation.

La qualité la mieux adaptée à un rendu rapide est celle de la représentation en fil de fer. Elle ne gère ni le rendu des surfaces ni les sources de lumières, d'où un gain de temps important, et elle est beaucoup plus précise que la représentation en nuages de points. En effet celle-ci n'affichant que les points définissant les objets, elle n'est facile à utiliser que pour des objets ayant un grand nombre de tels points, comme les sphères, le cônes et généralement toutes les surfaces de révolution ; pour un cube, par contre, seuls huit points sont affichés et l'objet est plutôt difficile à voir, rendant cette représentation délicate d'utilisation. Le rendu en fil de fer peut éventuellement éliminer les faces cachées des objets représentés pour augmenter la qualité du rendu.

Pour des raisons similaires de coût d'affichage, les déplacements de l'utilisateur dans la scène sont par défauts réalisés dans un mode de rendu minimal où seules les boîtes englobantes des objets sont visibles. Cela permet d'avoir une idée correcte de la représentation de la scène au fur et à mesure que le point de vue change sans pour autant être obligé d'attendre plusieurs secondes entre deux mouvements.

Nous avons indiqué précédemment que toutes les visualisations doivent permettre la manipulation directe des informations qu'elles présentent. Dans une représentation tridimensionnelle la difficulté principale est de trouver l'objet sur lequel l'utilisateur agit puisqu'il peut y avoir, à une même coordonnée planaire, plusieurs objets dans des plans différents. La caméra virtuelle de Scope se charge de la localisation des objets manipulés et transmet les informations voulues à la visualisation, la déchargeant ainsi de cette tâche délicate. Ceci est fait à travers un mode spécial de la caméra virtuelle, dit mode de manipulation, dans lequel la visualisation et la caméra coopérent pour réaliser la manipulation souhaitée par l'utilisateur.

Lorsque la caméra est en mode de manipulation elle agit en fait comme relais pour les outils de manipulation directe de la visualisation qu'elle contrôle (l'interface du choix de ces outils est laissée à la visualisation). Lorsque l'utilisateur effectue une action avec un tel outil la caméra localise les objets concernés et les met à la disposition de la visualisation qui est libre de les utiliser comme elle l'entend.

Ce processus est répété tant que l'utilisateur agit sur la représentation. Ce mécanisme évite aux visualisations d'avoir à se préoccuper de la position de la caméra et les décharge des transformations de coordonnées et des parcours de la scène nécessaires à l'obtention d'un objet à partir des coordonnées bi-dimensionnelles du point sur lequel l'utilisateur agit ; elles sont alors libres de se consacrer seulement aux effets de l'outil utilisé sur les objets concernés.

4.3 Visualisation de graphes

Nous présentons dans cette section une visualisation tridimensionnelle réalisée pour Scope. Cette visualisation est actuellement utilisée dans l'environnement pour présenter une animation des états de l'application au cours du temps, ces états étant produits par les simulateurs de Scope, mais n'est absolument pas dépendante du type d'informations représentés par le graphe dont elle permet la visualisation. Elle respecte les principes des visualisations de Scope que nous avons définis à savoir la personnalisation, l'interactivité et la possibilité d'extension.

4.3.1 Principe

La visualisation réalisée (figure 4.6 page suivante) utilise directement le graphe produit par le simulateur. Celui-ci contient en effet des informations sur la topologie de l'application (le graphe lui-même) et sur l'état de chacun des objets la composant (tâches et liens de communications). La visualisation présente les sommets du graphe reliés par des cônes représentant les liens de communication et leur orientation. Le simulateur n'ayant aucune raison d'associer des coordonnées tridimensionnelles aux objets qu'il manipule notre visualisation enrichit le graphe qui lui est passé en associant des enregistrements privés à chaque élément du graphe [2], ces enregistrements contenant des informations telles que l'objet apparaissant dans la représentation, ses coordonnées, etc.

2. Nous constatons là encore l'intérêt d'avoir choisi une bibliothèque de manipulation de graphes aussi puissante que celle de NORTH.

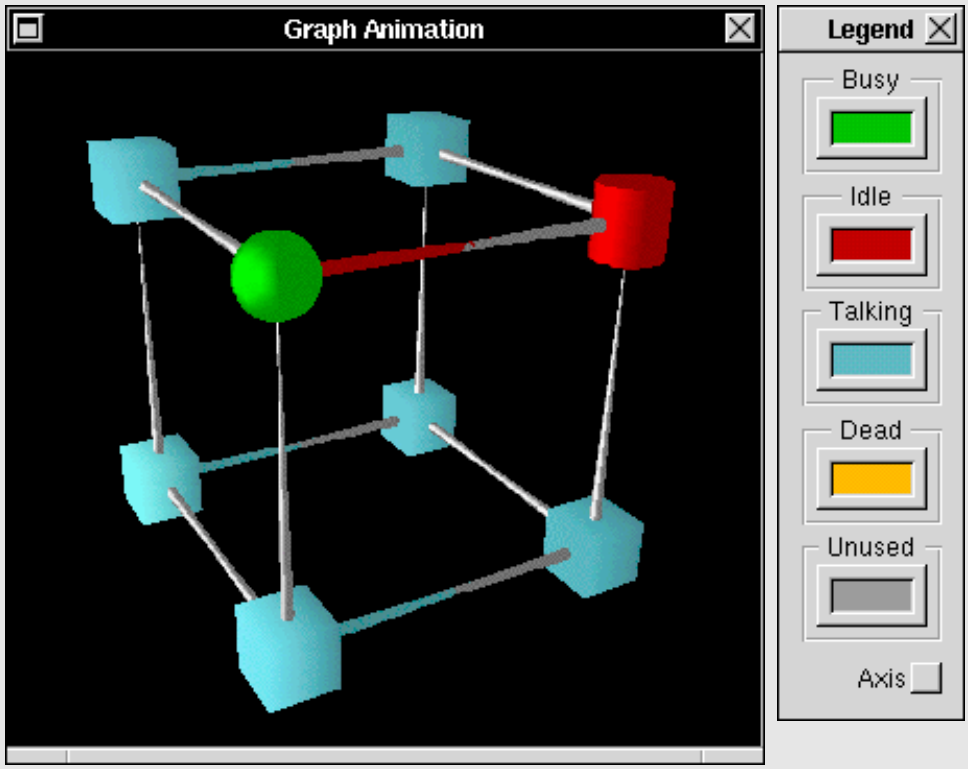


FIG. 4.6. – *Une séquence de l'animation tridimensionnelle de l'état d'une application parallèle.*

4.3.2 Personnalisation

Rappelons que les informations sur l'état de l'application sont fournies par les simulateurs de Scope comme attributs des sommets et des arêtes du graphe qu'ils produisent. La visualisation de graphe utilise directement ces attributs pour sa personnalisation : la figure 4.3 page 153 présentant le gestionnaire d'associations de formes est un exemple de cette utilisation, où l'utilisateur associe une forme différente à chacun des états que peuvent prendre les tâches et les liens de communication. Seuls les sommets du graphe supportent la personnalisation par association de formes pour garantir que l'information sur l'orientation des liens de communication donnée par la représentation conique des arêtes ne peut pas

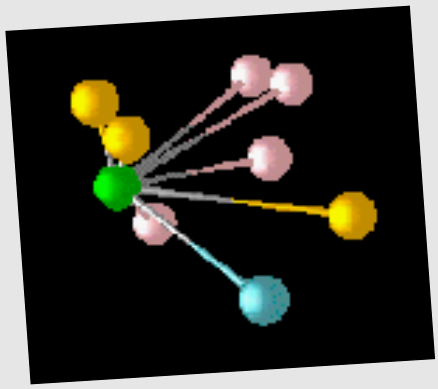
être perdue par un mauvais choix de forme. Tous les objets du graphe supportent la personnalisation de la couleur utilisée pour leur représentation ainsi que de leur taille.

L'utilisation classique des personnalisations sur une telle représentation est la suivante. La forme et la couleur des tâches sont souvent utilisées pour indiquer leur état ou leur appartenance à un groupe particulier ; la variation de leur taille peut être utilisée pour indiquer le temps spuis lequel elles sont dans leur état actuel ou pour donner une information globale telle que leur efficacité. La couleur des liens de communication est principalement utilisée pour indiquer leur état (en train de communiquer, réservés pour un échange de données ou inactifs) alors que leur taille (dans notre cas, leur diamètre) permet de coder une information telle que le volume de données circulant sur le lien. On peut ainsi afficher jusqu'à six informations différentes sur cette représentation tridimensionnelle de graphe.

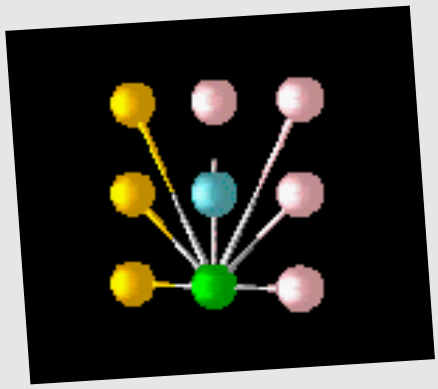
4.3.3 Placement des sommets

Même si l'utilisateur a la possibilité de se déplacer dans la scène représentant le graphe, il est souhaitable que le placement des sommets du graphe puisse être contrôlé pour qu'il corresponde au mieux à la façon dont l'utilisateur se représente la topologie de l'application et les relations entre ses tâches. La représentation développée supporte donc un certain nombre d'algorithmes de dessin de graphes permettant un placement rapide des sommets du graphe dans l'espace (figure 4.7 page suivante). Elle accepte également la spécification du placement de ces sommets par l'utilisateur : la figure 4.6 page ci-contre est un exemple d'une telle spécification pour représenter les tâches de l'application telles qu'elles sont effectivement placées sur une machine parallèle ayant un réseau de communication en forme de cube.

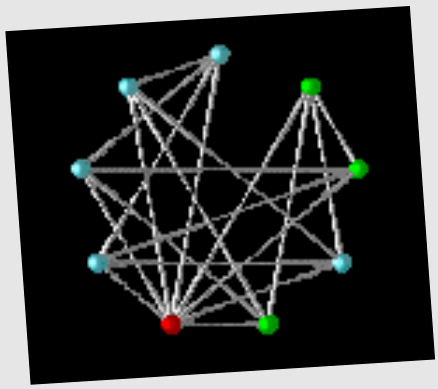
Les algorithmes de dessin offerts sont de deux types : statiques ou dynamiques. Les algorithmes statiques placent les sommets du graphe une fois pour toute alors que les algorithmes dynamiques sont invoqués chaque fois que le composant de représentation reçoit à nouveau le graphe à visualiser. Un exemple d'un tel algorithme est celui du placement après tri qui est présenté dans la figure 4.7 page suivante : les sommets du graphe sont triés suivant la valeur de deux de leurs attributs, le premier déterminant leur position dans le plan et le second, optionnel, la dernière de leurs coordonnées. Si l'on fait par exemple un tri décroissant d'après



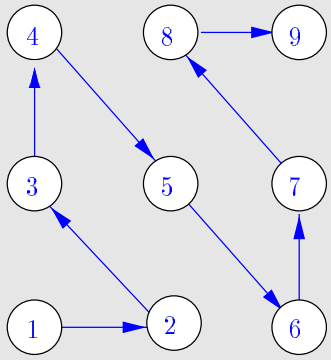
(a) Représentation naturelle.



(b) Représentation triée.



(c) Représentation naturelle.



(d) Principe du tri.

FIG. 4.7. – *Quelques manières de placer les objets du graphe en trois dimensions. Si un sommet est déplacé ses arêtes le suivent naturellement.*

la valeur de l'attribut représentant l'efficacité d'une tâche on obtient une animation où les tâches utilisant le mieux leur temps de calcul « tombent » toutes vers un même point, les autres tâches restant à la périphérie de la représentation.

Tous les algorithmes de dessin de graphe associent des enregistrements privés aux objets du graphe s'ils ont besoin de maintenir des informations durant leur exécution. Le composant de visualisation de graphe utilise ces algorithmes en res-

pectant une interface qui leur est commune et permet d'ajouter très facilement de nouveaux algorithmes de dessin de graphe. Il se charge de refléter les algorithmes utilisables en construisant dynamiquement un menu réservé à la visualisation de graphes et en installant ce menu dans l'interface de Scope.

4.3.4 Interaction

La visualisation de graphe supporte l'inspection des objets présentés par manipulation directe. Lorsque l'utilisateur agit sur un objet avec l'outil d'inspection qui lui est fourni la visualisation affiche un inspecteur listant les différents attributs de cet objet ainsi que leurs valeurs (figure 4.8 page suivante). Tant que ces inspecteurs restent visibles la visualisation met à jour leurs informations à chaque réaffichage du graphe.

L'outil d'inspection n'est absolument pas dépendant du type de l'objet inspecté : il se contente d'obtenir la liste de leurs attributs et de présenter celle-ci sous forme tabulée. Il est donc utilisable quelles que soient les informations passées à la visualisation de représentation de graphe, comme le Il peut également être réutilisé dans toute visualisation présentant des informations auxquelles des attributs ont été associés.

4.4 Conclusion

Nous avons présenté dans ce chapitre les principes de la visualisation pour l'évaluation de performance, d'abord dans de manière générale puis pour Scope. Les visualisations de Scope doivent supporter un certain nombre de capcités communes : possibilité de personnaliser leur manière de présenter les informations, support d'interactions standardisées avec l'utilisateur et manipulation directe des informations présentées, extensibilité pour être à même de traiter des volumes de données importants. Nous avons spécifié autant que possible la manière dont ces visualisations traitent ces différents points, afin de permettre le développement d'un ensemble de visualisations cohérentes dans leur interaction avec l'utilisateur. Une des tendances actuelles de la visualisation pour l'évaluation des performances est l'utilisation de représentations tridimensionnelles ; nous avons présenté les outils disponibles dans l'environnement pour faciliter la construction de

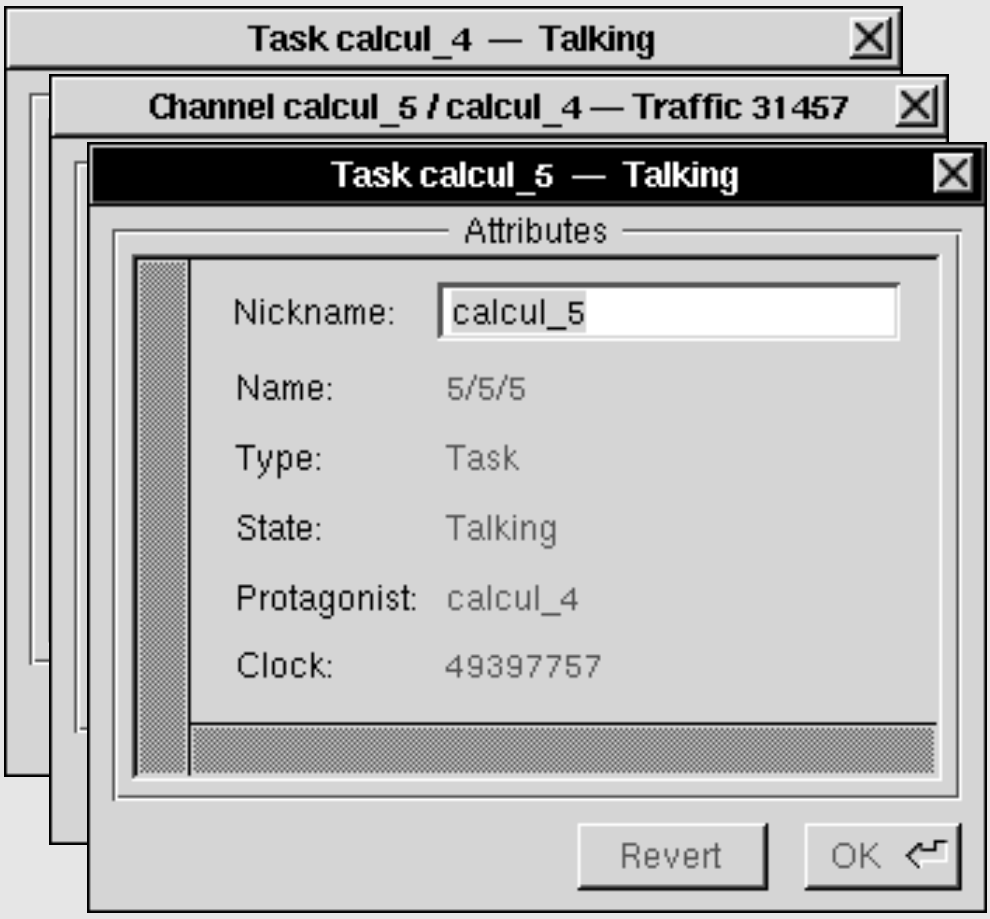


FIG. 4.8. – *Inspecteurs des objets d'un graphe représentant l'état d'une application écrite en* OUF/TRITA.

visualisations tridimensionnelles. Nous avons enfin décrit une telle visualisation, représentant un graphe en trois dimensions, et avons détaillé la manière dont elle respectait les principes des visualisations de Scope que nous avons défini en début de chapitre.

5

Schémas d'évaluation

L'ÉVALUATION de performance d'une application dans un environnement tel que Scope est faite principalement à travers la visualisation du comportement de l'application au cours de son exécution. Lors d'une session d'évaluation de performance on cherche à vérifier l'*efficacité* de l'application, *i.e* la qualité de son utilisation des ressources de la machine sur laquelle elle est exécutée. Pour pouvoir faire cette observation, c'est-à-dire construire des visualisations, il faut tout d'abord produire les données qu'elles présentent, ces données étant obtenues par analyse des informations contenues dans la trace d'exécution de l'application évaluée. C'est le traitement et la transformation des données qui nous intéresse ici.

Les performances d'une application peuvent être caractérisées par un certain nombre d'*indices de performance*, ou *indicateurs*, qui synthétisent les informations caractérisant une exécution. Ces indices de performance peuvent être très simples, comme le temps total d'exécution de l'application, ou plus complexes comme le pourcentage de temps passé par celle-ci à calculer effectivement ou la moyenne et la variance des temps de transmission de données entre différentes tâches. La difficulté de production de ces indicateurs réside non pas dans la complexité de leur calcul mais dans leur choix et la manière de les calculer afin de garantir la pertinence de l'information qu'ils apportent. Hormis quelques informations très générales il est difficile de trouver des indices de performance qui soient adaptés à toutes les applications, surtout lorsque le nombre de tâches impli-

quées augmente. Il est encore plus délicat de déterminer quels sont les indicateurs à exploiter pour obtenir rapidement une évaluation juste de la qualité des performances d'une application car ce choix dépend de la connaissance de l'application et de la machine d'exécution.

Une évaluation de performance se fait par observation simultanée de plusieurs indicateurs, chacun pouvant éventuellement être produit à partir d'autres indices de performance. Dans Scope, ces indicateurs sont calculés par un programme visuel formé de composants d'analyse de données. Nous appelons de manière générale *schéma d'analyse* le programme visuel caractérisant l'évaluation de performance qui est menée, ce schéma représentant le flot des données et leurs transformations pour passer de la trace d'exécution à la visualisation.

Nous nous intéressons dans ce chapitre au problème de la production et de l'utilisation des indices de performance. Nous présentons tout d'abord les problèmes liés aux composants d'analyse. Cette présentation est suivie d'une étude de la réalisation de deux tels composants, de l'évaluation de leur coût de développement et de leur impact sur le temps de traitement d'une trace par l'environnement et d'un exemple d'utilisation. Nous terminons par la discussion d'une classe de composants que nous appelons *composants « intelligents »* et qui sont capables de mener une action à la place de l'utilisateur.

5.1 Composants d'analyse et de visualisation

Nous venons d'évoquer les raisons pour lesquelles les composants d'analyse existent : ils calculent les indices de performance sur lesquels se base l'évaluation des performances d'une application. Sans eux l'évaluation serait limitée à la visualisation de l'évolution de l'état de l'application au cours du temps, laquelle, même si elle donne une vision qualitative de l'application, n'est pas une représentation idéale pour évaluer précisément le comportement d'une application.

Les opérations d'analyse de données susceptibles d'être utilisées dans un environnement d'évaluation de performance sont plutôt simples. Les opérations les plus courantes sont des sommes, des calculs de moyennes et de variances, des échantillonnages de données pour produire des histogrammes, etc. Il est certain que nous aurons besoin de faire appel à des techniques plus sophistiquées dans le futur, par exemple pour le traitement de données multi-dimensionnelles, mais

l'usage de techniques d'analyse de données sophistiquées nécessite tout d'abord une étude approfondie de l'apport potentiel de ces outils pour l'évaluation de performance, et nous ne nous y intéresserons pas ici.

Les composants de visualisation sont étroitement liés aux composants d'analyse car leur intérêt dépend principalement de la qualité des informations fournies par ces derniers. Ils doivent également être conçus de manière à présenter ces informations de manière intuitive et intelligente. C'est là la plus grande difficulté de la conception d'un composant de visualisation : trouver le bon compromis entre la densité d'information présentée et la simplicité d'interprétation.

Le problème qui nous occupe plus particulièrement ici est celui de l'efficacité des composants utilisés. En effet, étant donné le volume de données important à traiter au cours de l'évaluation de performance (ne serait-ce qu'à cause du nombre élevé d'événements qui peuvent être produits par une application lors de son exécution), il est primordial que le traitement de ces données ne représente pas une part trop importante du temps nécessaire au traitement de la trace par l'environnement d'évaluation de performance.

Nous décrivons ci-dessous la réalisation de deux composants. Pour chacun d'entre eux nous nous intéressons à leur raison d'être puis à la complexité de leur développement et enfin à leur coût d'utilisation en terme de temps nécessaire au traitement d'une trace par le composant.

5.1.1 Conditions d'expérience

L'évaluation du coût de chacun des composants que nous présentons a été faite de manière identique dans les conditions d'expérience suivantes.

Les mesures ont été prises sur une machine ayant pour base un processeur Intel 486DX2/66 et disposant de 16 Mo de mémoire. On notera que lors des expériences effectuées nous n'avons observé aucune pagination mémoire et que les mesures n'ont donc pas été perturbées. La machine n'était de plus soumise à aucune charge réseau.

Afin de déterminer le surcoût lié à l'utilisation d'un composant nous avons mesuré le temps nécessaire au traitement complet des traces de cinq exécution d'une application en augmentant pour chaque d'elles le nombre de communications ef-

fectuées et donc le nombre d'événements à traiter. Les deux premières mesures réalisées correspondent au temps de traitement en utilisant un schéma d'analyse comprenant simplement le simulateur adapté au modèle de programmation de la trace puis ce schéma augmenté du composant évalué, privé de son interface. Les deux dernières mesures correspondent à l'utilisation d'un schéma d'analyse comprenant et le simulateur et une visualisation textuelle de la trace [1] et d'autre part à ce même schéma augmenté du composant complet.

La première série de mesures (sans interface) permet de connaître l'augmentation de temps due aux calculs effectués par le composant pour traiter les données qu'il reçoit et générer celles qu'il propage. La seconde série (avec interface) correspond à une mise en situation réelle d'où la présence d'une visualisation, en l'occurence la plus simple — et la plus rapide — de Scope. C'est surtout dans ce cas que l'évaluation du coût de l'utilisation d'un composant est intéressante puisqu'elle donne une bonne idée des parts relatives d'une visualisation assez peu coûteuse et du composant évalué dans le temps de traitement d'une trace.

5.1.2 Construction d'un histogramme

Motivation

Il est souvent souhaitable d'avoir une vision globale de la variation d'un indicateur de performance à caractère instantané, comme par exemple le temps de communication d'un message, la durée d'une phase de calcul ou le débits d'un lien de communication. Cela permet de savoir si une application se comporte de manière régulière ou si au contraire elle est très perturbée, la variation de l'indicateur choisi étant alors importante.

L'une des méthodes envisageables pour obtenir une indication globale sans pour autant trop réduire la quantité d'information disponible est d'observer la distribution des valeurs prises par un indicateur au cours du temps, c'est-à-dire une représentation de leur répartition dans le domaine des valeurs de l'indicateur.

Pour connaître la distribution des valeurs d'une variable sur un intervalle de valeurs donné il suffit de découper cet intervalle en n sous-intervalles de longueurs

1 . Une visualisation textuelle de la trace affiche en langue naturelle les événements fournis par le lecteur.

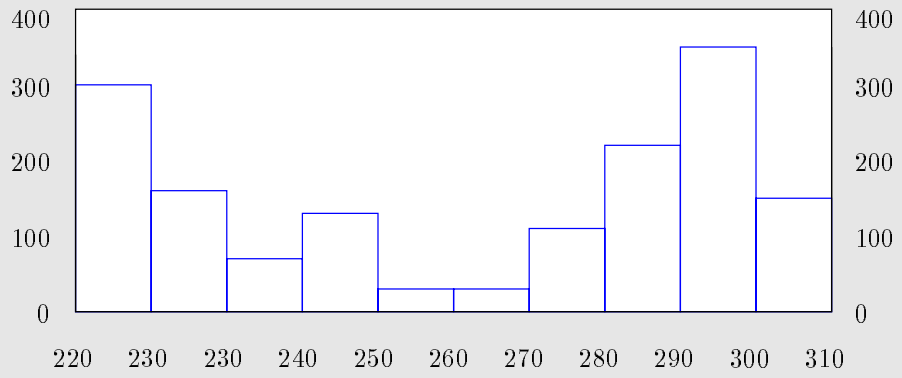


FIG. 5.1. – *Représentation de la distribution d'une variable sur une intervalle de valeurs. L'axe des abscisses représente le découpage de l'intervalle. Chaque barre de l'histogramme indique par sa hauteur le nombre de fois où la variable a pris une valeur dans le sous-intervalle correspondant.*

égales puis de compter pour chacun d'entre eux le nombre de fois où la variable observée a pris une valeur dans celui-ci. Le résultat de cette opération d'échantillonnage est un *histogramme* de la distribution des valeurs observées dans l'intervalle choisi (figure 5.1).

Réalisation

Nous avons réalisé un composant d'analyse de données produisant un histogramme de la distribution d'une variable sur un intervalle.

Ce composant a pour paramètres les bornes de l'intervalle de valeurs à considérer et le nombre de sous-intervalles servant à l'échantillonnage des données. Il réserve en outre deux sous-intervalles pour les valeurs hors-limites, *i.e.* hors des bornes de l'intervalle d'échantillonnage. Ces sous-intervalles, de longueur infinie, permettent d'obtenir un histogramme complet même si l'intervalle des valeurs initialement choisi n'inclut pas toutes les valeurs observées. Le compte des valeurs dans ces sous-intervalles hors-limites donne également une bonne indication de la validité de ce choix : s'il est faible alors le reste de l'histogramme représente bien ce que l'on souhaitait observer.

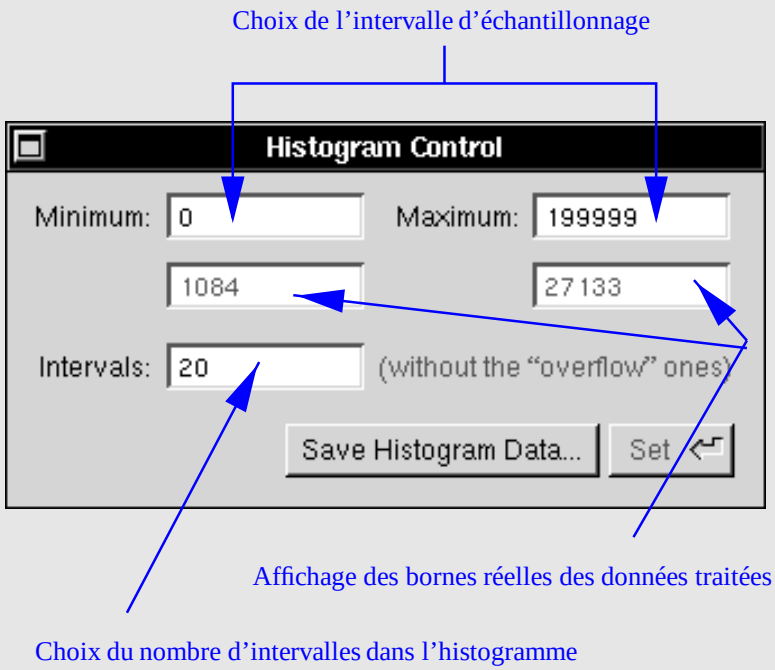


FIG. 5.2. – *L'interface du composant de construction d'histogramme permet de choisir les bornes de l'intervalle considéré et le nombre de sous-intervalles souhaités. Elle sert également au composant pour afficher les bornes réelles des données manipulées.*

Chaque fois que le composant reçoit une donnée, il incrémente le compteur associé au sous-intervalle auquel la donnée appartient et produit le nouvel histogramme sur un port de sortie. Il calcule également les bornes effectives de l'intervalle des valeurs des données reçues, lesquelles sont également disponibles sur des ports de sortie du composant.

L'interface de ce composant d'analyse (figure 5.2) permet à l'utilisateur de définir l'intervalle d'échantillonnage ainsi que le nombre de sous-intervalles composant l'histogramme. Elle affiche également les bornes effectives calculées pour information et permet enfin d'enregistrer l'histogramme produit par le composant afin de pouvoir l'exploiter avec des applications externes, que ce soit pour effectuer de nouvelles manipulations ou simplement pour pouvoir inclure ces données dans un document.

Évaluation

Le code du composant développé est composé d'un peu moins d'une centaine d'instructions Objective C, la moitié au moins d'entre elles étant consacrée à la gestion de l'interface et de la sauvegarde de l'histogramme pour usage externe. Ce n'est vraiment pas ce que l'on peut appeler un code important...

Le composant est générique car il accepte des données sans se soucier de leur sémantique. Il est également polymorphique et traite indifféremment des nombres entiers ou réels. Il supporte enfin un nombre quelconque de sous-intervalles dans la limite de la mémoire disponible pour leur allocation.

Le tableau 5.1 page suivante présente les mesures de temps de traitement d'une trace par l'environnement, avec et sans le composant de construction d'histogramme. On notera que les temps pour des petites tailles de traces sont moins précis que les autres à cause de la rapidité du traitement. Si nous considérons donc les plus grandes tailles de traces nous constatons que l'augmentation du temps de traitement est un peu supérieure à 5 % dans l'étude de cas sans interfaces et inférieure à 1 % dans le second cas. La raison en est que le coût de l'affichage textuel des événements est bien plus important que celui de la mise à jour des extrema des données dans l'interface de notre composant. Étant donné que la seconde expérience correspond aux conditions réelles d'utilisation de l'environnement, c'est-à-dire avec au moins une visualisation, nous pouvons affirmer que le composant de production d'histogramme a un coût d'utilisation presque nul.

5.1.3 Calcul du surcoût lié aux échanges de données

Motivation

Les applications parallèles sont développées afin de gagner du temps par rapport aux implémentations séquentielles. Le principal souci concernant ces applications est leur efficacité, *i.e.* la manière dont elles utilisent les processeurs de calcul des machines parallèles. L'augmentation de cette efficacité est intéressante sur deux plans : le coût d'utilisation de la machine parallèle pour la résolution d'un problème donné diminue puisqu'elle est utilisée moins longtemps ; ensuite l'application peut, à temps d'exécution constant, traiter des problèmes de taille plus importante ou avec une plus grande finesse de modélisation.

Taille	Simulateur	Sim. et hist.	Sim. et texte	Tous comp.
1	0,309996	0,379987	4,549883	4,789872
2	0,549973	0,589983	8,839743	8,889759
4	0,879970	0,9434592	17,169513	17,294851
16	3,469866	3,652728	66,938044	67,478061
32	6,989725	7,369715	134,066081	135,176247

TAB. 5.1. – *Temps de traitement de traces dans différentes conditions (de gauche à droite : avec juste un simulateur, avec un simulateur et le composant de construction d'histogramme sans interface, avec un simulateur et un composant de visualisation textuelle de la trace, avec ces deux composants et le composant de construction d'histogramme complet. Les temps sont exprimés en secondes.*

Nous pouvons distinguer deux activités principales dans une application parallèle : le calcul ou traitement des données et l'échange de données entre les tâches de l'application. Le temps consacré au mouvement de données entre les processus lorsqu'ils ne calculent pas est appelé *surcoût lié aux échanges de données*. L'efficacité de l'application peut être définie par le rapport entre le temps de calcul (temps d'exécution diminué de ce surcoût) et le temps d'exécution. La recherche d'une efficacité maximale passe donc par la minimisation du surcoût lié aux échanges de données.

Les mesures permettant d'obtenir ce surcoût sont prises par échantillonnage : à des instants donnés nous calculons le temps cumulé consacré par l'application aux échanges de données et divisons ce temps par le temps d'exécution cumulé des différentes tâches. Le rapport obtenu exprimé sous forme de pourcentage indique le temps « perdu » par les tâches de l'application à s'échanger des données. Le choix des instants échantillonnage détermine la fréquence avec laquelle cette information est mise à jour mais également sa qualité. Deux exemples de tels choix sont ceux de l'échantillonnage par intervalles de temps (figure 5.3 page ci-contre) et de l'échantillonnage par phases (figure 5.4 page suivante). La première méthode garantit que des mesures seront produites à intervalles réguliers ; elle donnera des valeurs importantes lorsque toutes les tâches ont été bloquées en attente de données pendant un temps couvrant un intervalle d'échantillonnage complet. La seconde ne calcule le surcoût lié aux échanges de données qu'à la fin d'une phase de communication et de calcul : les valeurs produites auront donc

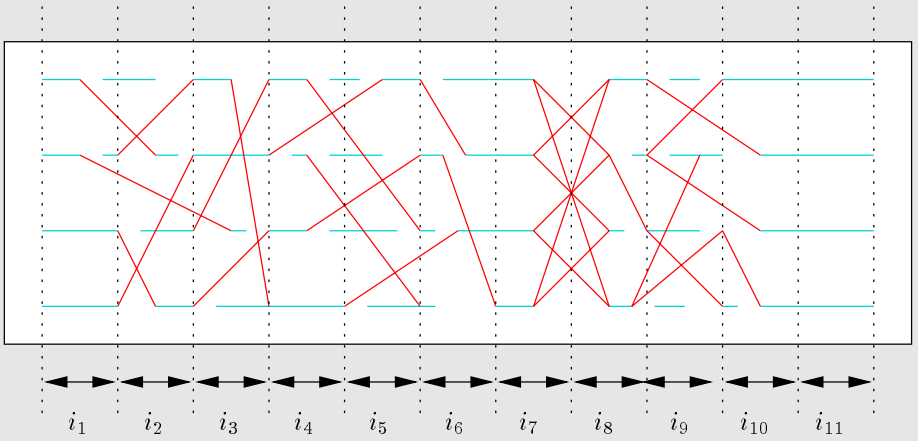


FIG. 5.3. – *L'échantillonnage par intervalles de temps consiste à prendre des mesures toutes les n unités de temps. Son inconvénient est d'avoir tendance à exagérer le surcoût lié aux échanges de données (cette exagération diminuant cependant au cours du temps).*

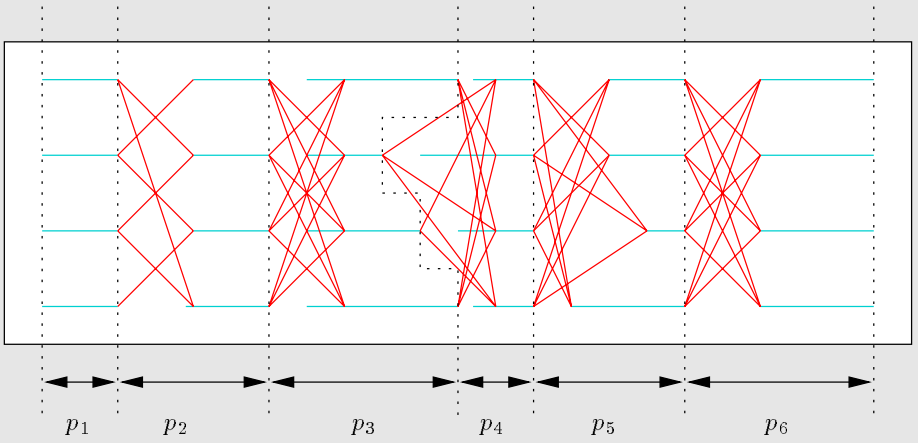


FIG. 5.4. – *L'échantillonnage par phase est celui qui donne les meilleures informations. Il consiste à prendre des mesures chaque fois que toutes les tâches de l'application ont fini de recevoir leurs données et de travailler dessus.*

tendance à mieux représenter l'évolution du temps consacré aux échanges de données même si la fréquence de mise à jour de l'indicateur peut être moindre que dans le cas de l'échantillonnage par intervalles de temps.

Réalisation

Le composant dont nous décrivons ici la réalisation est un « composant intégré » se chargeant d'effectuer une transformation de données mais aussi de présenter les informations qu'il produit par l'intermédiaire d'une visualisation. Nous l'avons conçu ainsi afin d'évaluer le coût de la réalisation d'un composant un tant soit peu complexe.

Ce composant travaille sur des événements tels qu'ils sont présentés par la partie spécifique au modèle de programmation de l'environnement Scope. Les événements qui lui sont fournis sont les événements de communication lesquels comportent d'une part une durée indiquant le temps consacré par l'application à la communication et d'autre part un indicateur permettant de savoir si l'événement représente le début ou la fin d'une communication. En plus de ces informations le composant utilise la date de l'événement et l'identification de l'entité de l'application (*e.g.* une tâche) l'ayant produit.

Le calcul du surcoût lié aux communications peut se faire suivant six méthodes : par intervalles de temps, toutes les n fins de réception (en comptant celles-ci soit globalement soit pour chaque tâche), toutes les n fins de phases (en considérant le surcoût cumulé sur ces dernières phases ou sur la totalité des phases depuis le début de l'exécution) ou une seule fois à la fin du traitement de l'ensemble des données. Le composant calcule également les minimum et maximum des valeurs du surcoût lié au communications.

En plus de ces indicateurs le composant produit un certain nombre d'autres informations. Ce sont tout d'abord les comptages d'envois de données effectués par les tâches (que ces envois soient terminés ou non) et le nombre de fois où elles ont effectivement reçu des données, ces informations permettant de déterminer le nombre d'échanges de données en cours et de s'apercevoir que des tâches prennent du retard en ne réceptionnant pas assez rapidement les données qui leur sont destinées. C'est également un comptage du nombre de phases échange-calcul qui ont eu lieu depuis le début du traitement des données. Le composant tient également à jour un indicateur de tendance de la valeur du surcoût lié aux échanges

de données : cet indicateur donne le sens de variation actuel du surcoût — en hausse ou en baisse — ainsi qu'un comptage indiquant depuis combien de temps (en nombre d'échantillonnages) il varie dans ce sens.

L'ensemble des données calculées est affiché pour information dans l'interface du composant (figure 5.5 page suivante). Le comptage du nombre de phases n'est mis à jour dans l'interface que lorsqu'un échantillonnage a lieu alors que les comptages de début et de fin d'échanges de données sont affichés dès qu'ils changent. Au moment de l'échantillonnage le nombre d'échanges de données terminés, *i.e.* la valeur du comptage de fins de réception de données, de même que les informations de l'indicateur de tendance, sont mises à jour.

La partie visualisation du composant se présente sous la forme d'une *jauge* affichant les extrema et la valeur courante de l'indicateur de surcoût. Après chaque mise à jour de cet indicateur le composant passe sa valeur courante à la jauge qui calcule elle-même les extrema (la raison en est que la jauge est une représentation générique qui peut être utilisée dans de nombreuses visualisations et ne doit donc ni accéder directement à des informations sur les données qu'elle doit représenter ni exiger des informations simples qu'elle est capable de calculer d'elle-même, afin d'être réutilisable sans contraintes). Elle affiche ensuite ces extrema et la valeur courante de sa donnée sous forme d'une barre découpée en sections de couleurs différentes en fonction de ce qui est représenté (le minimum, la valeur courante ou le maximum).

Évaluation

Le composant est fait avec deux classes, l'une pour la jauge de visualisation et l'autre pour le composant proprement dit c'est-à-dire le calcul des indicateurs et autres informations, la gestion de l'interface et la mise à jour de l'affichage. Le code propre au composant représente deux cents instructions Objective C, la plupart d'entre elles se chargeant de l'interface ou de la gestion mémoire assez complexe utilisée pour garantir l'extensibilité du composant. Le code de la jauge, quant à lui, fait moins de cent cinquante instructions dont une cinquantaine pour la seule gestion de la personnalisation des couleurs par glisser-jeter de couleurs sur les sections de la jauge. Nous pensons que cela représente un coût de développement plutôt faible étant donné ce que réalise le composant.

Le composant peut être considéré comme étant générique au sens où il ne

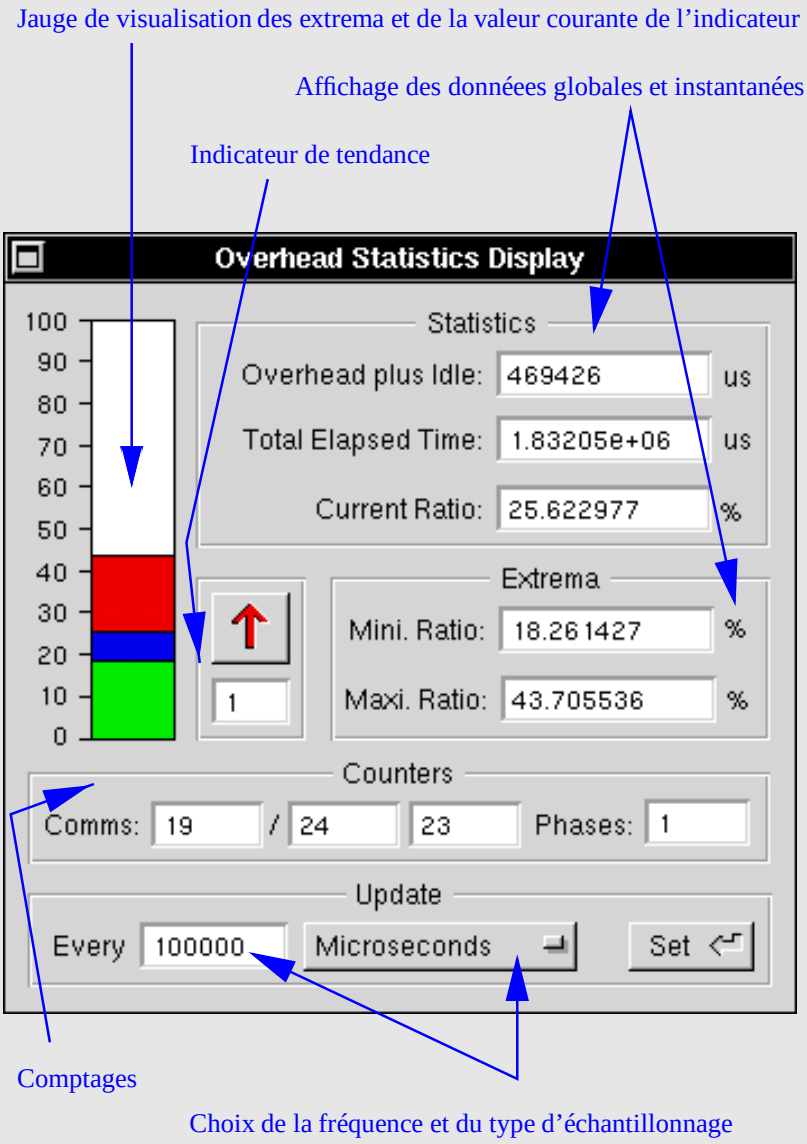


FIG. 5.5. – *L'interface du composant de calcul du surcoût lié aux échanges de données fournit des indications à caractères instantané et global. Les informations sont présentéees sous formes textuelles et graphiques. Cette interface permet également de choisir le type et la fréquence de l'échantillonnage de l'indicateur de surcoût.*

dépend pas du format des événements mais seulement d'un certain nombre d'informations associées à chaque événement. Il est de plus facile de produire une version épurée de ce composant qui soit réutilisable pour la visualisation d'indicateurs différents, en le transformant par exemple pour qu'il reçoive une donnée, affiche celle-ci et les valeurs extrêmes reçues et conserve la jauge et l'indicateur de tendance. Dans sa forme actuelle (avec échantillonnage et partitionnement par entités d'exécution) le composant est extensible et supporte la tenue de statistiques pour un nombre de tâches quelconque. Il est de plus conçu pour qu'il soit facile, le cas échéant, d'ajouter de nouvelles méthodes d'échantillonnage.

La jauge servant à représenter des informations est réellement générique mais également polymorphique par son acceptation de nombres entiers ou réels. Elle est de plus personnalisable et accepte le choix des couleurs utilisées pour la représentation des différentes sections de la jauge par un simple glisser-jeter de couleurs sur la section dont l'utilisateur veut changer l'apparence. Elle supporte enfin la suppression sélective des différentes sections pour ne représenter par exemple que la valeur courante d'un indicateur.

Le lecteur notera que le problème de ne maintenir des statistiques que pour certaines tâches de l'application n'est pas du ressort du composant développé. Il suffit en effet de placer, entre les présentateurs d'événements et le composant, un composant de filtrage ne propageant que les événements de certaines tâches pour que notre composant ne calcule plus que les statistiques voulues.

Le tableau 5.2 page suivante présente les mesures du temps de traitement de nos traces de test réalisées avec ou sans le composant de calcul de surcoût. En gardant à l'esprit les remarques du § 5.1.2 page 173, nous retenons que le coût d'utilisation en conditions normales de notre composant avec son interface est de l'ordre de 3 % du temps de traitement, ce qui est assez faible. Cette proportion diminuera d'ailleurs avec l'ajout d'autres visualisations dans le schéma d'évaluation et sera réellement négligeable si ce dernier comporte des visualisations complexes telles que des représentations tridimensionnelles.

5.2 Schémas et composants intelligents

Le nombre de problèmes de performance que l'on peut détecter est fonction du nombre d'indicateurs distincts caractérisant différents facteurs de la perfor-

Taille	Simulateur	Composant	Sim. et texte	Tous comp.
1	0,309996	0,369988	4,549883	4,559872
2	0,549973	0,659975	8,839743	9,039766
4	0,879970	0,973512	17,169513	17,629521
16	3,469866	3,829858	66,938044	69,028126
32	6,989725	7,579701	134,066081	138,516223

TAB. 5.2. – *Temps de traitement de traces dans différentes conditions (de gauche à droite : avec juste un simulateur, avec un simulateur et le composant de calcul de surcoût sans interface, avec un simulateur et un composant de visualisation textuelle de la trace, avec ces deux composants et le composant de calcul de surcoût complet. Les temps sont exprimés en secondes.*

mance d'une exécution. Il faut cependant utiliser également des indicateurs qui se recoupent, pour confirmer les observations effectuées mais aussi parce que cela minimise les chances qu'un problème passe inaperçu.

Que ce soit à cause de la multiplicité des sources d'informations utilisées ou simplement à cause de la taille des problèmes analysés, l'utilisateur peut se retrouver rapidement en présence d'un volume d'informations tellement important qu'il devient difficile de l'analyser et surtour de l'utiliser pour localiser des problèmes de performance. L'utilisateur effectue également un certain nombre de tâches identiques au cours de ses différentes sessions d'évaluation de performance, comme par exemple chercher à détecter un type de problème particulier par l'observation de certains indicateurs.

Dans ces deux cas il est souhaitable que l'environnement puisse assister l'utilisateur dans sa tâche non pas par le biais des visualisations mais en étant capable d'agir en lieu et place de l'utilisateur dans certaines conditions. Dans Scope cette aide est apportée par l'utilisation de composants spécialisés capables de contrôler l'environnement. Nous appelons *schéma intelligent* un schéma d'analyse dont une partie a pour rôle d'agir pour l'utilisateur. Par abus de langage nous parlerons également de *composants intelligents* en nous référant aux composants acteurs d'un tel schéma.

5.2.1 Principes

Le composant de contrôle des sessions de Scope (cf. § 1.6 page 80) possède un port de contrôle qui permet à d'autres composants de le piloter. Toutes les actions du panneau de contrôle de Scope sont ainsi accessibles à un composant par simple envoi d'une commande au composant de contrôle des sessions.

La partie d'un schéma intelligent — ou d'un simple composant intelligent — qui contrôle Scope pour l'utilisateur est donc très facile à réaliser : il suffit simplement d'émettre une commande appropriée sur un port de sortie connecté au port de contrôle du composant de contrôle des sessions. Ce port étant asynchrone cette commande est immédiatement reçue et l'action désirée est aussitôt réalisée. Ce type de support permet déjà de réaliser des schémas qui surveillent l'apparition d'un problème de performance et, dès sa détection, préviennent l'utilisateur après avoir par exemple arrêter le déroulement de la trace. Avoir une telle possibilité permet de se décharger de la tâche de surveillance des informations qui représente la plus grande partie du temps consacré à l'évaluation de performance d'une application, au moins durant la phase de recherche de problèmes.

Il n'est pas nécessaire de réaliser soi-même un composant spécialisé pour pouvoir contrôler l'environnement. Scope fournit en effet un composant générique qui émet une commande de contrôle définie par l'utilisateur dès qu'il reçoit une donnée sur son unique port d'entrée, sans réaliser aucun calcul. Il suffit de connecter une instance de ce composant à un composant produisant une donnée suivant certaines conditions (les composants réalisant des structures de contrôle conditionnelles de type « si ... alors ... sinon ... » étant particulièrement utiles ici) et l'action voulue sera effectuée lorsque chaque fois que la condition en question sera réalisée.

D'autres types d'actions intelligentes manipulant l'environnement sont également envisageables. Un composant peut par exemple modifier dynamiquement le schéma d'analyse utilisé pour l'évaluation de performance d'une application pour ajouter des visualisations adaptées à l'investigation d'un problème particulier après sa détection. Il peut aussi demander à Scope de mettre en avant une visualisation particulière si celle-ci est présente dans le schéma d'analyse utilisé pour l'évaluation de performance en cours. Il est également possible de déclencher différents outils, éventuellement externes, après avoir fait une sélection des informations pouvant aider à trouver l'origine du problème détecté.

Étant donné la facilité de développement d'un nouveau composant utilisable dans Scope il est aisé d'ajouter de nouvelles capacités « intelligentes » à l'environnement. Les commandes passées au composant de contrôle des sessions étant des objets elles peuvent contenir des informations précisant l'action à effectuer. Cela simplifie également la création de nouveaux types de commandes de contrôle de l'environnement. Bien qu'actuellement de tels ajouts nécessitent la regénération du composant de contrôle des sessions afin qu'il soit capable d'interpréter les nouveaux types de commandes, un mécanisme d'ajout dynamique de commandes est à l'étude. Ce mécanisme permettra l'enregistrement d'objets capables de traiter certaines classes de commandes, ces objets ayant un accès privilégié aux informations de l'environnement pendant l'exécution des commandes.

5.2.2 Surveillance de données

Pour illustrer nos propos nous avons réalisé un composant de surveillance de données capable d'agir en cas de problème.

Ce composant vérifie si les données qu'il reçoit sont dans un intervalle choisi par l'utilisateur. Ce dernier spécifie également le comportement du composant lorsqu'une donnée invalide est reçue (figure 5.6 page ci-contre). Quand cela arrive le composant effectue l'action désirée et, s'il est autorisé à avertir l'utilisateur, affiche un panneau indiquant son action et sa raison (figure 5.7 page 184).

Cette réalisation est extrêmement simple et le code du composant est très compact : une cinquantaine d'instructions Objective C dont moins d'une dizaine pour la surveillance de données qui se résume à des comparaisons entre la valeur de la donnée reçue et les bornes de l'intervalle, puis éventuellement à l'envoi d'une commande de contrôle sur le port de sortie du composant.

5.3 Conclusion

Nous avons présenté dans ce chapitre les principes directeurs des composants d'évaluation de performance dans Scope. Cette présentation a été faite à travers la description de la réalisation de différents types de composants après en avoir justifié le besoin. Ces composants ont été développés de manière à respecter les

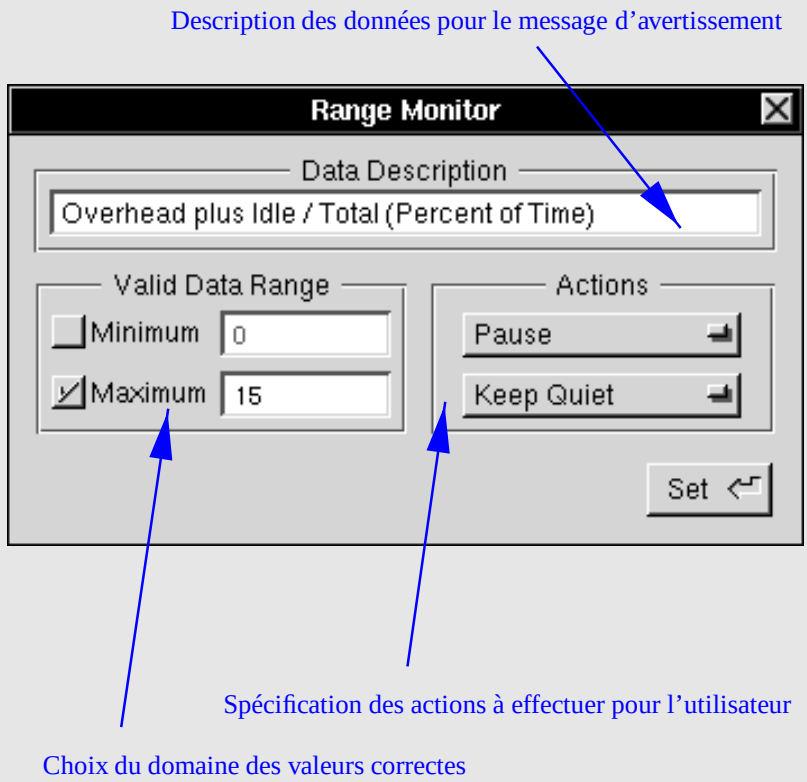


FIG. 5.6. – *L'interface du composant de surveillance de données permet de spécifier l'intervalle des données valides et les actions du composant lorsqu'il reçoit une donnée incorrecte (l'action du haut détermine la commande émise et celle du bas la manière dont le composant avertit l'utilisateur du problème).*

objectifs de Scope c'est-à-dire la généricité, l'extensibilité, la facilité de personnalisation mais aussi l'agrément et la souplesse d'utilisation ainsi que la facilité de réutilisation. L'impact de l'utilisation des composants sur le temps de traitement d'une trace par l'environnement a été mesuré afin de donner une bonne idée de ce que coûte leur présence, ces mesures ayant mis en évidence l'efficacité de ces composants et leur faible usage des ressources du système sur lequel Scope est exécuté. Une évaluation de leur coût de développement a mis en évidence la simplicité de la réalisation de nouveaux composants pour Scope.

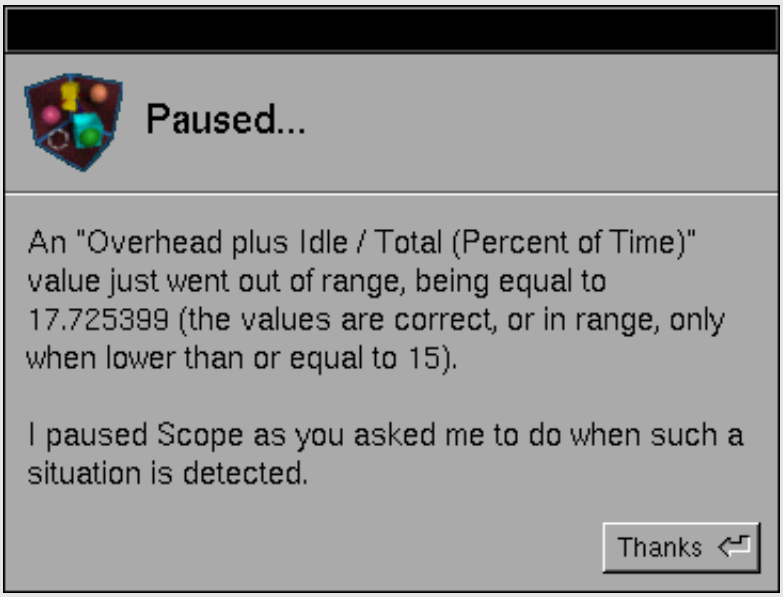


FIG. 5.7. – *Lorsqu'une donnée incorrecte est reçue par le composant de surveillance de données celui-ci effectue l'action voulue et avertit éventuellement l'utilisateur par un panneau similaire à celui-ci.*

6

Utilisation de l'environnement

LES CHAPITRES précédents ont présenté en détail la conception et la réalisation de l'environnement Scope. Le chapitre précédent, consacré aux schémas d'analyse, nous a en particulier permis de démontrer la facilité de réalisation de composants d'évaluation de performance destinés à être intégrés dans un programme visuel d'exploitation de traces d'exécution.

Nous nous intéressons dans ce chapitre à deux exemples de l'exploitation des composants introduits dans le chapitre précédent pour l'évaluation de performance d'une application réelle. Après avoir présenté l'application parallèle que nous avons choisi d'évaluer ainsi que les objectifs de performance à atteindre, nous décrivons la manière dont nous avons utilisé les composants de production d'histogramme, de calcul du surcoût lié aux communications et de surveillance de données pour mener à bien l'évaluation de performance de notre application.

6.1 Présentation

L'application choisie pour illustrer l'utilisation des composants présentés dans ce chapitre calcule une transformée de Fourier bi-dimensionnelle. Cette application est réalisée suivant le modèle SPMD (*Single Program Multiple Data*) en utilisant la bibliothèque de programmation parallèle et d'envoi de messages PVM

$r \leftarrow n/p$ — Nombre de lignes par tâche.
$l \leftarrow n/s$ — Nombre d'étapes.

fft_monodim_seq$(X[0][], n)$

pour $i \leftarrow 1$ **jusqu'à** $l - 1$
faire
    transpose$(X[(i-1) \cdot s][], X^t[(i-1) \cdot s][])$
    fft_monodim_seq$(X[i \cdot s][], n)$
fin faire

transpose$(X[(l \cdot s][], X^t[(l-1) \cdot s][])$
pour $i \leftarrow 1$ **jusqu'à** $r - 1$
faire
    fft_monodim_seq$(X^t[i][], n)$
fin faire

FIG. 6.1. – *Algorithme parallèle de la transformée de Fourier rapide par la méthode* transpose-split *avec recouvrement des communications par le calcul. Les paramètres de l'algorithme sont : la matrice X, sa taille n, le nombre de tâches de calcul p et le nombre de lignes des blocs de données s.*

(cf. § A.1.2 page 235).

L'algorithme utilisé est celui de la méthode *transpose-split* avec recouvrement des communications par les calculs (CALVIN 1995). Le principe de cet algorithme est de recouvrir la phase de transposition de la matrice, qui nécessite des communications entre les tâches, par le calcul local des transformées de Fourier mono-dimensionnelles. En effet il est possible de communiquer un bloc de s lignes dès qu'il a été calculé tout en calculant simultanément une transformée de Fourier mono-dimensionnelle sur un autre bloc de lignes, le temps de communication étant alors mis à profit pour effectuer ce calcul. Cet algorithme est présenté dans la figure 6.1.

Le but de cet algorithme avec recouvrement est de diminuer le temps d'exé-

cution de l'application en maximisant l'utilisation des processeurs de calcul de la machine. Si les communications sont bien recouvertes par les calculs alors l'algorithme sera très efficace et l'application passera la presque totalité de son temps à calculer, le surcoût lié aux communications étant presque nul. Dans le cas contraire toutes les tâches de l'application perdent du temps lors de la transposition de la matrice à cause du temps passé à attendre les données dont elles ont besoin.

La difficulté de la mise au point de l'optimisation d'une application utilisant cet algorithme consiste à déterminer la valeur optimale de la taille de bloc s pour une taille de matrice et un nombre de tâches donnés. Cette taille de bloc dépend non seulement de ces paramètres mais également de la vitesse de calcul des processeurs et des qualités du réseau de communication de la machine sur laquelle l'application est exécutée puisque le temps de communication à recouvrir est fonction de la rapidité de transfert des données entre les tâches de l'application. Le problème que l'on rencontre alors est, étant donné qu'il est rare d'avoir un modèle correct du comportement de ce réseau, qu'il n'est pratiquement pas possible de déterminer la meilleure taille de bloc *a priori* : une taille de bloc très satisfaisante sur une machine donnée peut se révéler très mal adaptée à une autre machine.

Nous présentons ci-dessous deux études de performance sur l'application que nous venons de décrire, études réalisées avec les composants que nous avons décrits dans ce chapitre et en démontrant l'exploitation.

6.2 Conditions d'expérience

L'application a été exécutée sur une machine parallèle IBM SP1 en utilisant le réseau de communication rapide de cette dernière. Les traces d'exécution ont été prises avec le traceur logiciel TAPE/PVM.

La taille de la matrice a été fixée à 256 éléments (des réels double précision) et le nombre de tâches à 4. Nous avons exécuté l'application pour des tailles de blocs s de 64, 32, 16, 4 et 2 lignes. Chaque exécution a fait l'objet d'une synchronisation des horloges au début de la prise de trace afin de garantir une datation correcte des événements. On notera également que les processeurs de la machine n'étaient exploités que par cette application lors de la génération des traces, et que chaque tâche était sur un processeur différent, ce qui signifie que l'exécution de

l'application n'a pas été perturbée par l'ordonnancement de tâches concurrentes.

6.3 Histogramme des temps d'attentes

L'algorithme de calcul d'une transformée de Fourier bi-dimensionnelle que nous utilisons est un algorithme très régulier et l'application évaluée étant de type SPMD nous avons la garantie que toutes les tâches exécutent exactement le même code même si leurs données sont différentes. Le comportement de chacune de ces tâches est une succession de phases (calcul séquentiel d'une transformée de Fourier mono-dimensionnelle puis échange de données pour la transposition de la matrice) terminée par une série de calculs. Théoriquement les temps de calcul et de communication sont les mêmes pour toutes les tâches.

Un moyen de vérifier le bon déroulement de l'exécution est de regarder l'évolution des temps d'attentes de réception de messages au cours du temps. Si tout se passe bien tous les temps d'attentes devraient être sensiblements égaux, alors que s'ils varient beaucoup nous nous trouvons face à un problème de performance.

Nous avons donc construit, à l'aide du composant présenté au § 5.1.2 page 170, les histogrammes des temps d'attentes pour chacune des exécutions que nous avons tracé afin de vérifier cette propriété. Ces histogrammes sont présentés dans les figures 6.2 à 6.4 pages suivantes.

L'exécution pour une taille de bloc de 64 lignes n'effectue pas de recouvrement et ne fait donc qu'une seule phase de communication. Cependant l'histogramme des temps d'attente de cette exécution montre une distribution des temps d'attentes très peu homogène. Cette anomalie provient du réseau de communication de la machine utilisée, qui gère mal les messages de grande taille échangés entre les tâches, chacune d'elle expédiant simultanément 384 Ko de donnés vers les autres tâches.

L'histogramme correspondant à une taille de bloc de 32 lignes semble indiquer un problème de synchronisme entre les tâches de l'application. C'est effectivement ce qui s'est passé durant cette exécution : les temps de communications entre tâches ne sont pas vraiment homogènes, sans doute à cause des piètres qualités du réseau de communication de la machine lorsque la taille des données envoyées dans un message n'est pas assez petite.

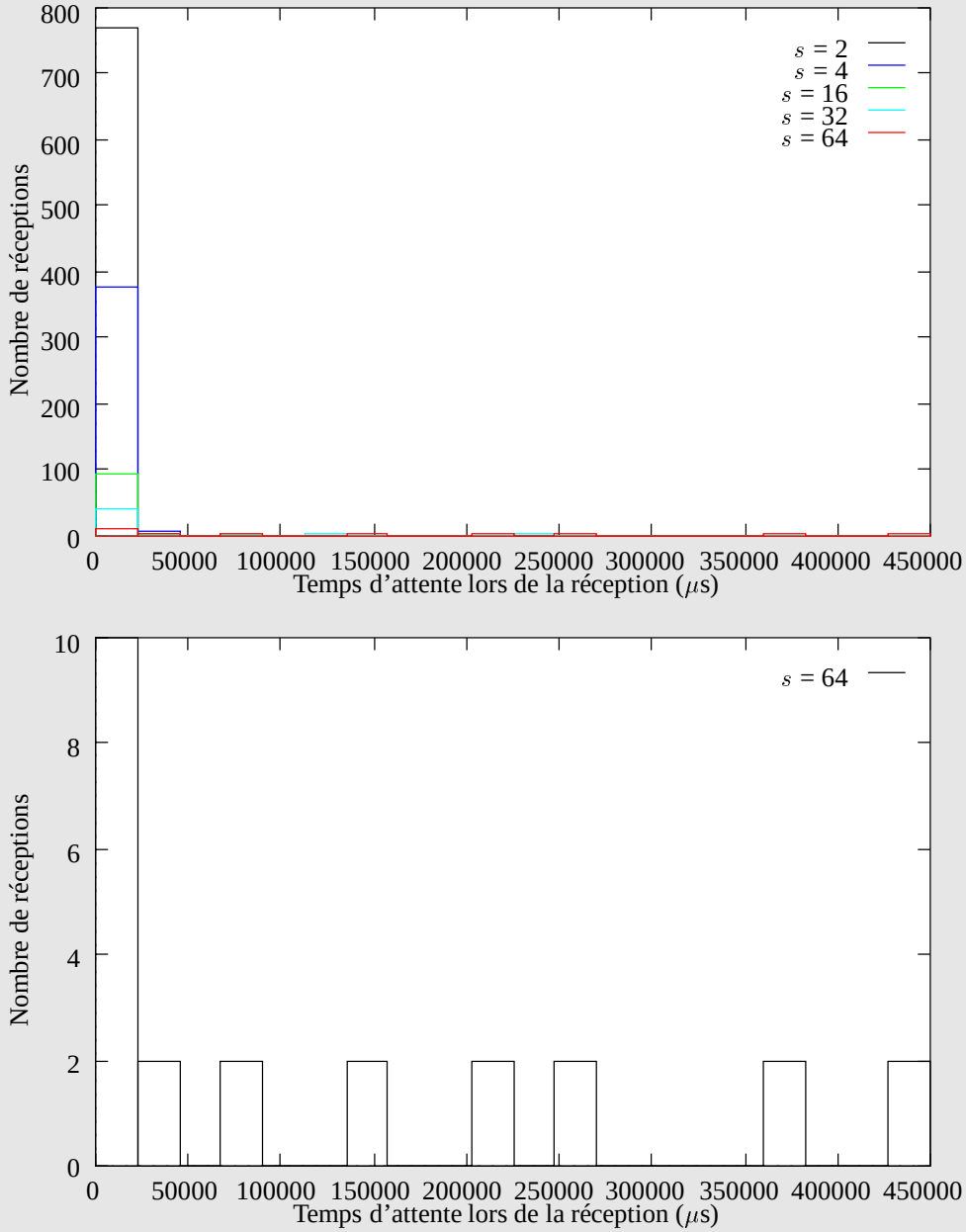


FIG. 6.2. – *Temps d'attente d'une FFT 2D avec recouvrement pour une matrice de 256 éléments, l'algorithme étant exécuté sur 4 processeurs en faisant varier la taille des blocs (paramètre b). L'histogramme du haut récapitule les exécutions effectuées et celui du bas correspond à une exécution sans recouvrement.*

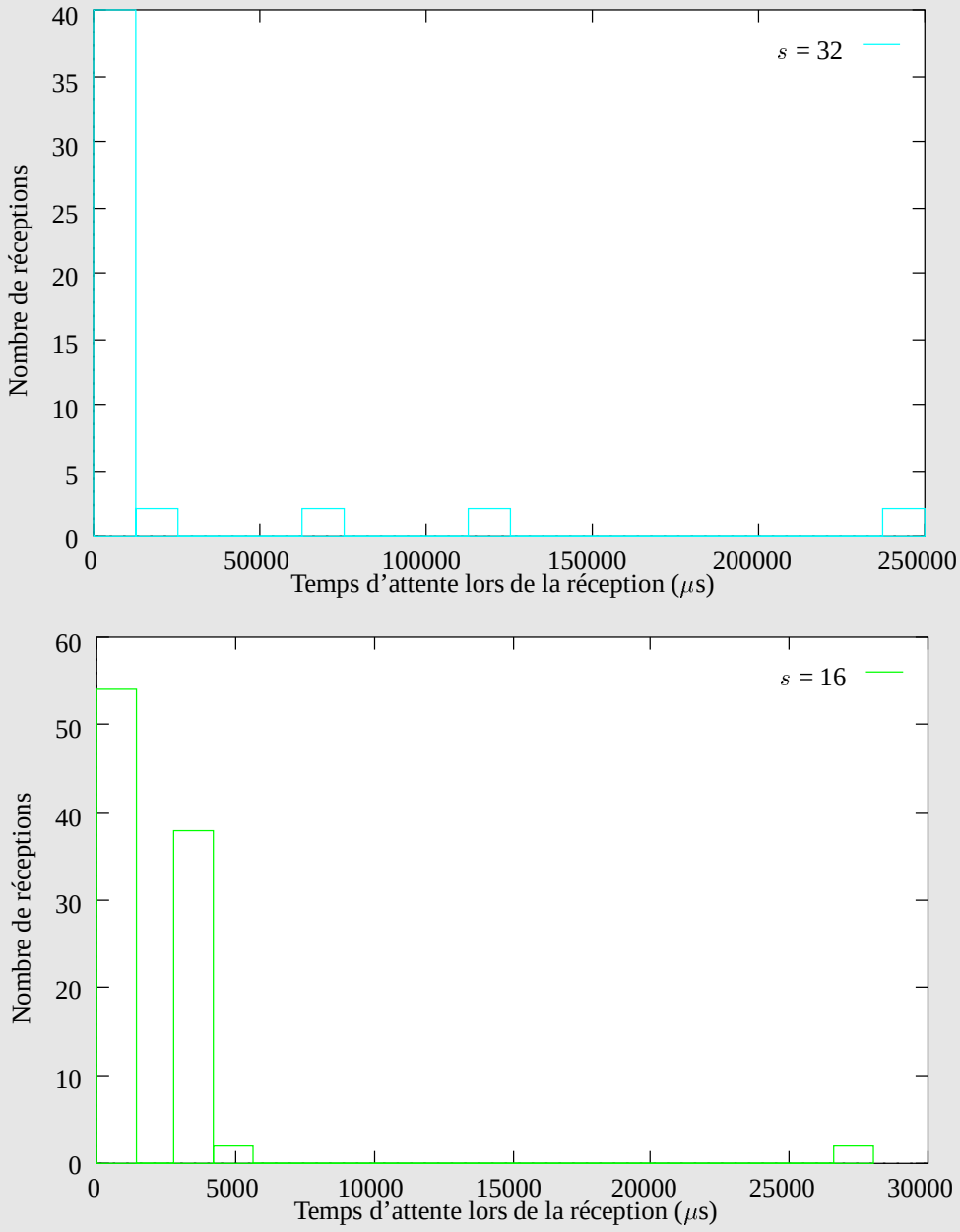


FIG. 6.3. – *Temps d'attente d'une FFT 2D avec recouvrement pour une matrice de 256 éléments, l'algorithme étant exécuté sur 4 processeurs. Les histogrammes présentés ici donnent les performances des exécutions pour des tailles de bloc de 32 lignes et de 16 lignes.*

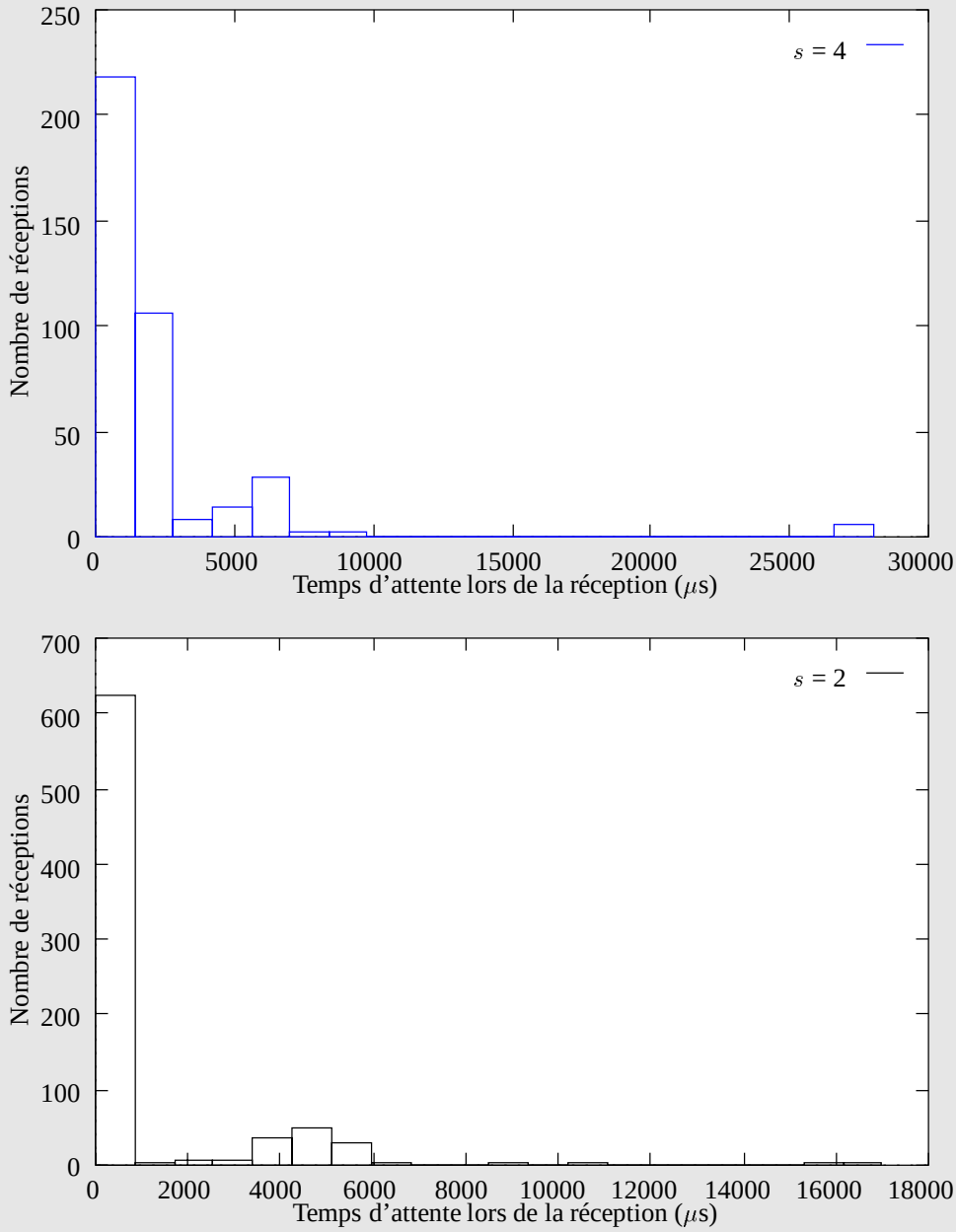


FIG. 6.4. – *Temps d'attente d'une FFT 2D avec recouvrement pour une matrice de 256 éléments, l'algorithme étant exécuté sur 4 processeurs. Les histogrammes présentés ici donnent les performances des exécutions pour des tailles de bloc de 4 lignes et de 2 lignes.*

La répartition des temps d'attente est très bonne pour une taille de bloc de 16 lignes et les réceptions de données n'accusent presque jamais de retard au cours de l'exécution. En effet 98 % des réceptions ont un temps d'attente inférieur à 5 000 μs. Au regard du critère d'homogénéité des temps d'attente cette exécution est la meilleure et cette observation tend à montrer que cette taille de bloc est très bien adaptée aux caractéristiques de rapidité de calcul et de communication de la machine utilisée.

On observe enfin sur les deux derniers histogrammes que certains temps de réception sont beaucoup plus grands que la normale. Cela est dû à une accumulation de retards dans l'exécution des différentes tâches, ces retards étant causés par un trop petit choix de taille de bloc. Le temps d'initialisation de l'envoi d'un message est alors trop important par rapport au temps de propagation du message dans le réseau et le recouvrement des communications par le calcul n'est plus possible.

6.4 Détection automatique d'un problème

Nous avons vu que le surcoût lié aux échanges de données est un bon indicateur des problèmes de performance d'une application. Cela est d'autant plus vrai dans le cas de l'application que nous avons choisi que celle-ci cherche à recouvrir les communications par le calcul. Si la taille de bloc choisie est telle que le temps de traitement d'un bloc est supérieur au temps de la communication de ce bloc le surcoût lié aux communications tend vers zéro. À l'inverse une valeur trop importante de ce surcoût indique que la taille de bloc a été mal choisie.

Le but de notre deuxième expérience est de voir si l'utilisation d'un composant de détection automatique de problème peut faciliter la recherche expérimentale de la meilleure taille de bloc en évitant de traiter l'ensemble des traces produites.

Nous avons commencé par évaluer le surcoût des communications pour une exécution arbitraire avec le composant présenté au § 5.1.3 page 173. Nous avons choisi l'exécution avec une taille de bloc de 16 lignes et avons retenu le plus grand surcoût après une phase c'est-à-dire un peu moins de 22 % du temps d'exécution total (figure 6.5 page ci-contre).

Une fois cette mesure obtenue, nous avons configuré le composant de surveillance de données pour qu'il vérifie que les données qui lui sont fournies ne dé-

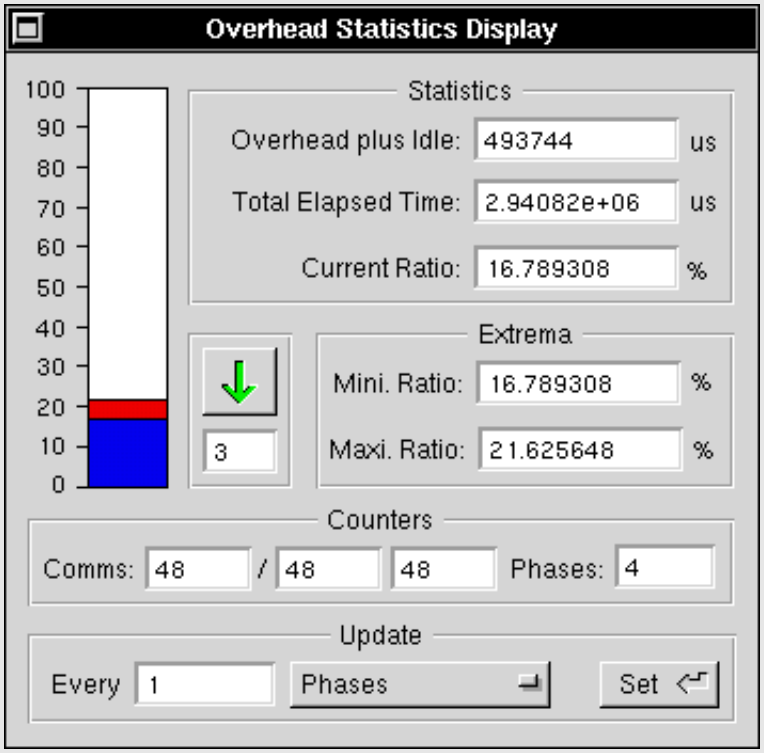


FIG. 6.5. – *Obtention du surcoût lié aux échanges de données pour une exécution de l'application de calcul de transformée de Fourier avec recouvrement dans le cas où la taille des blocs est fixée à 16 lignes de la matrice.*

passent pas la valeur notée précédemment. Nous avons choisi de demander d'une part que le déroulement de la trace soit suspendu en cas de réception d'une donnée invalide (« pause » du traitement) et d'autre part que le composant affiche silencieusement un panneau d'alerte signalant le problème (figure 6.6 page suivante).

Le composant de surveillance est connecté à la sortie du composant de calcul de surcoût qui produit la valeur du surcoût après chaque échantillonnage de cet indicateur. Étant donnée la nature de l'application nous avons choisi un échantillonnage par phases. Le schéma d'analyse ainsi réalisé permet donc d'arrêter le traitement de la trace dès que le surcoût des communications correspondant à une phase est supérieur au surcoût calculé pour l'exécution avec une taille de bloc de 16 lignes.

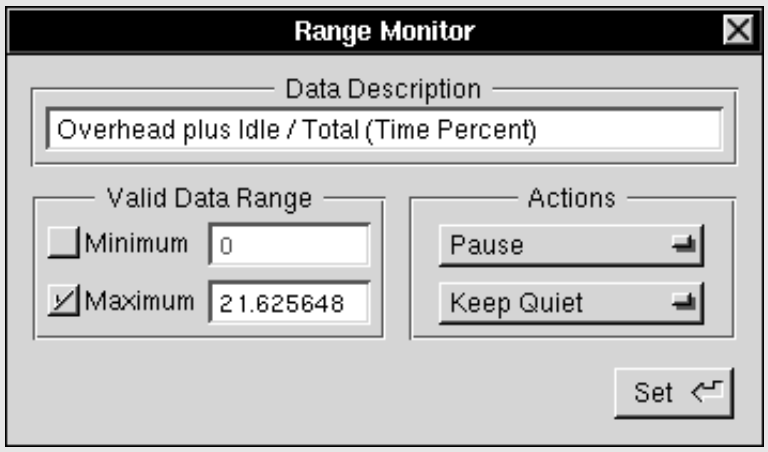


FIG. 6.6. – *Configuration du composant de surveillance de données pour qu'il signale un trop grand surcoût.*

Cette mise en place terminée nous avons chargé tour à tour les traces des autres exécutions. Une fois une trace ouverte par l'environnement nous avons lancé son déroulement et regardé si le composant de surveillance de données indiquait un problème ou non. Les figures 6.7 page ci-contre et 6.8 page 196 illustrent par exemple ce qui s'est passé à la fin de la première phase de l'exécution de l'application pour une taille de bloc de 32 lignes.

Il s'est avéré en fait que l'exécution de référence que nous avions choisi est celle qui a le plus faible surcoût de communication. Pour chacune des autres traces le composant de surveillance de données a suspendu le déroulement de la trace dès la fin de la première phase parce que le surcoût de communication était trop important. Les tailles de blocs correspondantes étaient donc mal adpatées à la taille du problème ou au réseau de communication de la machine parallèle : trop grandes elles provoquaient une augmentation du surcoût de communication à cause des attentes de gros volumes de données, trop petites elles avaient le même effet à cause de l'impossibilité de recouvrir correctement les communications avec des temps de calculs top faibles.

L'intérêt de l'utilisation du composant de surveillance de données a été double. En premier lieu, et comme son nom l'indique, il nous a dispensé de la surveillance de l'indicateur qui nous intéressait, ce qui n'est pas non négligeable. En second

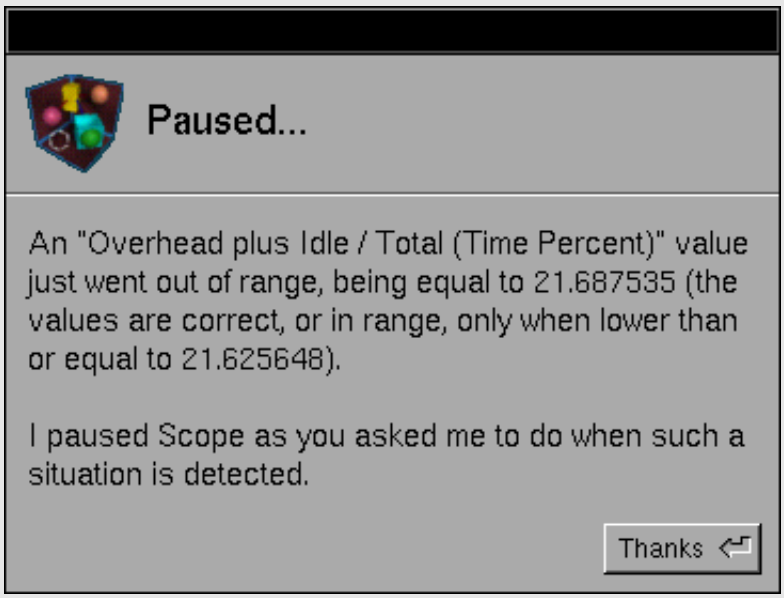


FIG. 6.7. – *Panneau d'alerte affiché par le composant de surveillance de données après qu'il ait reçu une donnée invalide et effectué l'action correspondant à la détection de ce problème.*

lieu il a permis de limiter le déroulement de la trace au strict minimum nécessaire à la détection du problème qui nous intéressait, réduisant ainsi le temps nécessaire à l'observation de l'ensemble des exécutions à comparer avec l'exécution initialement choisie.

6.5 Conclusion

Nous avons démontré l'exploitation des composants introduits dans le chapitre précédent par leur utilisation pour l'évaluation de performance d'une application réelle. Cette étude nous a permis d'illustrer la conception d'un schéma d'analyse adapté à un besoin particulier ainsi que les interactions entre les composants d'un tel schéma. Elle a nous a également permis de confirmer l'intérêt de la possibilité de contrôler l'environnement et le déroulement d'une session d'évaluation de performance depuis un programme visuel, sur des critères choisis par l'utilisateur.

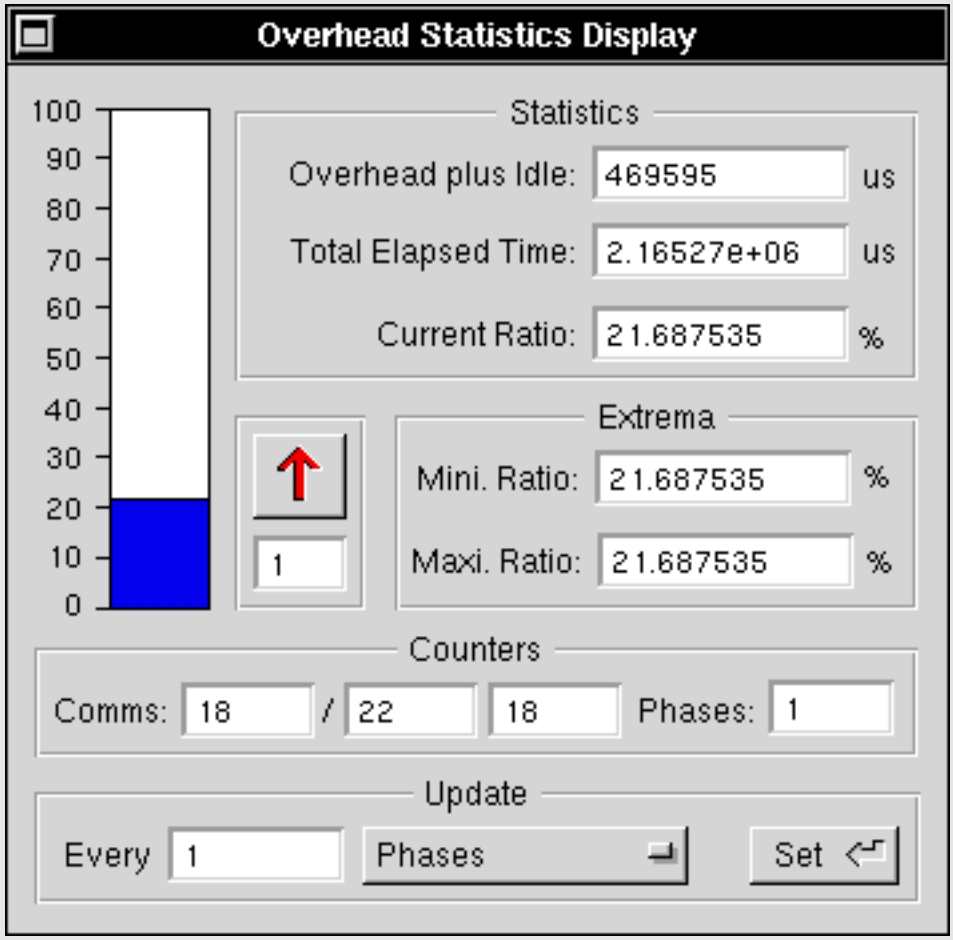


FIG. 6.8. – *Situation ayant provoqué le déclenchement du composant de surveillance de données parce que le surcoût affiché ici est supérieur à la limite fixée par l'exécution de référence.*

A

Fonctionnement des simulateurs

CETTE annexe présente le fonctionnement des simulateurs que nous avons réalisés, le premier étant dédié à la simulation des événements produits par TAPE pour les machines parallèles à base de Transputer et le second permettant la simulation des événements produits par TAPE/PVM lors de l'exécution d'applications PVM.

Les descriptions sont groupées par classes d'événements et donnent les traitements effectué pour chaque type d'événement reconnu par le simulateur sous la forme de l'influence de l'événement sur l'état du système simulé.

Pour chaque objet l'attribut Clock de chaque objet est changé lors de la mise à jour de ce dernier, de même que les attributs indiquant le pourcentage de temps passé par un objet dans un état donné sont mis à jour à chacun de ses changements d'état.

A.1 Support des applications sur Transputer

Les tableaux A.1 et A.2 page suivante rappellent quels sont les états possibles des différents objets manipulés par le simulateur d'OUF/TRITA et donnent le code de couleur qui sera utilisé dans les figures qui suivent.

Nom		Signification
Unused		Inutilisé
Dead		Mort
Idle		Inactif
Talking		Communicant
Busy		Actif

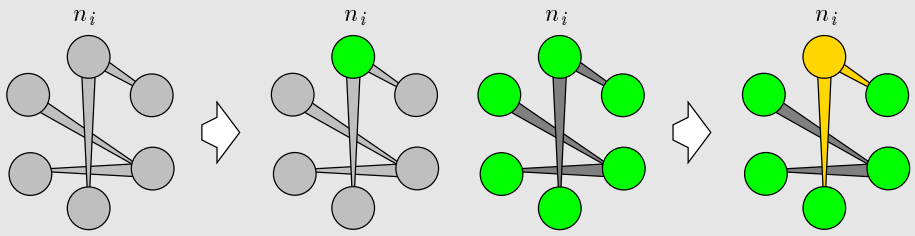
TAB. A.1. – *États donnés aux tâches par le simulateur d'*OUF/TRITA.

Nom		Signification
Unused		Inutilisé
Dead		Mort
Idle		Inactif
Talking		Communicant

TAB. A.2. – *États donnés aux liens par le simulateur d'*OUF/TRITA.

Naissance et mort d'une tâche

La naissance d'une tâche fait passer celle-ci de l'état Unused à l'état Busy. Lorsqu'une tâche meurt, elle passe de l'état Busy à l'état Dead, et tous les liens de communication issus de cette tâche passent également dans l'état Dead puisqu'ils ne pourront plus être utilisés. (Figure A.1.)

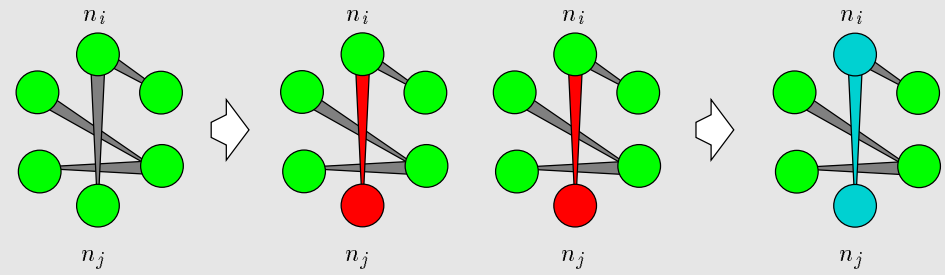


(a) Naissance d'une tâche. (b) Mort d'une tâche.

FIG. A.1. – *Simulation de la naissance et de la mort d'une tâche* OUF/TRITA.

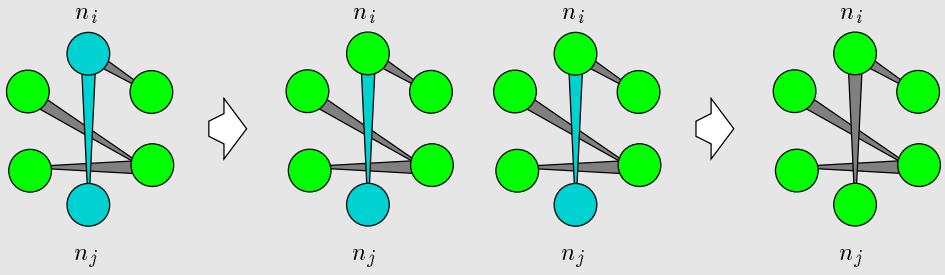
Communication point à point

Un événement de communication concerne trois objets : le nœud demandant à effectuer la communication, le nœud avec lequel il communique et le lien de communication utilisé pour transférer les données (figure A.2).



(a) Début de réception ($n_j \leftarrow n_i$).

(b) Début d'émission ($n_i \rightarrow n_j$).



(c) Fin d'émission ($n_i \not\rightarrow n_j$).

(d) Fin de réception ($n_j \not\leftarrow n_i$).

FIG. A.2. – *Simulation d'une communication entre deux tâches* OUF/TRITA.

Dans le cas d'un début de communication, le nœud demandant à effectuer la communication est forcément dans l'état Busy. L'autre nœud, en revanche, peut être soit dans l'état Idle s'il est déjà arrivé au rendez-vous soit dans un état quelconque dans le cas contraire. Si le nœud avec lequel se fait la communication est dans l'état Idle, alors le rendez-vous s'effectue et les deux nœuds ainsi que le lien de communication passent dans l'état Talking : le trafic sur le lien (attribut Traffic) est alors mis à jour. Dans le cas contraire, le nœud demandant à communiquer et le lien passent dans l'état Idle, indiquant qu'ils sont bloqués en attendant que le

rendez-vous s'effectue. Dans tous les cas, l'attribut Protagonist du nœud est mis à jour, et la valeur de l'attribut Usage du lien est incrémentée.

Lors de la fin d'une communication, le nœud passe dans l'état Busy, son attribut Protagonist devient nul et la valeur de l'attribut Usage du lien est décrémentée. Si cet attribut est alors nul, la communication a pris fin pour les deux parties : l'attribut TotalTraffic est augmenté de la valeur de l'attribut Traffic, puis ce dernier est mis à zéro et le lien de communication passe dans l'état Unused.

A.2 Support des applications PVM

Les tableaux A.1 et A.2 page 198 rappellent quels sont les états possibles des différents objets manipulés par le simulateur de PVM et donnent le code de couleur qui sera utilisé dans les figures qui suivent.

Naissance et mort d'une tâche

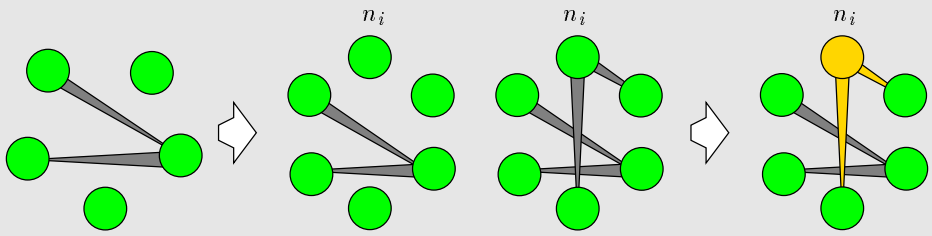
La naissance d'une tâche provoque la création d'un nouveau sommet dans le graphe d'état pour la représenter, et l'initialisation de son état à Busy. Lorsqu'une tâche meurt, elle passe de l'état Busy à l'état Dead, et tous les liens de communication issus de cette tâche passent également dans l'état Dead puisqu'ils ne

Nom		Signification
Busy		Actif
Overhead		Système
Idle		Inactif
Talking		Communicant
Dead		Mort

TAB. A.3. – *États donnés aux tâches par le simulateur de PVM.*

Nom		Signification
Unused		Inutilisé
Overhead		Système
Idle		Inactif
Talking		Communicant
Dead		Mort

TAB. A.4. – *États donnés aux liens par le simulateur de PVM.*



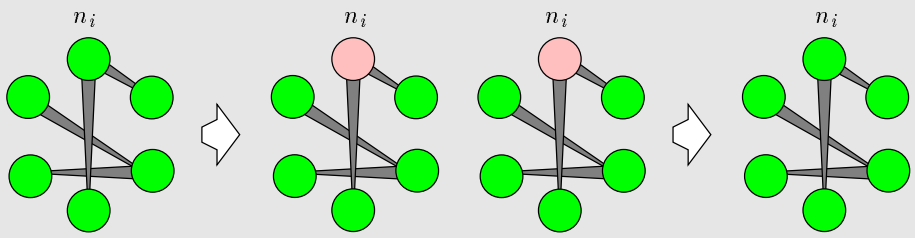
(a) Naissance d'une tâche. (b) Mort d'une tâche.

FIG. A.3. – *Simulation de la naissance et de la mort d'une tâche PVM.*

pourront plus être utilisés. (Figure A.3.)

Empaquetage et désempaquetage

Le début de l'empaquetage ou du désempaquetage de données se traduit par le passage dans l'état Overhead et la fin de l'événement provoque un retour à l'état initial, soit Busy (figure A.4 page suivante).



(a) Début d'empaquetage. (b) Fin d'empaquetage.

FIG. A.4. – *Simulation de la transformation de données de PVM.*

Entrée et sortie de groupe

Ces événements ne modifient donc pas l'état d'une tâche, mais seulement son attribut Groups qui donne la liste des groupes auxquels la tâche appartient.

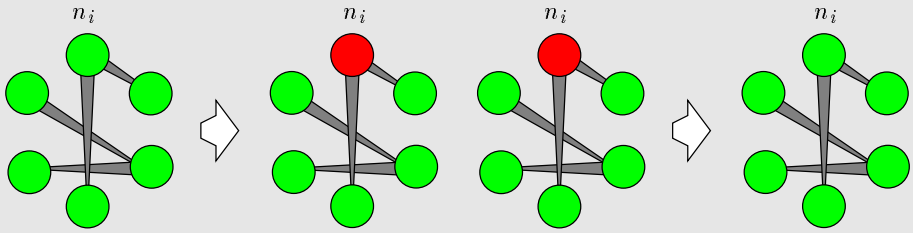
Opération dans un groupe

L'événement Barrier correspondant à la seule opération de groupe disponible en PVM donne la date à laquelle la tâche cherche à passer la barrière ainsi que la durée du passage. Tant que la tâche n'a pas passé la barrière, elle est bloquée.

La tâche passe donc dans l'état Idle lorsqu'il cherche à passer la barrière et revient dans l'état Busy une fois celle-ci passée (figure A.5 page ci-contre).

Communication point à point

Les envois de messages dans PVM sont non-bloquants ; la durée d'un événement d'émission correspond simplement au temps nécessaire au départ du message sur le lien de communication mais en aucun cas au temps effectif de la communication. Le début de l'émission d'un message se traduit par le passage de la tâche dans l'état Overhead. Nous avons choisi de ne pas changer l'état du lien de communication à ce moment-là puisque les données ne sont pas encore parties.



(a) Arrivée à la barrière. (b) Passage de la barrière.

FIG. A.5. – *Simulation du passage d'une barrière de PVM.*

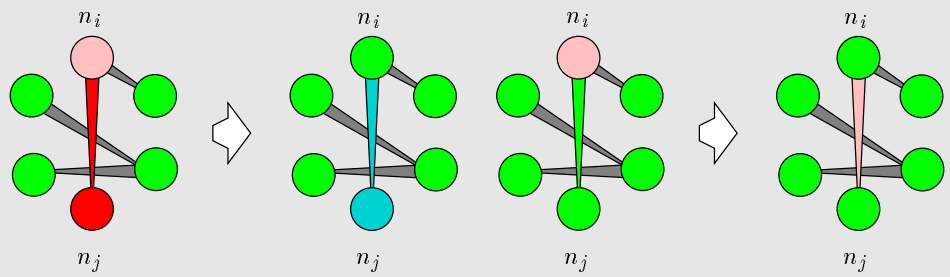
Lors de la fin de l'émission nous sommes sûrs que le message est sur le lien de communication. Son attribut Origin prend pour valeur le surnom de tâche émettrice, son attribut Usage est mis à jour et son attribut Traffic est augmenté de la taille du message. La tâche émettrice passe dans l'état Busy puisqu'elle a terminé son émission.

On peut alors distinguer deux cas suivant qu'une tâche attendait le message ou non (figure A.6 page suivante). Si le message n'était pas attendu, le lien de communication passe dans l'état Overhead pour indiquer qu'il transporte des données non attendues, et on ajoute à la tâche destinataire une information disant que le message lui a été adressé. Dans le cas contraire le lien de communication passe dans l'état Talking, de même que la tâche réceptrice qui est en train d'obtenir le message demandé. De plus l'attribut Traffic du lien prend la valeur de la taille du message[1].

Il existe plusieurs types de réception en PVM et elles nécessitent des traitements adaptés suivant qu'elles sont bloquantes ou non.

Normalement une réception bloquante devrait donner lieu aux opérations suivantes : un temps système dû à l'appel de la fonction PVM, suivi d'un temps de blocage durant l'attente du départ du message, lui-même suivi d'un temps de com-

1. L'attribut Traffic d'un lien de communication s'interprète différemment suivant que l'état de ce dernier est Talking ou non : dans le premier cas il indique le volume de données correspondant à la communication en cours alors que sinon il donne le volume global de données en cours d'acheminement sur le lien.



(a) Récepteur prêt ($n_i \rightarrow n_j$) (b) Récepteur non prêt ($n_i \rightarrow n_j$)

FIG. A.6. – *Simulation de la fin d'une émission de PVM.*

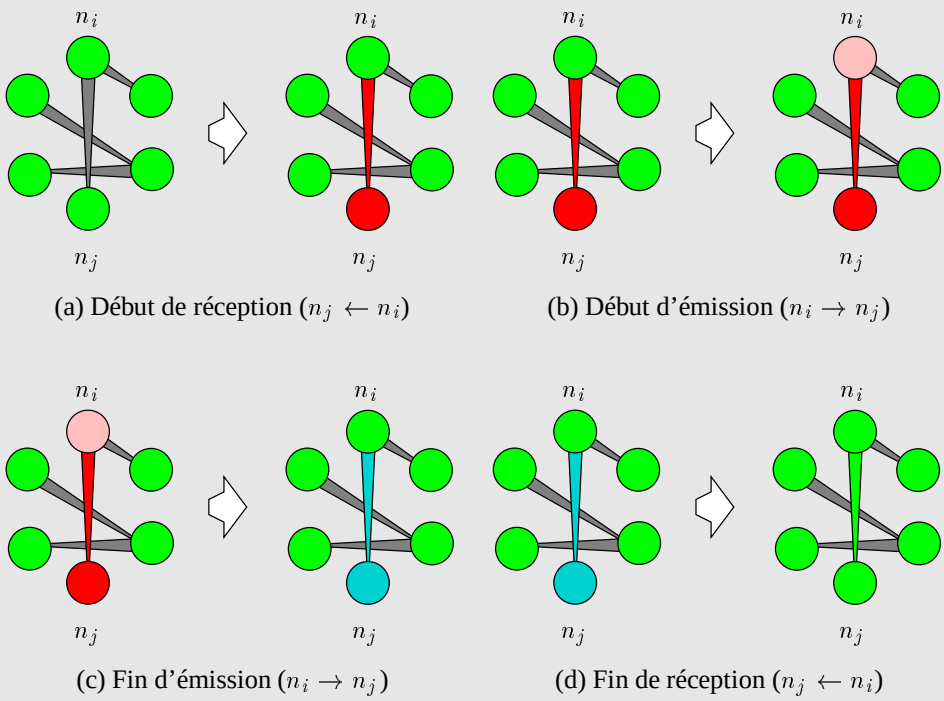
munication correspondant au temps effectif de transfert du message. Cependant le traceur TAPE/PVM effectue une prise de trace au niveau de l'application et non de PVM lui-même : cela signifie qu'il ne peut donner d'informations sur ce qui se passe entre le début de l'appel d'une fonction PVM et sa fin. Même s'il est possible de savoir si un message était arrivé ou non lors de l'appel de la fonction de réception, rien n'indique combien de temps le message est attendu dans le dernier cas. Il est donc impossible de simuler précisément une réception bloquante.

La réception bloquante est traitée comme suit : si le message que l'on cherche à recevoir n'a pas été envoyé alors la tâche passe dans l'état Idle, de même que le lien de communication concerné (figure A.7 page ci-contre); sinon la tâche passe dans l'état Overhead pour indiquer la recopie du message, le lien de communication étant lui-même déjà dans cet état (figure A.8 page 206).

Les réceptions avec délai étant en fait des communications bloquantes avec une limite de temps après laquelle la tâche retourne dans l'état Idle, elles se traitent exactement de la même façon.

Les communications non-bloquantes sont également traitées de façon similaire sauf qu'en cas d'absence du message à recevoir la tâche passe dans l'état Overhead et non dans l'état Idle.

Les attributs Usage et Traffic sont mis à jour de façon évidente lors du traitement des événements de réception.



(a) Début de réception ($n_j \leftarrow n_i$)

(b) Début d'émission ($n_i \rightarrow n_j$)

(c) Fin d'émission ($n_i \rightarrow n_j$)

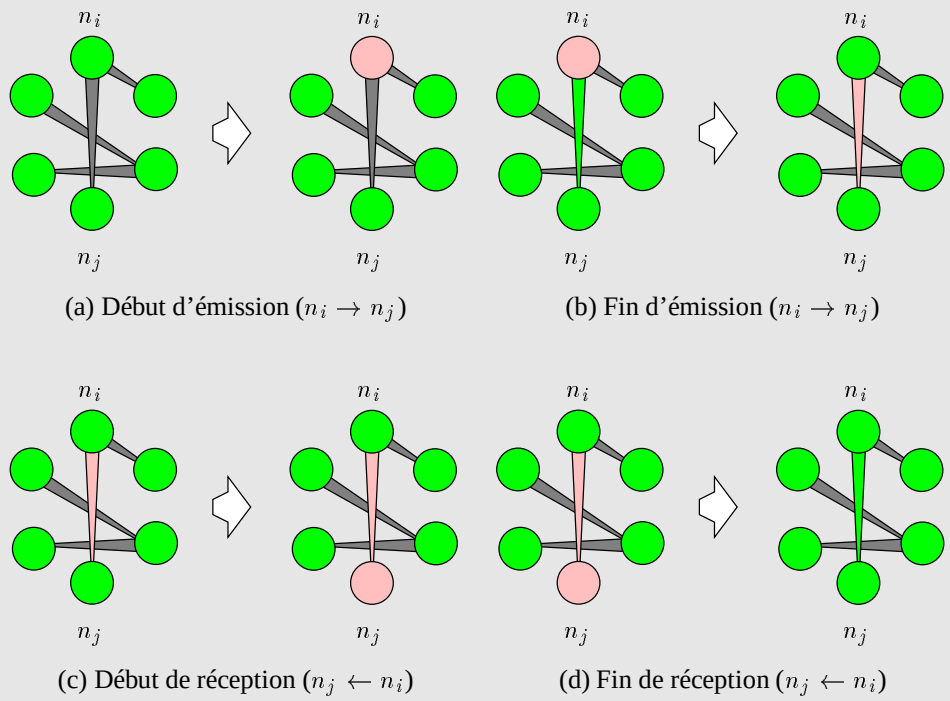
(d) Fin de réception ($n_j \leftarrow n_i$)

FIG. A.7. – *Simulation d'une communication de PVM avec blocage. Le message n'étant pas arrivé lorsque le récepteur cherche à l'obtenir, ce dernier se bloque.*

La fin d'une réception se traduit par un retour à l'état Busy de la tâche concernée et l'augmentation de l'attribut TotalTraffic du lien de la taille du message. L'attribut Traffic du lien est remis à la valeur qu'il possédait avant la communication. Si cette valeur est nulle, le lien passe dans l'état Unused, sinon il repasse dans l'état Overhead puisqu'il y a encore des messages à délivrer.

Communication globale

Une diffusion réussie en PVM est faite de la manière suivante : une tâche diffuse des données à un ensemble de tâches en une seule opération et les tâches réceptrices sont libres de choisir quand et comment recevoir ces données.



(a) Début d'émission ($n_i \rightarrow n_j$)

(b) Fin d'émission ($n_i \rightarrow n_j$)

(c) Début de réception ($n_j \leftarrow n_i$)

(d) Fin de réception ($n_j \leftarrow n_i$)

FIG. A.8. – *Simulation d'une communication de PVM sans blocage. Le message étant déjà arrivé lorsque le récepteur cherche à l'obtenir, celui-ci a juste à le recopier : il ne passe donc pas dans l'état* Talking.

La diffusion peut être vue comme une série de communications ayant lieu simultanément et est traitée ainsi par le simulateur. Le lecteur se rapportera donc à la description de la simulation des communications point à point au § A.2 page 202 pour les détails des opérations effectuées.

Communication dans un groupe

Les fonctions de communication dans un groupe comprennent la diffusion personnalisée, la diffusion partielle, la collecte et la réduction.

En ce qui concerne la simulation les diffusions dans un groupe sont identiques

à la diffusion à ceci près que la liste des tâches réceptrices est donnée par la liste des membres du groupe dans lequel s'effectue la communication. Les opérations effectuées par le simulateur sont donc exactement les mêmes que celles décrites au § A.2 page 205.

Les autres fonctions de communication dans un groupe doivent être appelées par tous les participants à la communication.

Pour la diffusion personnalisée, chacun des événements produits peut alors être considéré comme une communication point à point, toutes ces communications étant des réceptions sauf pour l'émetteur de la diffusion qui effectue autant d'émissions simultanées qu'il y a de participants, lui excepté.

À l'inverse, les événements produits par le regroupement peuvent être considérés comme étant des émissions sauf pour le destinataire des données qui effectue autant de réceptions simultanées qu'il y a de participants, lui excepté.

En ce qui concerne la réduction, et à défaut de connaître l'algorithme qui est employé par PVM, nous avons choisi de faire un traitement similaire à celui du regroupement.

B

Coût de développement

DÉVELOPPER un environnement de visualisation pour l'évaluation de performance est une tâche importante qui nécessite beaucoup de temps, que ce soit pour sa conception que pour sa réalisation. Cette annexe présente quelques informations sur ce qu'a coûté la réalisation de l'environnement Scope.

Le tableau B.1 page suivante présente le classique compte des lignes de code composant l'environnement Scope : un peu moins de 40 000. C'est peu et c'est — à notre avis — très bien : les différents composants de l'environnement sont tous de taille raisonnable et peuvent être gérés facilement. Leur maintenance et leur évolution peuvent facilement être contrôlées par une seule personne.

La taille maniable de l'environnement est le fruit d'un long travail de réflexion sur son architecture, celle-ci étant bien adaptée à la construction incrémentale de petits modules venant s'insérer dans l'environnement. Elle provient également des nombreuses réécritures que l'environnement a subi au cours de son développement, afin de l'optimiser pour le rendre non seulement plus efficace mais également plus modulaire, et aussi pour partager du code entre différentes parties de l'environnement dès que possible.

Ce nombre de lignes de codes restant à la fin de notre travail représente très mal l'effort que constitue la conception et le développement d'un environnement de visualisation pour l'évaluation de performance. Il ne tient aucun compte du temps de réflexion sur les buts de l'environnement et son architecture, et ne saurait

Eve				7 859
	Base			3 901
	Interface			3 521
	Divers			437
Scope				25 646
	Base			1 569
	Associations			7 802
		Formes		3 851
		Autres		3 951
	Composants			14 450
		Base		899
		Contrôle		1 334
		Lect. traces		6 439
			TAPE	2 752
			TAPE/PVM	3 687
		Simulation		2 031
			OUF/TRITA	696
			PVM	1 335
		Vis. textuelle		146
		Graphe		3 601
			Base	1 603
			Algo. dessin	1 998
	Support 3d			1 466
	Inspecteurs			359
Support				6 162
				39 667

TAB. B.1. – *Volume approximatif de code de l'environnement, en lignes (une fois les commentaires enlevés). La rubrique « Support » correspond au dévelopement de bibliothèques de classes générales utilisées par Eve et Scope.*

refléter les — nombreux — développements qui ont simplement servi à tester certaines hypothèses ou qui ont dû être abandonnés faute de succès.

Le développement de Scope a nécessité beaucoup de temps de réflexion et de validation de ces composants à tous les stades du développement. Mais c'est grâce

à cela qu íl est maintenant possible de se consacrer entièrement à l'intégration de nouveaux composants dans Scope et à son utilisation.

Bibliographie

ADOBE SYSTEMS INCORPORATED, *Programming the Display PostScript System with NeXTstep*. Addison-Wesley, Reading, Massachusetts, 1992.

ARROUYE (Yves), *The* TAPE *Reference Manual*, LMC–IMAG, 46, av. Félix Viallet, 38031 Grenoble CEDEX 9, France, 1992.

ARROUYE (Yves), OUF/TRITA *Reference Manual (for release 2.1)*, LMC–IMAG, 46, av. Félix Viallet, 38031 Grenoble CEDEX 9, France, 1993.

ARROUYE (Yves), *The EVE Extensible Visual Environment*, LGI-IMAG, 46, av. Félix Viallet, 38031 Grenoble CEDEX 9, France, 1995.

ARROUYE (Yves), BOUVRY (Pascal), KITAJIMA (J P.), PRÉVOST (Joëlle), ROCH (Jean-Louis), TRON (Cécile) et VILLARD (Gilles), « Manuel du Meganode », rapport technique 98, LMC–IMAG, 46, av. Félix Viallet, 38031 Grenoble CEDEX 9, France, 1993.

BROWN (Marc H.) et HERSHBERGER (John), « Color and Sound in Algorithm Animation », *IEEE Computer*, 1992, t. XXV, n° 12, p. 52–63.

CALVIN (Christophe), *Minimisation du sur-coût des communications dans la parallélisation des algorithmes numériques*, thèse de troisième cycle, Institut National Polytechnique de Grenoble, 46, av. Félix Viallet, 38031 Grenoble CEDEX 9, France, 1995.

CHABERT (Annie), *La programmation visuelle*, mémoire de dea d'informatique, Laboratoire de Génie Informatique, Institut National Polytechnique de Grenoble, Domaine Universitaire, B.P. 53, 38041 Grenoble CEDEX, France, 1991.

CHANG (Shi-Kuo), *Principles of Visual Programming Systems*. Prentice-Hall International, Englewood Cliffs, NJ; London; Singapour, 1990.

COUCH (Alva L.), « Categories and Context in Scalable Execution Visualization », *Journal of Parallel and Distributed Computing*, 1993, t. XVIII, n° 2, p. 195–204.

DEBBAGE (Mark), HILL (Mark) et NICOLE (Denis), « Virtual Channel Router — Version 2.0 User Guide », rapport technique, Department of Electronics & Computer Science, University of Southampton, 1991.

GEIST (Al), BEGUELIN (Adam), DONGARRA (Jack), JIANG (Weicheng), MANCHEK (Robert) et SUNDERAM (Vaidy S.), « PVM3 User's Guide and Reference Manual », rapport technique ORNL/TM–12187, Oak Ridge National Laboratory, Mathematical Sciences Section, P.O. Box 2008, Bldg. 6012, Oak Ridge, TN 37831–6367, 1994.

GEIST (Al), BEGUELIN (Adam), DONGARRA (Jack), JIANG (Weicheng), MANCHEK (Robert) et SUNDERAM (Vaidy S.), *PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing*, Scientific and Engineering Computation. The MIT Press, Cambridge, Massachusetts, 1994.

GLOVER (John), *Developing Applications With DiagramKit 2.1*, University of Houston, 1994.

HACKSTADT (Steven D.), MALONY (Allen D.) et MOHR (Bernd), « Scalable Performance Visualization for Data-Parallel Programs », dans *Proceedings of SHPCC (Scalable High-Performance Computing Conference) 94*, p. 342–349. IEEE Computer Society Press, Knoxville, Tenessee, 1994.

HACKSTADT (Steven T.) et MALONY (Allen D.), « Next-Generation Parallel Performance Visualisation: A Prototyping Environment for Visualization Development », rapport technique CIS-TR-93-21, Department of Computer and Information Science, University of Oregon, Eugene, OR 97403, 1993.

JÉZÉQUEL (Jean-Marc), « Building a Global Time on Parallel Machines », rapport technique 513, Irisa, 1990.

KOUTSOFIOS (E.) et NORTH (Stephen C.), *Drawing graphs with dot*, AT&T Bell Laboratories, 1992.

LAMPORT (Leslie), « Time, Clocks and the Ordering of Events in a Distributed System », *Communications of the ACM*, 1978, t. XXI, n° 7.

LEROUDIER (Jacques), « La simulation à événements discrets », 1979.

MAILLET (Éric), *Perturbations induites par l'instrumentation de programmes parallèles — Mise au point du traceur* TAPE, mémoire de maîtrise, INPG & UJF, 1994.

MAILLET (Éric), TAPE*/PVM an efficient performance monitor for PVM applications — User guide*, LMC–IMAG, 46, av. Félix Viallet, 38031 Grenoble CEDEX 9, France, 1995.

MAILLET (Éric) et TRON (Cécile), « On Efficiently Implementing Global Time for Performance Evaluation on Multiprocessor Systems », *Journal of Parallel and Distributed Computing*, 1995, t. XXVIII, n° 1, p. 84–93.

MILLER (Barton P.), « What to Draw? When to Draw? An Essay on Parallel Program Visualization », *Journal of Parallel and Distributed Computing*, 1993, t. XVIII, n° 2, p. 265–269.

NEXT COMPUTER, INC., *NeXTstep and the Objective-C Programming Language*. Addison-Wesley, Reading, Massachusetts, 1992.

NEXT COMPUTER, INC., *NEXTSTEP User's Guide*. NeXT Computer, INc., 900 Chesapeake Drive, Redwood City, CA 94063, 1994.

NORTH (Stephen C.) et VO (Kiem-Phong), « Dictionary and Graph Libraries », dans *Proceedings of the 1993 Winter USENIX Techical Conference*, p. 1–11. USENIX Association, 1993.

PANCAKE (Cherry), GANNON (Dennis), UTTER (Sue) et BERGMARK (Donna), « Supercomputing '90 BOF Session on Standardizing Parallel Trace Formats », 1990.

POST (Frits) et HIN (Andrea), *Advances in Scientific Visualization*, Focus on Computer Graphics, Eurographics, Springer Verlag, 1991.

RASURE (John) et ARGIRO (Danielle), *Khoros User's Manual*, University of New Mexico.

REED (Daniel A.), AYDT (Ruth A.), MADHYASTHA (Tara M.), NOE (Roger J.), SHIELDS (Keith A.) et SCHWARTZ (Bradley W.), « An Overview of the Pablo Performance Analysis Environment », rapport technique, Department of Computer Science, University of Illinois, Urbana, Illinois 61801, 1992.

SHU (Nan C.), *Visual Programming*. Van Nostrand Reinhold, New-York, 1988.

TANIMOTO (Steven L.), « VIVA: A Visual Language for Image Processing », *Journal of Visual Languages and Computing*, 1990, t. I, n° 2, p. 127–139.

UPSTILL (Steve), *The RenderMan Companion—A Programmer's Guide to Realistic Computer Graphics*, 2e édition. Addison-Wesley, Reading, Massachusetts, 1992.

WIENER (Richard S.), *Objective-C: Object-Oriented Programming Techniques*. Addison-Wesley, Reading, Massachusetts, 1991.

III

Conclusion

Bilan et perspectives

LE BUT des concepteurs d'applications parallèles est de réaliser des applications les plus performantes possible *i.e* qui s'exécutent très rapidement et avec efficacité. Lorsqu'une application semble ne pas être aussi efficace que prévu il est nécessaire de pouvoir étudier en détail son fonctionnement afin de détecter les causes de sa non-performance.

Le meilleur moyen — et aujourd'hui encore le seul qui soit viable — d'observer effectivement la manière dont l'exécution d'une application parallèle s'est déroulée est d'utiliser un environnement de visualisation. Ces outils sont particulièrement utiles pour l'évaluation de performance d'une application car ils permettent de produire des indices de performances représentatifs de son fonctionnement et de son utilisation des ressources à sa disposition, puis de présenter graphiquement ces indicateurs dans des vues synthétiques, instantanées ou décrivant l'évolution de l'application au cours du temps.

Notre travail a précisément porté sur l'étude, la conception et la réalisation d'un environnement de visualisation pour l'évaluation de performance des applications parallèles, Scope. Nous avons accordé une importance particulière à l'étude des besoins des utilisateurs de l'évaluation de performance que ce soit au niveau des informations que l'environnement doit leur fournir que de la manière dont ces informations doivent être présentées et peuvent être exploitées. Un grand soin a également été apporté dans le choix de l'architecture de l'environnement

afin de garantir sa pérennité, son extensibilité mais aussi sa facilité d'utilisation et son support des besoins et des actions de l'utilisateur, qu'il soit novice ou expert. Nous nous sommes enfin intéressés à la possibilité de réduire le temps nécessaire à l'évaluation de performance d'une application que ce soit en minimisant le coût d'utilisation de l'environnement ou en développant des mécanismes explicites d'aide à l'utilisateur et de support de ses actions.

L'environnement de développement que nous avons choisi, NEXTSTEP, s'est révélé particulièrement agréable et puissant. Nombre des caractéristiques de notre environnement auraient sans aucun doute nécessité un temps de développement bien plus important dans tout autre environnement. Nous n'avons regretté à aucun moment le choix de cet environnement : même si son apprentissage est parfois difficile et s'il a tout de même quelques lacunes, nous avons trouvé qu'il est globalement de très haute qualité et qu'il permet vraiment la réalisation d'applications complexes dans des conditions très satisfaisantes. NEXTSTEP est de plus très agréable pour l'utilisateur, un point qui est de grande importance pour nous, la finalité d'un environnement d'évaluation de performance étant tout de même d'être exploité par les utilisateurs de machines parallèles.

Scope est architecturé autour d'un langage de programmation visuelle orienté objet et extensible, Eve, qui est la base de l'environnement. Tous les objets de l'environnement sont des composants Eve et interagissent par propagation de données dans un programme visuel représentant le graphe d'analyse et de visualisation utilisé pour l'évaluation de performance. L'environnement est entièrement redéfinissable par l'utilisateur par simple remplacement de ses composants ; il est de même extensible par ajout de nouveaux composants dans le programme visuel utilisé pour l'évaluation de performance.

L'environnement est conçu de manière à supporter aisément l'évaluation de performance des applications parallèles futures en étant capable de traiter de manière homogène différents modèles de programmation parallèles, présents ou à venir. Scope interagit de manière particulière avec les composants de lecture de traces afin d'offrir à l'utilisateur une interface générique pour l'exploitation de différents types de traces correspondant à différents modèles de programmation. Cette interaction est faite par l'intermédiaire de protocoles bien définis et faciles à mettre en place pour intégrer un nouveau modèle de programmation dans Scope. Nous avons montré par la réalisation de composants adaptés au traitement de deux modèles qu'une telle intégration n'était pas particulièrement difficile et que l'ajout d'un nouveau modèle de programmation ne représentait pas un effort de dévelop-

pement démesuré.

Hormis quelques composants spécifiques au modèle de programmation correspondant aux traces manipulées l'ensemble des outils d'analyse et de visualisation présents dans Scope peut être conçu de manière générique, sans aucune dépendance par rapport à la syntaxe et à la sémantique des événements décrivant les actions de l'application. Scope encourage ainsi la réalisation d'une boîte à outils d'analyse et de visualisation, l'utilisateur pouvant choisir dans ces outils ceux qui sont les plus appropriés pour l'évaluation de performance de son application *i.e.* à la production et à la représentation des indices de performance qui l'intéresse. L'association de programmes visuels complets à un modèle de programmation donné permet la constitution d'une palette de schémas d'analyses prêts à l'emploi dans laquelle un utilisateur peut puiser librement en fonction de ses besoins.

Nous avons également étudié le problème de l'exploration interactive de l'exécution d'une application parallèle c'est-à-dire la possibilité de se déplacer à tout moment vers un instant quelconque de cette exécution et d'observer l'état de l'application et des indicateurs de performance sélectionnés à ce moment précis. Scope est à notre connaissance le seul environnement comprenant des mécanismes pour le support du parcours non séquentiel d'une exécution.

Au niveau de la visualisation Scope introduit un certain nombre de concepts dont le respect permet de garantir la qualité des représentations graphiques ainsi qu'une grande souplesse d'utilisation des visualisations proposées. Scope prône l'utilisation systématique d'outils de manipulation directe des informations présentées. Il encourage également la standardisation de ces outils et leur utilisation cohérente par différentes visualisations. De tels outils permettent par exemple l'accès sélectif aux informations sous-jacentes à une visualisation donnée et l'obtention de données précisant ces informations. Ils permettent également d'agir sur l'organisation des informations dans une visualisation afin de la faire correspondre aux schémas de représentation auxquels l'utilisateur est habitué.

Dans ce même esprit de support des habitudes ou des besoins de l'utilisateur, toutes les visualisations de Scope supportent également la personnalisation c'est-à-dire la modification de leur apparence par l'utilisateur. Nous avons introduit dans Scope un mécanisme de personnalisation par association qui permet de spécifier de manière générale et standardisée un certain nombre de contraintes d'apparence, par exemple au niveau des formes ou des couleurs des objets représentés dans les visualisations, celles-ci étant clientes des gestionnaires d'associations de

Scope. Cette organisation assure une forte cohérence des visualisations de Scope et facilite grandement la localisation d'informations en relation dans différentes visualisations. Enfin Scope fournit un cadre générique d'exploitation de visualisations tridimensionnelles, ce type de représentation étant amené à être de plus en plus présent dans les environnements de visualisation à cause de la plus grande densité d'information qu'il supporte.

Une des idées directrices de Scope est de faciliter la construction de nouveaux schémas d'analyse et l'essai de nouvelles méthodes d'évaluation de performance. Le fait d'être réalisé autour d'un langage de programmation visuelle contribue grandement à cet esprit de « brancher-tester » en rendant l'environnement extrêmement versatile et en facilitant sa modification et son extension. Nous avons validé cette facilité en démontrant la réalisation de nouveaux composants d'analyse et de visualisation puis en les exploitant pour évaluer les performances respectives de différentes variantes d'une application parallèle.

Le dernier point essentiel de notre environnement est son support de l'utilisateur à tous les niveaux de sa tâche d'évaluation de performance. Les mécanismes de personnalisation globale ou de manipulation directe des visualisations font partie de support puisqu'ils facilitent la présentation des informations suivant les désirs de l'utilisateur et l'accès aux données associées à ces informations. La possibilité de modifier interactivement l'environnement en éditant le programme visuel représentant l'analyse effectuée, de même que la constitution d'une bibliothèque de schémas d'analyse, contribuent également à ce support. Mais la caractéristique la plus originale de Scope dans ce domaine est la possibilité de contrôler le déroulement d'une session d'évaluation de performance depuis le schéma d'analyse. Nous avons ainsi démontré la faisabilité de schémas d'analyse « intelligents » agissant en lieu et place de l'utilisateur. Nous avons illustré cette capacité en réalisant un tel schéma, celui-ci arrêtant automatiquement le déroulement de la trace dès la détection d'un problème de performance spécifié par l'utilisateur.

Pour résumer, la contribution de notre travail au domaine de la visualisation pour l'évaluation de performance a porté sur plusieurs thèmes : la généricité et l'extensibilité de l'environnement et des outils qu'il supporte, l'étude des moyens d'exploration non linéaires d'une exécution, l'offre de mécanismes généraux de personnalisation et la définition de nouveaux standards de qualité des visualisations, l'automatisation de certaines tâches d'évaluation de performance qui étaient jusqu'ici du ressort de l'utilisateur et enfin, de manière générale, la facilité d'utilisation et d'extension d'un environnement de visualisation.

Perspectives

Scope est un prototype d'environnement de visualisation pour l'évaluation de performance d'applications parallèles. Son existence démontre la faisabilité de l'approche que nous avons choisie et la validité des concepts ayant présidé à nos choix de conception. Scope nous a également permis de tester et valider un certain nombre d'idées sur ce que doit offrir un tel environnement.

Il n'en est pas moins évident que cet environnement n'est pas définitif et qu'il doit continuer à évoluer dans plusieurs directions, que ce soit pour compléter ses capacités actuelles ou pour expérimenter de nouvelles idées.

Nous nous proposons tout d'abord de compléter l'offre de Scope en matière de composants d'analyse et de visualisation. En effet celle-ci est aujourd'hui cruellement limitée, notre effort de réalisation ayant porté sur la mise en place d'une architecture à même de pouvoir supporter l'évolution future des modèles de programmation et de l'évaluation de performance des applications. Si nous pensons avoir atteint ce but et garanti la pérennité de l'environnement développé il n'en reste pas moins que son utilisation quotidienne nécessite la présence d'une bibliothèque d'analyse et de visualisation couvrant les besoins les plus courants.

Nous voulons de même compléter notre étude de l'utilisation des visualisations tridimensionnelles pour l'évaluation de performance. Nous avons développé un support pour ce type de représentations dans Scope et avons démontré son utilisation en réalisant une visualisation de graphes tridimensionnelle. Telle qu'elle est utilisée, cette représentation ne donne que des informations instantannées sur l'état de l'application évaluée ; une telle visualisation ne fait pas partie des plus utilisées en évaluation de performance. Nous voulons donc étudier d'une part l'extension de cette visualisation, par exemple en développant une variant permettant de représenter des diagrammes espace-temps en trois dimensions, et d'autre part la construction de visualisations statistiques tridimensionnelles, en débutant par exemple par une variante du diagramme de Kiviat gérant une dimension temporelle que nous avons commencé à développer.

Nous souhaitons ensuite explorer le domaine de l'analyse et de la visualisation multi-modèles, c'est-à-dire la mise en relation des événements générés par une application écrite dans un langage de haut niveau utilisant lui-même une bibliothèque de communication produisant des traces. L'intérêt d'une telle étude est de pouvoir faire le lien entre les différents niveaux d'expression du parallélisme

et de déterminer si un problème détecté à un niveau donné ne provient pas en fait de la manière dont les couches inférieures fonctionnent.

Un axe de recherche également intéressant concerne le couplage de la mise au point pour les performances et pour la correction par l'exploitation d'un mécanismes de réexécution déterministe. La part des problèmes de performance dus effectivement à des problèmes de correction est en effet plus grande qu'on ne le pense et il n'existe actuellement pas d'outils capables de montrer simultanément ces deux aspects lors de l'observation d'une application.

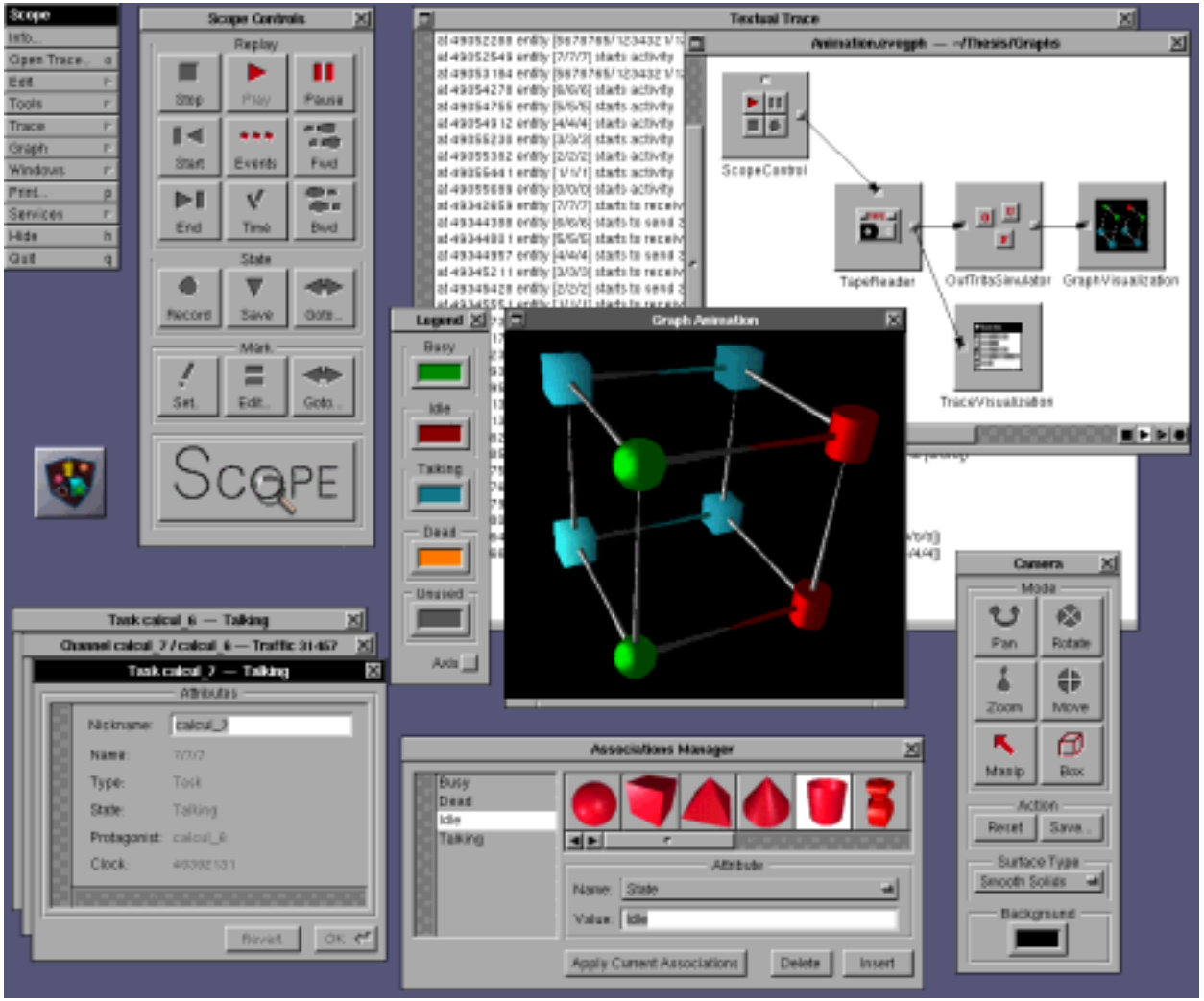
Un autre thème de recherche est le développement de nouvelles techniques d'analyse et présentation des informations. Outre la sonorisation, certainement prometteuse mais très peu exploitée par manque de compréhension de la manière dont il faut l'utiliser, nous souhaitons travailler sur l'exploitation des techniques de regroupement statistiques dans le but de permettre l'identification de tâches représentatives du comportement d'une application afin de réduire les informations à manipuler tout en gardant une vision de qualité de ce comportement.

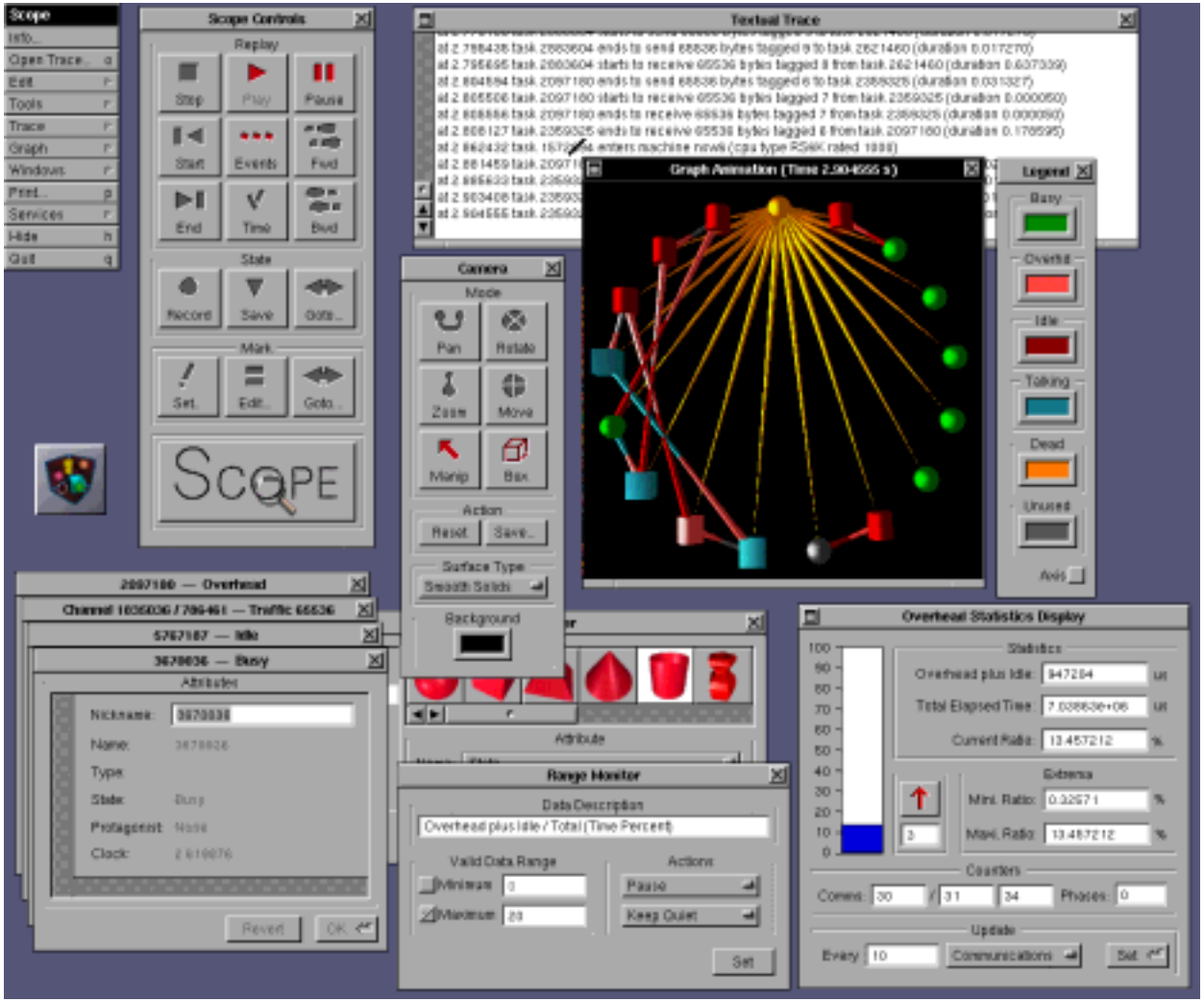
Nous avons abordé dans notre travail la visualisation de traces d'exécution *post-mortem*. Les volumes de données générées par les applications parallèles lors d'une exécution tracée sont aujourd'hui tellement importants qu'il devient nécessaire de trouver des moyens de réduire la taille des traces sous peine de ne plus pouvoir les stocker pour exploitation. Il est donc intéressant d'étudier la possibilité de n'utiliser que des informations de performances « légères » obtenues par exemple par échantillonnage et non plus par analyse de traces décrivant complétement une exécution.

Un problème ouvert lié à ce dernier point est l'exploitation de données de performance en temps réel afin d'éviter le stockage de traces d'exécution chaque jour plus grandes. Ceci pose cependant nombre de problèmes tant au niveau du choix des informations à produire et de leur forme que de la manière d'interagir avec une application en cours d'exécution. Une telle possibilité, même partielle, permettrait également d'explorer de nouvelles techniques d'analyse du comportement d'une application parallèle en instaurant par exemple un dialogue entre cette dernière et l'environnement d'évaluation de performance, celui-ci indiquant à l'application les données dont il a besoin à un instant donné.

Enfin, dans le cadre du support de multiples modèles de programmation, nous pensons qu'il serait intéressant d'étudier les problèmes de l'évaluation de performance d'applications utilisant le parallèlisme de données par l'intermédiaire

de langages de type HPF. La principale difficulté de cette tâche est due à l'impossibilité d'instrumenter le code source des applications pour la production de traces ou de données de performance. En effet les compilateurs associés à ces langages manipulent tellement le code de l'application pour essayer de l'optimiser que les informations produites par une instrumentation classques ne sont pas du tout reliables au code initial de l'application. Il faut donc redéfinir les besoins de l'évaluation de telles applications et mettre au point des méthodes permettant à l'utilisateur de spécifier ce qu'il souhaite observer, ces spécifications étant utilisées par le compilateur pour produire les informations voulues au bon moment et par l'environnement d'évaluation de performance pour relier les indicateurs obtenues au code de l'utilisateur.


Scope
Info...
Open Trace... o
Edit r
Tools r
Trace r
Graph r
Windows r
Print... p
Services r
Hide h
Quit q
Scope Controls
Replay
Stop
Play
Pause
Start
Events
Fwd
End
Time
Bwd
State
Record
Save
Goto...
Mark
Set
Edit...
Goto...
SCOPE
Textual Trace
Animation.vrmgph — ~/Thesis/Graphs
ScopeControl
TapeReader
OutTritsSimulator
GraphVisualization
TraceVisualization
Legend
Busy
Idle
Talking
Dead
Unused
Axis
Graph Animation
Camera
Mode
Pan
Rotate
Zoom
Move
Manip
Box
Action
Reset
Save...
Surface Type
Smooth Solids
Background
Task calcul_6 — Talking
Channel calcul_7 / calcul_6 — Traffic 31457
Task calcul_7 — Talking
Attributes
Nickname: calcul_7
Name: 7/7/7
Type: Task
State: Talking
Protagonist: calcul_6
Clock:
Revert
OK
Associations Manager
Busy
Dead
Idle
Talking
Attribute
Name: State
Value: Idle
Apply Current Associations
Delete
Insert


Scope
Info...
Open Trace... o
Edit r
Tools r
Trace r
Graph r
Windows r
Print... p
Services r
Hide h
Quit q
Scope Controls
Replay
Step
Play
Pause
Start
Events
Fwd
End
Time
Bwd
State
Record
Save
Goto...
Mark
Set
Edit...
Goto...
SCOPE
Textual Trace
Graph Animation (Time 2.90-4555 s)
Legend
Busy
Overhd
Idle
Talking
Dead
Unused
Axes
Camera
Mode
Pan
Rotate
Zoom
Move
Manip
Box
Action
Reset
Save...
Surface Type
Smooth Solids
Background
2097180 — Overhead
Channel 1036036 / 786461 — Traffic 65536
5767167 — Idle
3670036 — Busy
Attributes
Nickname:
Name:
Type:
State:
Protagonist:
Clock:
Revert
OK
Attribute
Range Monitor
Data Description
Overhead plus Idle / Total (Time Percent)
Valid Data Range
Minimum
Maximum
Actions
Pause
Keep Quiet
Set
Overhead Statistics Display
Statistics
Overhead plus Idle:
Total Elapsed Time:
Current Ratio:
Extrema
Mini. Ratio:
Maxi. Ratio:
Counters
Comms:
Phases:
Update
Every:
Communications
Set

IV

Annexes

A

Modèles de programmation

De même que pour la programmation séquentielle, il existe en parallélisme différents modèles de programmation, plus ou moins bien adaptés au type des applications développées et à l'architecture de la machine sur laquelle les applications seront exécutées.

La tendance actuelle est cependant d'essayer de faire abstraction de l'architecture de la machine et de choisir un modèle de programmation uniquement en fonction des contraintes liées à l'application ; la réalisation du modèle choisi sur la machine parallèle se charge de fournir une vitesse d'exécution maximale étant données les caractéristiques et les contraintes physiques de la machine.

À l'heure actuelle, on distingue quatre grands modèles de programmation représentant différentes conceptions de l'écriture d'applications parallèles :

1. la *programmation par échange de messages* (« message passing ») est un modèle simple mais puissant, particulièrement adapté aux applications utilisant principalement le parallélisme de contrôle ;
2. la *communication par variables partagées* (« shared-memory communication ») est très souvent utilisée sur les systèmes à mémoire partagée, dont les systèmes traditionnels disposant d'extensions pour le parallélisme ;
3. la *programmation par parallélisme de données* (« data parallelism ») répond

aux besoins spécifiques de la méthode de programmation du même nom ;

4. le modèle *client-serveur* ou *objet* est approprié pour les applications écrites en termes de services utilisés par des processus clients.

Nous donnons ci-dessous les principes de ces modèles de programmation et décrivent les possibilités des langages ou bibliothèques d'écritures de programmes parallèles les plus connues qui les exploitent.

A.1 Programmation par envoi de messages

Le modèle de programmation le plus populaire sur machines à mémoire distribuée (et *a priori* le mieux adapté à celles-ci) est certainement celui de l'*envoi de messages*. Il consiste à écrire une application comme un ensemble de tâches qui s'exécutent indépendamment et dont toutes les données sont privées : lorsqu'une tâche a besoin d'informations disponibles dans une autre tâche, elle ne peut l'obtenir qu'en effectuant une communication afin d'obtenir ces données.

La programmation par envoi de messages telle que nous la connaissons a été introduite par HOARE (1978) avec CSP (*Communicating Sequential Processes*). Nous présentons ce modèle de programmation et son évolution jusqu'à sa forme actuelle à travers trois outils représentatifs, le premier étant CSP et les deux derniers représentant les standards *de facto* et à venir en matière de programmation par envoi de messages.

A.1.1 CSP (processus séquentiels communicants)

En CSP, les processus séquentiels disposent de deux primitives de communication, l'une pour envoyer et l'autre pour recevoir des données. Un processus peut alterner envois et réception au cours de son éxécution sans contraintes d'ordre ni de nombre (figure A.1 page suivante).

Les envois de données peuvent être faites entre deux processus uniquement, *i.e.* CSP ne permet que des *communications point-à-point*. Qui plus est, ce sont des *communications bloquantes synchrones, i.e.* une communication entraîne une

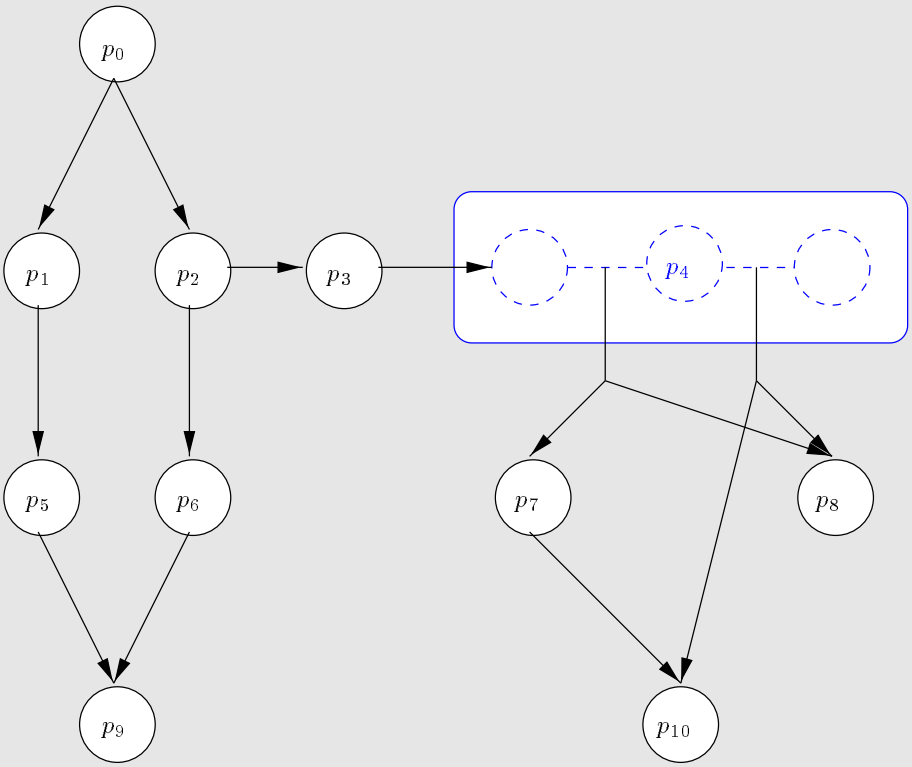


FIG. A.1. – *Modèle de programmation CSP (*Sequential Communicating Processes*). Les processus séquentiels communiquent par envoi de messages. Sur cette figure les cercles représentent des parties de code sans échanges de messages.*

synchronisation par *rendez-vous* entre les processus communicants : le processus qui entame la communication doit attendre que celle-ci soit établie avec un autre processus et le rendez-vous ne se termine que lorsque le message est arrivé à destination, débloquant alors les deux processus (figure A.2 page suivante).

Enfin, CSP offre des mécanismes appelés *gardes* et *alternatives* : une garde est une condition qui doit être vérifiée pour qu'une communication ait lieu, et une alternative permet de spécifier qu'un certain nombre de communications sont attendues mais que leur ordre est *a priori* indéterminé.

Bien que CSP ne soit pas un langage de programmation, il a donné naissance

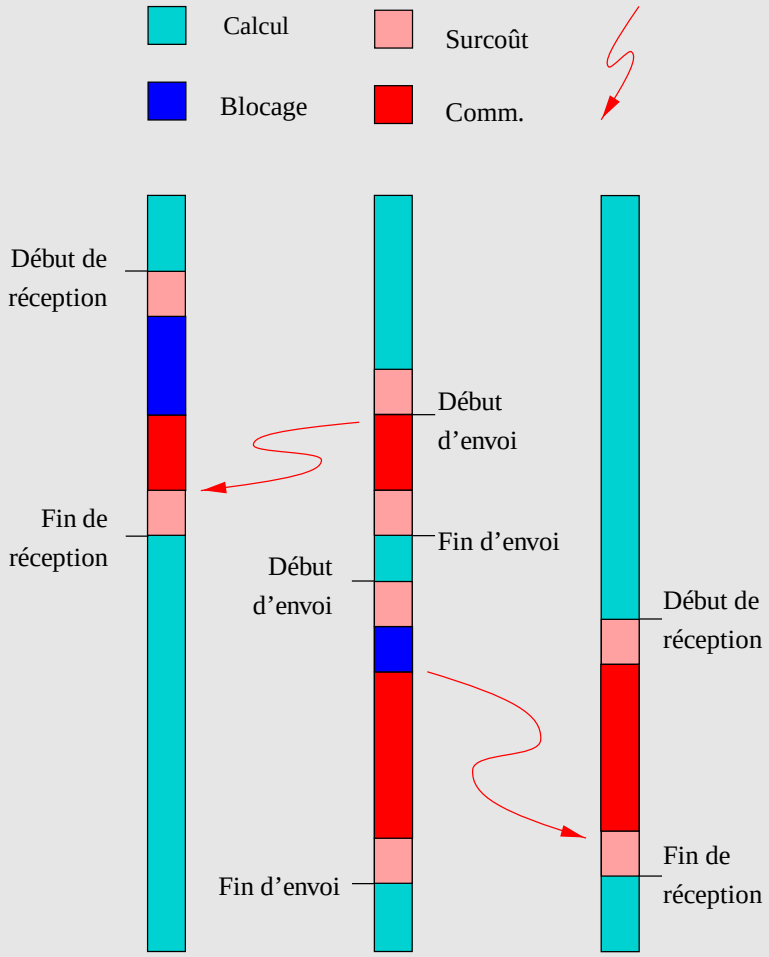


FIG. A.2. – *Communications bloquantes synchrones. Les processus prennent rendez-vous pour communiquer,* i.e. *le premier à commencer la communication est bloqué en attendant que son interlocuteur veuille communiquer avec lui. Les deux processus attendent ensuite la fin de la communication avant de reprendre leurs activités.*

au langage OCCAM, utilisé sur les processeurs Transputer et dont la première version était une traduction presque littérale de CSP. Il est de plus un très bon outil d'introduction à la programmation par envoi de messages et sa simplicité permet de prouver la correction des algorithmes parallèles développés.

A.1.2 PVM (machine parallèle virtuelle)

PVM (*Parallel Virtual Machine*, GEIST *et al.* 1994a, GEIST *et al.* 1994b) est une bibliothèque de programmation parallèle qui permet de considérer un ensemble quelconque de machines hétérogènes comme une machine parallèle (homogène), permettant de développer et d'exécuter des applications parallèles avec un investissement matériel réellement minimal.

En plus des communications bloquantes synchrones présentées à la section précédente, les premières versions de PVM offraient des *communications bloquantes asynchrones* : les processus communiquants prennent toujours rendez-vous, mais seul le récepteur reste bloqué jusqu'à la fin du transfert de données, le rendez-vous prenant fin dès son arrivée (figure A.3 page suivante). Un tel type de communication offre l'avantage de libérer l'émetteur dès que possible, seul le processus ayant besoin des données restant bloqué jusqu'à leur réception effective.

En fait, PVM fournit des *tampons de communication* à l'application parallèle, tampons dans lesquels sont placées (ou les données à émettre ou reçues). Dès lors, pour l'application, une communication est terminée lorsque le tampon dédié à la communication peut être réutilisé : pour le processus émetteur, le fait que le message soit déjà arrivé à destination ou non à cet instant n'est d'aucune importance.

Les dernières versions de PVM se basent toutes sur ce principe pour l'envoi de données et ne proposent donc que des *émissions non bloquantes* (ou *émissions asynchrones*) dans lesquelles l'émetteur n'est jamais bloqué. Le récepteur peut choisir entre la réception bloquante ou non bloquante. Il est alors possible de tester l'existence d'un accusé de réception qui permet de savoir que les données ont été reçues (figure A.4 page 237).

Dans tous les cas, PVM garantit que les messages envoyés sont bien reçus dans leur ordre d'émission, quelque soit le chemin qu'ils doivent parcourir dans le réseau d'interconnexion de la machine pour arriver à destination.

Le modèle des communications point-à-point étant assez limité puisqu'il ne permet qu'à deux processus de communiquer, il a été étendu à plusieurs processus : les *communications globales* (ou *opérations globales* par extension) autorisent des envois de données entre un nombre de processus quelconque participant à l'opération. PVM fournit quelques primitives pour effectuer les communications

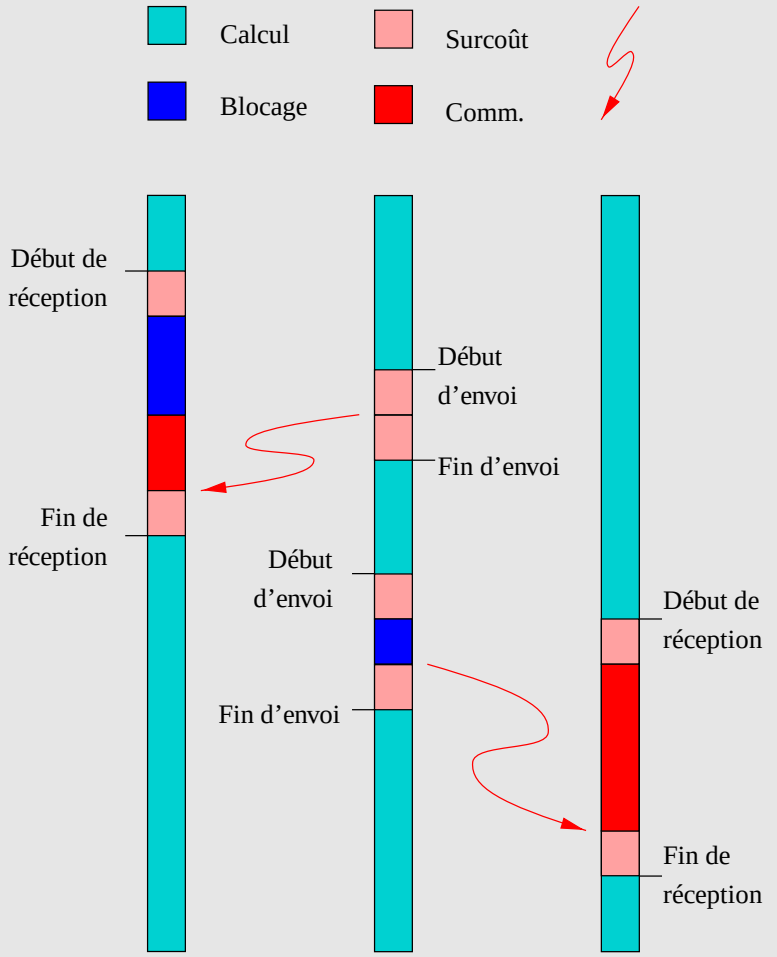


FIG. A.3. – *Communications bloquantes asynchrones. Les processus prennent rendez-vous pour communiquer,* i.e. *le premier à commencer la communication est bloqué en attendant que son interlocuteur veuille communiquer avec lui. Seul le processus récepteur attend ensuite la fin de la communication avant de reprendre son activité.*

globales les plus courantes (figure A.5 page 239), c'est-à-dire :

– la *diffusion* (*broadcast*) qui permet à un processus de diffuser des données à tous les participants (si certaines tâches ne participent pas à l'opération, on

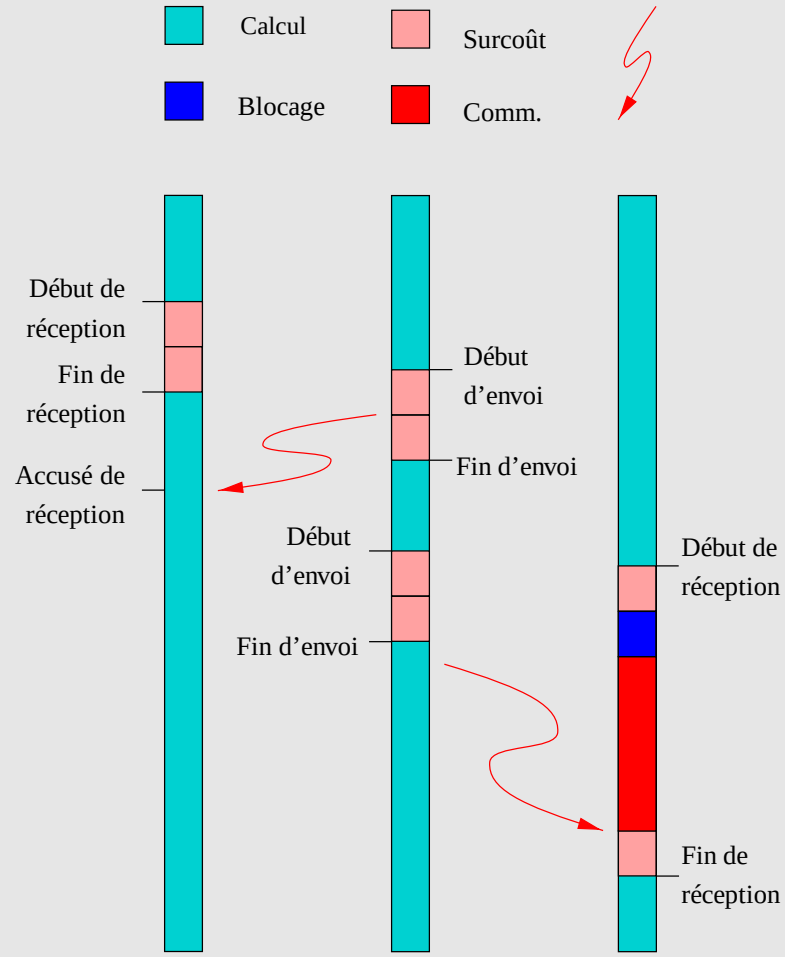


FIG. A.4. – *Émission non bloquante et réception non bloquante ou bloquante. Les processus ne prennent pas rendez-vous pour communiquer et le message est expédié à un instant quelconque. Lorsque la réception est non bloquante le processus récepteur peut tester l'existence d'un accusé de réception du message.*

parlera de *diffusion partielle — multicast*);

– la *diffusion personnalisée* qui permet à un processus de diffuser des données différentes à tous les participants ;

– la *collecte* (*gather*) qui permet à l'inverse à un processus de recevoir des

données de tous les participants ;

– la *réduction* (*reduce*) qui permet d'effectuer une opération arithmétique globale et d'en fournir le résultat à un participant donné ;

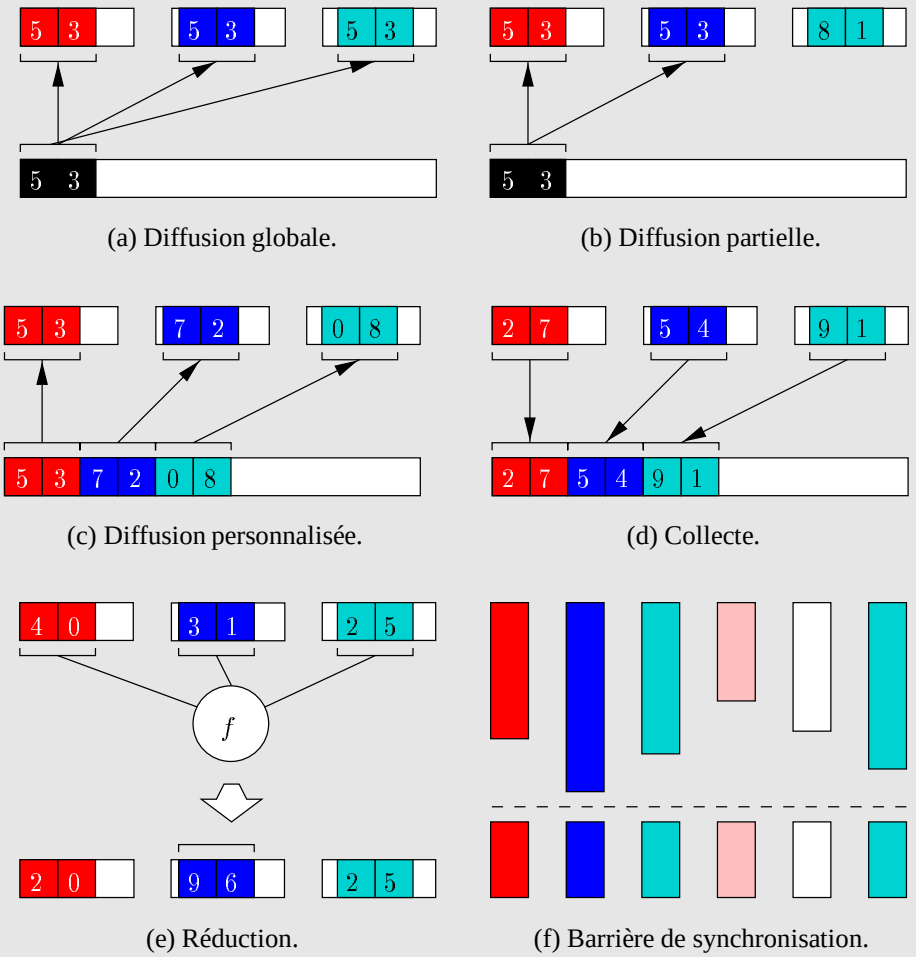
– la *barrière de synchronisation* (*barrier*) n'est pas à proprement parler une opération de communication puisqu'elle ne véhicule aucune information utile pour les processus concernés, mais elle synchronise tous les participants à l'opération.

Afin de faciliter les opérations mettant en cause plusieurs processus, il est possible en PVM de créer des *groupes* de processus et les communications globales se font au sein d'un groupe donné. Un processus pouvant entrer dans un groupe ou en sortir à tout instant, voire être membre de plusieurs groupes, il devient extrêmement facile de mettre en œuvre des opérations globales et de tirer parti de la structure de l'application pour augmenter ses performances grâce à ces opérations.

PVM est maintenant un « standard industriel » en matière de programmation parallèle grâce à ses qualités (dont une grande robustesse et une interface avec les langages C et Fortran), à sa libre diffusion et au nombre de machines sur lesquelles il fonctionne, garant de la portabilité et de la pérennité des applications développées.

Un grand nombre d'outils sont disponibles pour faciliter l'utilisation de PVM ou augmenter ses capacités : environnement visuel de développement et d'observation d'aplications parallèles (BEGUELIN, DONGARRA, GEIST, MANCHEK et SUNDERAM 1991, BEGUELIN, DONGARRA, GEIST, MANCHEK, SUNDERAM, MOORE, WADE et PLANCK 1991), support de processus légers dans une application PVM (FERRARI et SUNDERAM 1995), prise de traces et visualisation d'exécution (GEIST *et al.* 1994b) ou encore bibliothèques de fonctions mathématiques efficacement parallélisées.

Qui plus est de nombreux constructeurs de machines parallèles fournissent une version de PVM exploitant au mieux les spécificités de leurs machines (citons par exemple CRAY RESEARCH INC. (1993) ou encore IBM CORPORATION (1994) avec PVMe sur ses machines SP) : lorsque cette optimisation est correctement réalisée on peut utiliser PVM sans crainte de ne pas utiliser au mieux la (coûteuse) machine parallèle que l'on possède ou encore développer des applications sur un réseau de stations de travail en étant (presque) sûr qu'elles seront performantes une fois recompilées sur une véritable machine parallèle.



(a) Diffusion globale.

(b) Diffusion partielle.

(c) Diffusion personnalisée.

(d) Collecte.

(e) Réduction.

(f) Barrière de synchronisation.

FIG. A.5. – *Opérations simples de communication et de synchronisation globales. Les diffusions (a), (b) et (c) et le regroupement (d) sont de simples échanges de données, alors que la réduction (e) est un calcul global. Le schéma de la barrière de synchronisation (f) montre les périodes d'activités de processus appelant cette opération de synchronisation.*

A.1.3 MPI (interface d'envoi de messages)

Alors que PVM est un standard *de facto*, MPI (*Message-Passing Interface*, MESSAGE PASSING INTERFACE FORUM 1994) se veut le standard de l'avenir,

dépourvu des lacunes ou des contraintes des systèmes de programmation par envoi de message existants. D'après le MPIF (*Message Interface Passing Forum*), auteur du standard, le but de MPI est de « développer un standard qui soit largement utilisé pour l'écriture de programmes utilisant l'envoi de messages [et dont l'interface offrira] un standard pratique, portable, efficace et flexible [pour ce type d'applications] »[1].

MPI est défini par un consortium international constitué de constructeurs de machines parallèles et de chercheurs d'institutions publiques et privées. C'est ce qui garantit que MPI a effectivement été défini de façon à ce qu'il soit non seulement portable mais également efficace sur de nombreuses machines parallèles : l'interface retient des primitives qu'il est possible d'optimiser sur une machine parallèle même si elles sont complexes.

Par rapport à PVM, dont il retient quelques caractéristiques, MPI apporte un certain nombre de nouveautés que nous allons détailler dans les paragraphes qui suivent.

Les communications point-à-point en MPI se font suivant plusieurs *modes de communications* qui sont décrits ci-après du point de vue de l'émetteur :

- en *mode standard*, MPI décide s'il doit tamponner les messages ou non : s'il alloue un tampon il se peut qu'un envoi se termine avant qu'une demande de réception du message soit faite, sinon la communication est bloquante synchrone ; dans tous les cas, la communication est une *opération non locale*, c'est-à-dire qu'une réception doit correspondre à une émission pour que cette dernière soit considérée réussie ;

- le *mode tamponné* est similaire à l'allocation de tampon que nous venons d'évoquer, mais dans ce cas la communication est une *opération locale* et un envoi sans réception correspondante est réputé réussi ;

- le *mode synchrone* est semblable au rendez-vous mais un envoi se termine dès que le destinataire commence à recevoir le message, sauf si la réception est également synchrone auquel cas la communication est un véritable

1. « The goal of the Message Passing Interface simply stated is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message passing. » (MESSAGE PASSING INTERFACE FORUM 1994, page 2).

rendez-vous à la CSP ; les communications dans ce mode sont des opérations non locales.

– enfin, le *mode « prêt »* fait une communication qui n'est valide que si une réception correspondante est déjà en cours.

Au niveau des communications globales, outre des améliorations des fonctions illustrées par la figure A.5 page 239 (notamment pour leur apporter plus de flexibilité, voire des variantes) MPI ajoute quelques opérations plus complexes (figure A.6 page suivante) :

– la *collecte totale* (*all-gather*) au cours de laquelle chaque chaque processus reçoit les données de tous les autres participants ;

– l'*échange total* (*all-to-all*) qui permet à tous les processus impliqués de s'échanger des données, l'ensemble des processus disposant de l'ensemble des données à la fin de l'opération ;

– la *réduction globale* (*global reduce*) qui est une variante de la réduction dans laquelle tous les participants disposent du résultat de la réduction ;

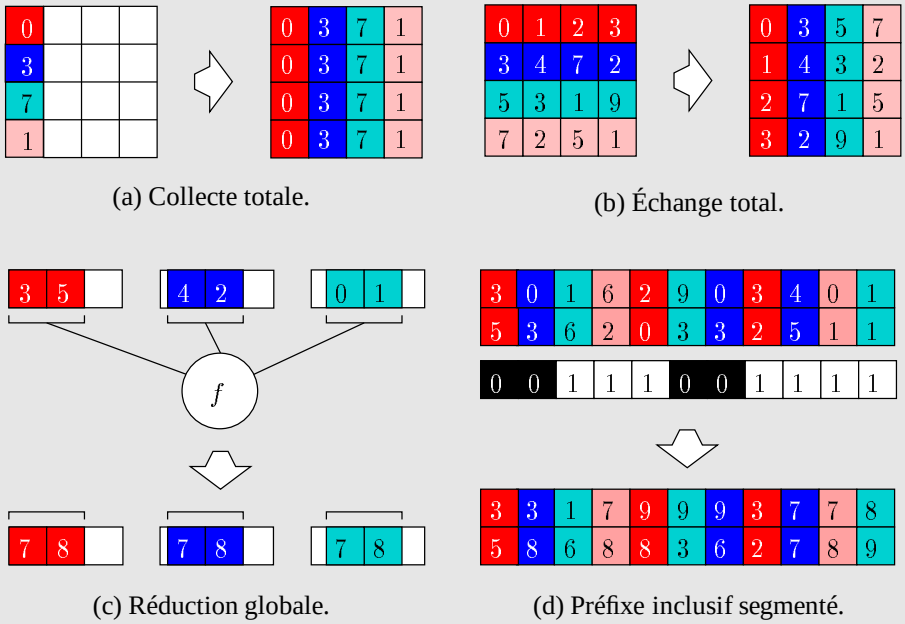
– un *préfixe parallèle inclusif* (*inclusive scan* ou *prefix*) qui renvoie au processus de rang i la réduction des valeurs fournies par les processus de rang 0 à i inclus.

Un dernier point intéressant dans MPI est celui des « *communicateurs* » (*communicators*) qui sont les objets utilisés pour spécifier les processus impliqués dans une communication. Un communicateur encapsule un certain nombre d'informations permettant de qualifier une communication :

– un *contexte de communication* qui est une sorte d'« univers » fermé dans lequel ont lieu les communications : les communications faites dans différents contextes sont absolument indépendantes et elles ne peuvent interférer ;

– un *groupe* qui définit un ordre sur les processus participant à la communication ;

– une *topologie virtuelle* qui établit une correspondance entre l'ordre propre au groupe et une topologie donnée, permettant un renommage aisé des processus ;



(a) Collecte totale. (b) Échange total.

(c) Réduction globale. (d) Préfixe inclusif segmenté.

FIG. A.6. – *Opérations complexes de communication et de synchronisation globales présentes dans MPI. La collecte totale (a) et l'échange total (b) permettent de distribuer des données en colonnes et en lignes. La réduction globale (c) et le préfixe (d) effectuent des calculs globaux.*

– des *attributs* qui contiennent des informations locales à la communication et qui sont donc disponibles pour les récepteurs.

Grâce aux communicateurs, il est facile de réaliser des bibliothèques totalement indépendantes (avec un contexte qui leur est propre), de cacher la topologie d'une machine ou encore d'étendre une notation d'envoi de messages (en ajoutant des attributs).

Il est probable que MPI sera adopté massivement par les programmeurs au fur et à mesure de la disponibilité des implantations sur les machines parallèles (et sans doute les réseaux de stations de travail), ne serait-ce qu'en raison de son support par les constructeurs (lequel se fera peut-être au détriment de la qualité de leurs implantations PVM) et de la garantie de portabilité et d'efficacité qui est

associée au standard [2]. Il existe d'ailleurs déjà plusieurs implantations gratuites de MPI pour machines parallèles et réseaux de stations de travail avec lesquelles il est possible de commencer à développer, certaines d'entre elles étant même très robustes, comme par exemple `mpich` (BRIDGES *et al.* 1995) du *National Argonne Laboratory*.

Bien qu'étant encore à peine au stade des premières implantations industrielles, MPI est en train d'évoluer afin de remédier à un certain nombre de lacunes (tels que l'absence de primitives d'entrées-sorties parallèles ou de gestion des tâches), d'intégrer des développements actuellement en cours (comme par exemple des opérations globales non bloquantes, le support d'exécution de procédures à distance ou de fils d'exécution standardisés par le MPIF) et il offrira à terme un environnement de programmation parallèle complet allant de la construction à la mise au point des applications.

A.2 Communication par variables partagées

Ce modèle qui suppose la présence d'une mémoire partagée (physique ou virtuelle) exploite un type de communication inter-processus (IPC ou *Inter-Process Communications*) basé sur le partage d'espace d'adressage entre plusieurs processus (HWANG 1993, p. 547–551).

Quelques points importants dans ce type de modèle sont la protection des *sections critiques* de code (partie d'une application qui ne peut être exécutée que par un processus à un instant donné), la synchronisation des processus et la cohérence des structures de données en mémoire.

Les outils disponibles pour la protection des sections critiques sont les *moniteurs*, les *sémaphores* et les *verrous* ; les verrous peuvent être à *attente active* ou *suspensifs* (*i.e.* les processus ne pouvant obtenir le verrou sont suspendus jusqu'à libération du verrou), les sémaphores binaires étant équivalent aux derniers. Ces différents outils peuvent par ailleurs servir à la synchronisation des processus [3]

2. Le standard prévoit un nombre assez impressionnant d'optimisations permettant d'exploiter les caractéristiques des machines les plus puissantes. Les programmeurs sont assurés que leurs applications ou bibliothèques, si elles sont soigneusement écrites, seront les plus efficaces possible sur une machine donnée.

3. Pour des raisons d'efficacité ces opérations de synchronisation sont souvent disponibles au

Le problème de la cohérence des données en mémoire est celui du conflit pouvant survenir lors de la mise à jour et de la lecture simultanée d'une structure de données par différents processus : il est possible qu'un processus lecteur obtienne une valeur incorrecte car non mise à jour. Si la machine utilisée ne dispose pas de mécanisme garantissant la cohérence des données dans un tel cas il est nécessaire de « protéger » les variables partagées en incluant leurs mises à jour dans des sections critiques.

A.3 Modèles du parallélisme de données

Le modèle du parallélisme de données étant basé sur la répartition préalable des ensembles de données en mémoire, le choix des structures de données parallèles conditionne les performances des applications et les échanges de données se font par le biais de structures de données interconnectées. La programmation par parallélisme de données favorise les opérations sur des données locales et les opérations de distribution de données (permutation, distribution, réduction, préfixe parallèle, etc.).

L'utilisation efficace de ce modèle suppose souvent le support des compilateurs par le biais de ce que l'on convient d'appeler des *extensions de manipulation de tableaux* (littéralement « array language extensions »). Par exemple, Fortran 90 (BRAINERD *et al.* 1990) dispose d'une syntaxe de manipulation de tableaux extrêmement bien appropriée à l'expression du parallélisme de données ainsi qu'à l'optimisation des manipulations de données par les compilateurs.

Les langages choisis pour la programmation par parallélisme de données sont des standards tels que Fortran ou C : l'idée est d'unifier le modèle d'exécution du programme, de faciliter le contrôle de systèmes massivement parallèles et de faciliter la migration des applications existantes vers une exécution parallèle. Les compilateurs doivent permettre aux extensions de manipulation de tableaux d'optimiser le placement des données afin de minimiser les échanges ou mouvements de données et d'exploiter au mieux le nombre de processeurs disponibles[4].

niveau du matériel dans les systèmes modernes.

4. On appelle « virtualisation des processeurs » le fait que le nombre de processeurs optimal est déterminé par l'application, à l'inverse d'une habitude ancienne où l'application est écrite en fonction du nombre de processeurs disponibles.

Les programmeurs utilisent le *découpage de tableaux* (*array sectioning*) pour adresser une région d'un tableau multi-dimensionnel en spécifiant une base, une borne supérieure et un pas. Ils disposent d'*indices vectoriels* (*vector-valued subscripts*) pour construire des permutations arbitraires de tableaux (ces indices permettent d'adresser uniquement certains éléments d'un tableau). Ensemble, ces deux techniques facilitent l'implantation des opérations de regroupement et de diffusion sur un vecteur d'indices et permettent donc une exécution efficace des applications.

On notera cependant qu'il est également possible de programmer en parallélisme de données tout en utilisant un modèle tel que l'échange de messages. Les langages ou compilateurs dédiés rendent simplement la tâche du programmeur plus facile tout en le forçant à respecter certaines contraintes. Ils permettent l'écriture des applications en utilisant uniquement les structures de données propres au domaine de l'application, sans se soucier de la façon dont elles seront échangées entre les tâches de l'application. Cette abstraction de données qui est un des points forts des langages dédiés au parallélisme de données est également un de leurs points faibles puisque l'efficacité d'une application parallèle ne peut excéder l'efficacité des systèmes automatiques qu'elle emploie.

A.3.1 HPF (Fortran hautes performances)

HPF (*High Performance Fortran*, HIGH PERFORMANCE FORTRAN FORUM 1993) est un langage de programmation parallèle plus particulièrement destiné aux applications de manipulations de données telles que le calcul scientifique ou la simulation. La spécification du langage inclut un certain nombre de nouvelles instructions, une bibliothèque d'opérations globales et la description d'un certain nombre d'*annotations* spécifiant la façon dont le compilateur doit répartir les données entre les différentes tâches.

HPF a pour but de « permettre le parallélisme de données (monoprogrammation, espace de noms global, faible synchronisation et calcul parallèle) avec des performances otpimales sur des machines MIMD ou SIMD à temps d'accès à la mémoire non-uniforme ». Le langage est défini de façon à autoriser un maximum d'optimisations sur diverses architectures.

Les programmeurs disposent de trois types d'outils pour écrire des applications parallèles : il peut placer des *annotations* indiquant au compilateur la façon

dont les données de l'application doivent être placées (en mémoire ou sur plusieurs processeurs), il peut utiliser de nouvelles instructions spécifiques au parallélisme (affectation de sections de tableaux, ou boucles sans dépendances séquentielles qui peuvent donc être optimisées) et enfin il a accès à un certain nombre de procédures effectuant des opérations globales (diffusion, regroupement, préfixe parallèle et réduction).

A.4 Programmation client-serveur et objet

Le modèle de programmation client-serveur ou objet décrit une application en terme de services et d'appels à ces services en distinguant les tâches fournissant des services de celles qui les utilisent.

Le modèle objet est une extension de ce modèle dans lequel une application est découpée en objets. Ces objets peuvent communiquer entre eux par l'appel de méthodes définissant leurs interactions possibles.

On parle également de modèle d'appel de procédures à distance (« remote procedure call ») lorsqu'on parle de ces modèles. L'idée est que les interactions entre les différentes composantes de l'application se font d'une manière familière au programmeur (*i.e.* par appel de procédures ou de méthodes) même si le code correspondant à l'interaction est exécuté sur un processeur quelconque, éventuellement différent de celui sur lequel s'exécute le code ayant provoqué l'appel.

A.4.1 Athapascan

Athapascan (PLATEAU 1994, CHRISTALLER 1994) est un modèle destiné à l'exécution de programmes parallèles portables à destination de systèmes à grand nombre de processeurs. Le principe de parallélisation d'un calcul est son découpage itératif en sous-calcul selon différentes règles. Ce découpage peut s'arrêter à tout moment et se réduire à un calcul séquentiel. Le degré de découpage ou *grain* du programme parallèle doit être adapté à celui des machines cibles potentielles étant donné que celles-ci diffèrent soit par leur grain soit par leur nombre de processeurs. Athapascan propose un moyen d'exprimer le découpage par des directives générales adaptables à toutes machines parallèles et spécifie qu'à chaque

étape du découpage un algorithme séquentiel est proposé afin de permettre l'arrêt du processus itératif de découpage.

Le modèle des applications Athapascan est un modèle en couches qui comporte deux couches (figure A.7):

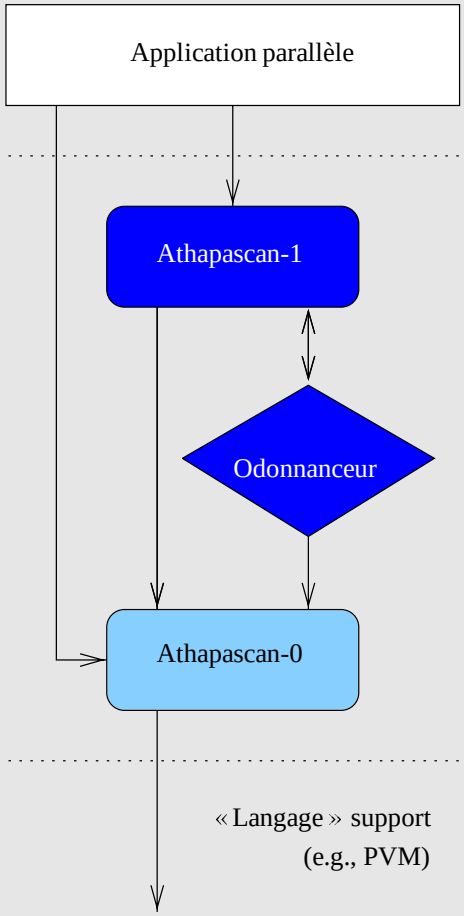


FIG. A.7. – *Vue schématique d'une application Athapascan et des interactions entre les différentes couches utilisées.*

1. le noyau exécutif parallèle Athapascan-0 définit un niveau de machine abstraite et offre des fonctionnalités pour la division en sous-calculs, la distribution des données initiales aux sous-calculs et la recombinaison des résultats

partiels en un résultat final ; il fournit les fonctions de base de la gestion de processus et de l'exécution de procédures à distance ainsi qu'un mécanisme de multi-procédures ;

2. l'équilibreur dynamique de charge d'Athapascan-1[5] comporte un ordonnanceur d'appels de procédures à distance qui choisit — en fonction d'indications fournies par le programmeur quant au coût du calcul en fonction de ses entrées — sur quel nœud effectuer un calcul en fonction de la charge du coût de son exécution sur différents nœuds potentiellement utilisables.

Ces couches interagissent du fait de l'influence du régulateur de charge sur l'appel de procédures à distance, puisqu'il en choisit le nœud d'exécution.

Une application parallèle écrite en Athapascan est un graphe d'appels de procédures. Ces procédures sont appelées à distance de manière synchrone ou asynchrone (*i.e.* en attendant ou non leur terminaison pour reprendre l'exécution de l'application). L'utilisation d'un noyau de processus légers autorise la multiprogrammation des nœuds de calcul.

Les tâches composant une application sont découpées en *services*. Ces services ont un *point d'entrée* qui peut être appelé par d'autres tâches (ce sont les procédures appelées à distance). Chaque service a un degré de parallélisme (ou nombre d'appels simultanés possibles) fixé par le programmeur. Lorsqu'un service est appelé le noyau exécutif crée éventuellement un nouveau fil d'exécution pour que le service puisse être exécuté. Si le degré maximal de parallélisme du service est atteint le processus appelant est bloqué en attendant de pouvoir être servi. Il est cependant possible d'appeler des services de façon asynchrone pour éviter d'être bloqué.

5. Athapascan-1 est destiné à aider à l'implantation de programmes parallèles. Il comporte également des outils de placement statique des tâches modélisant le programme parallèle.

Bibliographie

BEGUELIN (Adam), DONGARRA (Jack J.), GEIST (G. A.), MANCHEK (Robert) et SUNDERAM (Vaidy S.), « Graphical Development Tools for Network-Based Concurrent Supercomputing », rapport technique, Oak Ridge National Laboratory, P.O. Box 2009, Bldg. 9207-A, Oak Ridge, TN 37831-808, 1991.

BEGUELIN (Adam), DONGARRA (Jack J.), GEIST (G. A.), MANCHEK (Robert), SUNDERAM (Vaidy S.), MOORE (Keith), WADE (Reed) et PLANCK (Jim), *HeNCE: A Users' Guide, Version 1.2*, Oak Ridge National Laboratory, P.O. Box 2009, Bldg. 9207-A, Oak Ridge, TN 37831-808, 1991.

BRAINERD (W. S.), GOLDBERG (C. H.) et ADAMS (J. C.), *Programmer's Guide to Fortran 90*. McGraw-Hill, New York, 1990.

BRIDGES (Patrick), DOSS (Nathan), GROPP (William), KARRELS (Edward), LUSK (Ewing) et SKJELLUM (Anthony), *User's Guide to* `mpich`*, a Portable Implementation of MPI*, Argonne National Laboratory, 1995.

CHRISTALLER (Michel), « Athapascan-0 sur PVM 3 : définition et mode d'emploi», rapport APACHE 11, IMAG – Équipe APACHE, 46, av. Félix Viallet, 38031 Grenoble CEDEX 9, France, 1994.

CRAY RESEARCH INC., « PVM and HeNCE Programmer's Manual », rapport technique SR-2501, Cray Research Inc., 1993.

FERRARI (A. J.) et SUNDERAM (Vaidy S.), « TPVM: Distributed Concurrent Computing With Lightweight Processes », rapport technique CSTR-950201, Dept. of Math and Computer Science, Emory University, Atlanta, GA 30322, 1995.

GEIST (Al), BEGUELIN (Adam), DONGARRA (Jack), JIANG (Weicheng), MANCHEK (Robert) et SUNDERAM (Vaidy S.), « PVM3 User's Guide and Reference Manual », rapport technique ORNL/TM–12187, Oak Ridge National Laboratory, Mathematical Sciences Section, P.O. Box 2008, Bldg. 6012, Oak Ridge, TN 37831–6367, 1994.

GEIST (Al), BEGUELIN (Adam), DONGARRA (Jack), JIANG (Weicheng), MANCHEK (Robert) et SUNDERAM (Vaidy S.), *PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing*, Scientific and Engineering Computation. The MIT Press, Cambridge, Massachusetts, 1994.

HIGH PERFORMANCE FORTRAN FORUM, « High Performance Fortran Language Specification », rapport technique, Rice University, Houston, Texas, 1993.

HOARE (C. A. R.), « Communicating Sequential Processes », *Communications of the ACM*, 1978, t. XXI, n° 8, p. 666–677.

HWANG (Kai), *Advanced Computer Architecture (Parallelism, Scalability, Programmability)*, Computer Science Series. Mc Graw Hill International Editions, 1993.

IBM CORPORATION, *IBM AIX PVMe User's Guide and Subroutine Reference (IBM SP1 Documentation)*, 1994.

MESSAGE PASSING INTERFACE FORUM, « MPI: A Message-Passing Interface Standard », rapport technique, University of Tennessee, 1994.

PLATEAU (Brigitte), « APACHE : Algorithmique Parallèle et pArtage de CHargE », rapport APACHE 1, IMAG – Équipe APACHE, 46, av. Félix Viallet, 38031 Grenoble CEDEX 9, France, 1994.

V

Bibliographie commentée

ARROUYE (Yves), « Performance Evaluation of Parallel Systems: the Scope Extensible Interactive Environment », rapport APACHE 15, IMAG – Équipe APACHE, 46, av. Félix Viallet, 38031 Grenoble CEDEX 9, France, 1994.

ARROUYE (Yves), « Scope : un environnement interactif extensible pour l'évaluation des performances de systèmes parallèles », dans *RenPar'7 — Septièmes Rencontres Francophones du Parallélisme*, p. 88–91. Faculté Polytechnique de Mons, Belgique, 1995.

ARROUYE (Yves), « Scope: An Extensible Interactive Environment for the Performance Evaluation of Parallel Systems », *Microprocessing and Microprogramming Journal*, 1995. Hors-série « *Parallel Systems Engineering* ».

BEGUELIN (Adam), DONGARRA (Jack J.), GEIST (G. A.), MANCHEK (Robert) et SUNDERAM (Vaidy S.), « Graphical Development Tools for Network-Based Concurrent Supercomputing », rapport technique, Oak Ridge National Laboratory, P.O. Box 2009, Bldg. 9207-A, Oak Ridge, TN 37831-808, 1991.

Ce rapport présente HeNCE, un environnement graphique de développement d'applications parallèles. Les applications sont construites en créant des nœuds de calcul dont le code est édité par l'utilisateur, puis en organisant ces nœuds à l'aide d'opérateurs (communication, pipe line, fourche et jointure, etc…) placés graphiquement dans la fenêtre de construction du programme. HeNCE s'occupe de la génération de code pour passer les données aux nœuds et les faire s'exécuter.

HeNCE utilise PVM pour l'exécution de ses programmes, et permet deux sortes de visualisations : la première est une animation temps réel de l'exécu-

tion du programme, et la seconde utilise ParaGraph et une version de PVM instrumentée avec PICL.

BEGUELIN (Adam), DONGARRA (Jack J.), GEIST (G. A.), MANCHEK (Robert), SUNDERAM (Vaidy S.), MOORE (Keith), WADE (Reed) et PLANCK (Jim), *HeNCE: A Users' Guide, Version 1.2*, Oak Ridge National Laboratory, P.O. Box 2009, Bldg. 9207-A, Oak Ridge, TN 37831-808, 1991.

Ce document est le manuel de l'utilisateur de HeNCE, expliquant dans un premier temps les principes du langage et ses opérateurs, puis guidant l'utilisateur dans la réalisation d'une application, dans son exécution et dans la visualisation de celle-ci, le tout depuis l'environnement de programmation de HeNCE.

BEMMERL (T.), BODE (Arndt), BRAUN (Peter), HANSEN (O.), LUKSCH (P.) et WISSMÜLLER (R.), « TOPSYS—Tools for Parallel Systems (User's Overview and User's Manual », SFB-Bericht 342/25/90 A, Lehrstuhl für Rechnertechnik und Rechnerorganisation, Institut für Informatik, Technische Universität, München, Arcisstr. 21, 8000 München 2, FRG, 1990.

TOPSYS est un système parallèle ayant son propre modéle de programmation à passage de messages, MMK, et un système intégré de débogage réparti et de visualisation, VISTOP.

VISTOP présente un certain nombre de caractéristiques originales, comme le support direct des objets du langage (i.e une boîte à lettres peut être visualisée et examinée), la possibilité de faire la visualisation en ligne, celle de spécifier un modèle pour la visualisation et enfin la capacité de ne travailler que sur des objets choisis pour leur intérêt.

Le principal inconvénient de VISTOP, outre le fait qu'il est dépendant du modèle de TOPSYS, est le coût important de la spécification des modèles de visualisation, ceux-ci devant de plus être écrits pour chaque application.

CHRISTALLER (Michel), « Athapascan-0 sur PVM 3 : définition et mode d'emploi», rapport APACHE 11, IMAG – Équipe APACHE, 46, av. Félix Viallet, 38031 Grenoble CEDEX 9, France, 1994.

Ce rapport introduit dans un premier temps les concepts du noyau exécutif d'Athapascan-0 et présente la terminologie propre au « langage » Athapascan-0 ainsi que la définition d'un programme parallèle au sens d'Athapascan-0.

Il décrit ensuite l'interface de programmation proposée et un mode d'emploi de la version Athapascan-0 développée au-dessus de PVM 3.

Des exemples de programmes Athapascan-0 ainsi qu'une référence exhaustive des constructions et primitives disponibles complètent le rapport.

DONGARRA (Jack) et TOURANCHEAU (Bernard), *Environments and Tools for Parallel Scientific Computing*. SIAM Press, 1994.

FRANCIONI (Joan M.), ALBRIGHT (Larry) et JACKSON (Jay Alan), « Debugging Parallel Programs Using Sound», dans *ACM/ONR Workshop on Parallel and Distributed Debugging, Proceedings*, t. XXVI de « ACM Sigplan Notices », p. 68–75. Computer Science Department, University of Southwestern Louisiana, Lafayette, LA 70504, 1991.

Cet article justifie l'utilisation de la sonorisation par les raisons suivantes : 1/ la description du comportement d'un prgramme étant extrêmement subjective, l'ouïe est peut-être plus sensible à certaines erreurs que la vue ne l'est, 2/ la possibilité d'écouter passivement permet de remarquer plus facilement une anomalie si elle est associée à un son qui n'apparait pas dans la « mélodie » d'un comportement normal, 3/ l'oreille est capable de détecter des schémas de sons et 4/ entendre un accord permet de savoir que plusieurs événements sont produits simultanément.

L'article étudie ensuite un certain nombre d'associations entre des événements et des notes, en insistant sur le bénéficie du son dans chaque cas envisagé.

Les bénéfices observés sont 1/ une vision intéressante du comportement interne de l'application, 2/ la possibilité de facilement occulter une partie des informations en occultant un canal sonore, et 3/ la production de données MIDI permet une synchronisation aisée avec la visualisation, du fait du support de tops de synchronisation par la norme.

FRANCIONI (Joan M.), ALBRIGHT (Larry) et JACKSON (Jay Alan), « The Sounds of Parallel Programs », dans *Sixth Distributed Memory Computing Conference Proceedings*, sous la direction de STOUT (Quentin) et WOLFE (Michael), p. 570–577, 1991.

Cet article envisage l'utilisation du son pour l'évaluation des performances de programmes parallèles. Le son permet de solliciter un autre sens que la vue chez l'utilisateur, et semble offrir certains avantages. Par exemple, l'écoute, au contraire du regard, peut être passive, i.e. n'a pas besoin d'une aussi grande concentration. Il n'empêche que si un lien a été fait entre des notes et des événements tels que l'envoi et la réception de messages, une anomalie dans un schéma de communication régulier sera immédiatement détectée par la perception d'une cassure de la « mélodie » associée aux

communications, alors qu'elle a bien moins de chances d'être notée si les communications sont présentées visuellement.

Les exemples présentés dans l'article, quoique non inintéressants, portent sur des applications de taille si faible qu'il est difficile d'évaluer la validité de l'approche sonore pour l'évaluation de performance d'applications réelles. C'est néanmoins un domaine récent qui mérite d'être exploré, et qui offre une nouvelle dimension de présentation d'informations et des possibilités intéressantes lorsque le son est couplé à la visualisation graphique traditionnelle.

GEIST (Al), BEGUELIN (Adam), DONGARRA (Jack), JIANG (Weicheng), MANCHEK (Robert) et SUNDERAM (Vaidy S.), « PVM3 User's Guide and Reference Manual », rapport technique ORNL/TM–12187, Oak Ridge National Laboratory, Mathematical Sciences Section, P.O. Box 2008, Bldg. 6012, Oak Ridge, TN 37831–6367, 1994.

PVM (*Parallel Virtual Machine*) est un système permettant de considérer un réseau hétérogène de machines comme une seule ressource de calcul parallèle.

L'intérêt premier de PVM est sa grande portabilité : il est disponible sur de nombreuses machines, que celles-ci soient de type multi-processeurs ou non, et permet donc de réaliser immédiatement des applications parallèles avec les ressources dont dispose l'utilisateur, et de diffuser ensuite ces applications à une large communauté d'utilisateurs. Il est fourni avec des bibliothèques d'interface pour les langages C et Fortran 77, couvrant ainsi la quasi-totalité des utilisateurs potentiels de machines parallèles. De plus, la plupart des constructeurs de machines parallèles fournissent une version de PVM optimisée pour leurs machines, permettant au programmeur de ne pas sacrifier la performance à la portabilité de ses applications.

PVM est basé sur un modèle de processus communicants, et offre non seulement les primitives de base liées à ce modèle, mais également des primitives de gestion de groupes de processus associées à des opérations de groupes telles que les barrières de synchronisation et la multi-diffusion.

GEIST (Al), BEGUELIN (Adam), DONGARRA (Jack), JIANG (Weicheng), MANCHEK (Robert) et SUNDERAM (Vaidy S.), *PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing*, Scientific and Engineering Computation. The MIT Press, Cambridge, Massachusetts, 1994.

Cet ouvrage présente PVM (*Parallel Virtual Machine*) et fournit un cours concernant son utilisation et la façon dont il est implémenté.

Le livre commence par une introduction générale au calcul distribué en milieu hétérogène, et présente la structure de PVM et d'autres systèmes (qui sont p4, Express, MPI et Linda). Il poursuit avec l'utilisation de PVM, les techniques de base de la programmation parallèle, les interfaces utilisateurs et de programmation de PVM, puis donne des exemples de programmes typiques. Un long chapitre est ensuite dédié au fonctionnement de PVM et à la description des différents démons et protocoles qu'il utilise.

La fin de l'ouvrage couvre tout d'abord l'interface graphique XPVM, qui offre non seulement un outil interactif de contrôle de PVM mais aussi quelques vues de l'exécution d'une application en temps réel, puis se consacre au portage de PVM sur une nouvelle architecture et à la résolution des problèmes que l'utilisateur peut rencontrer en utilisant PVM.

Enfin, une référence complète des fonctions de PVM est fournie en annexe.

HACKSTADT (Steven T.) et MALONY (Allen D.), « Next-Generation Parallel Performance Visualisation: A Prototyping Environment for Visualization Development », rapport technique CIS-TR-93-21, Department of Computer and Information Science, University of Oregon, Eugene, OR 97403, 1993.

HARING (Günter), « Main Issues for Parallel Monitoring and Visualization Systems in the Future—General Discussion », dans *Workshop on Performance Measurement and Visualization of Parallel Systems*, sous la direction de HARING (Günter). Austrian Center for Parallel Computation, Springer-Verlag, Wien, Austria, 1992.

Ce document est le rapport d'une discussion abordant les points suivants : qui sont les utilisateurs d'un tel système, que faut-il mesurer, que — et quand — faut-il visualiser et quelle architecture logicielle est adaptée ?

Deux types d'utilisateurs ont été identifiés. Les utilisateurs sophistiqués (programmeurs d'applications numériques et concepteurs professionnels d'algorithmes) ont des idées précises concernant ce qu'ils veulent voir, et savent comment leur application devrait se comporter ; ils veulent la plus grande souplesse. Les utilisateurs moins sophistiqués (physiciens, biologistes, etc… qui souhaitent porter leurs programmes) veulent voir quelque chose tout de suite, et explorer leur programme sans avoir à intervenir sur leur environnement.

Les mesures devraient non plus être collectées systématiquement à bas niveaux puis abstraites à des niveaux supérieurs, mais spécifiées à haut niveau et traduites en requêtes de mesures de bas niveaux ; malheureusement,

les outils actuel ne le font pas par manque de modèles abstraits et par peur d' « oublier quelque chose d'important » lors de la prise de mesure.

En ce qui concerne la visualisation, les points les plus importants sont 1/ avoir une visualisation orientée vers l'application, 2/ avoir différentes vues, 3/ avoir de de multiples niveaux d'abstraction, 4/ posséder des techniques de visualisations appropriées et 5/ supporter les systèmes massivement parallèles. Des possibilités envisagées traitent du choix entre des vues prédéfinies et des vues que l'utilisateur peut se construire, ou encore la possibilité de comprimer intelligemment les données de façon à ce que les informations (e.g. un schéma de communication) soient toujours visible, ou l'utilisation de la réalité virtuelle et du son pour immerger l'utilisateur dans son application (!).

Enfin, les environnements du futur doivent être capable de supporter des sytèmes massivement parallèles et avoir une architecture modulaire. D'autre part, certaines applications ayant un besoin crucial de visualisation en temps réel, des outils nouveaux doivent être développés, permettant par exemple de se concentrer sur le plus important en temps réel et d'affiner le niveau de détail post mortem.

HEATH (Michael T.) et ETHERIDGE (Jennifer A.), *ParaGraph: A Tool for Visualizing Performance of Parallel Programs*, Oak Ridge National Laboratory, P.O. Box 2009, Bldg. 9207-A, Oak Ridge, TN 37831-808, 1991.

Ce document est le manuel de référence de ParaGraph, le premier outil de visualisation de programmes parallèles librement diffusé.

ParaGraph lit des traces produites lors de l'exécution d'applications utilisant la bibliothèque PICL, et permet de visualiser cette exécution au moyen de nombreuses vues, certaines étant même spécifiques à un type de réseau de communication donné. Il est également possible d'intégrer une vue personnalisée dans ParaGraph, à condition de savoir programmer avec la Xlib et d'utiliser l'interface d'accès à la trace fournie par ParaGraph.

Si ParaGraph est très facile à utiliser, si ses visualisations sont nombreuses (différentes topologies, diagrammes espac-temps, de Gantt et de Kiviat, statistiques, etc…), il a néanmoins l'inconvénient de n'être adapté qu'au modèle de mono-programmation avec échange de messages, et ses visualisations ne sont pas interactives.

HEATH (Michael T.) et ETHERIDGE (Jennifer A.), « Visualizing the Performances of Parallel Programs », *IEEE Software*, 1991, t. V, n° 8, p. 29–39.

Cet article présente l'utilisation de ParaGraph et la façon dont on peut

s'en servir pour obtenir des indications de performance (et surtout de contre-performance) de ses applications.

Les différentes vues de ParaGraph sont passées en revue, accompagnées de commentaires quant à leur utilité en fonction de ce que l'on recherche durant l'évaluation.

L'article finit par l'introduction des mécanismes de ParaGraph pour ajouter une vue, ce qui permet par exemple la construction de vues spécifiques aux applications observées.

HERRARTE (Virginia) et LUSK (Ewing), *Studying Parallel Program Behavior with Upshot*, 1992.

Upshot fournit un type de visualisation de trace, dans lequel les événements sont alignés sur des lignes parallèles correspondant au temps de chaque processus. Par rapport à ParaGraph, Upshot fait moins de choses, mais de façon plus profonde et plus générale. Upshot permet d'afficher l'état des processus au cours du temps, les événements ou l'ensemble de ces informations. Il est possible de cliquer sur un événement afin d'obtenir les informations de la trace correspondant à cet événement.

Ce document décrit le format de trace utilisé par Upshot, puis son utilisation. Upshot permet de se déplacer dans le temps en avant et en arrière, et la trace est découpée en pages correspondant à un certain nombre d'événements. Il est également possible de changer l'échelle de tracé.

On notera qu'Upshot ne fait pas d'interprétation de la trace, l'utilisateur définissant les états en indiquant les événements permettant d'entrer et sortir des états, ainsi que la couleur et le nom à associer à un état lors de l'affichage.

HOARE (C. A. R.), « Communicating Sequential Processes », *Communications of the ACM*, 1978, t. XXI, n° 8, p. 666–677.

Cet article décrit, commente et valide le formalisme de CSP, premier modèle de programmation adapté aux systèmes parallèles à mémoire distribué. Il justifie également les choix faits lors de la définition du langage et s'attache à en démontrer la validité en présentant des solutions à de nombreux problèmes classiques en algorithmique. Enfin, les perspectives pour un CSP réellement utilisé en tant que langage de programmation sont présentées et sont accompagnées de suggestion d'extension et de réflexions sur les problèmes d'implémentation.

En CSP, une application parallèle est décrite sous forme d'un ensemble de processus communicants, lesquels disposent de canaux de communications leur permettant de s'échanger des données.

Les communications en CSP sont de type point-à-point (entre deux processus) et sont synchrones, *i.e.* les processus en communication ont rendez-vous pour s'échanger des données.

CSP fournit un mécanisme de gardes permettant de n'effectuer une communication que lorsqu'une condition est vérifiée, ainsi qu'une primitive d'alternative offrant la possibilité de communiquer avec plusieurs processus de façon indéterministe.

HWANG (Kai), *Advanced Computer Architecture (Parallelism, Scalability, Programmability)*, Computer Science Series. Mc Graw Hill International Editions, 1993.

Cet ouvrage donne les principes et techniques nécessaires pour concevoir et programmer des systèmes parallèles et vectoriels, et couvre les points suivants (d'après la présentation du livre lui-même) :

–théorie du parallélisme : modèles de machines parallèles, propriétés des applications et des réseaux, lois de performances et analyse des systèmes ;

–technologie des machines de pointe : RISC, CISC, superscalaire, VLIW, processeurs super-pipelinés, cohérence de caches, hiérarchies de mémoires, pipelines de pointe et interconnexion de systèmes ;

–architectures parallèles : multi-processeurs, multi-ordinateurs, machines multi-vectorielles et SIMD, architectures extensibles, multi-fils et à flot de données ;

–programmation parallèle : modèles de programmation, langages, compilateurs, envoi de messages, développement d'applications, synchronisation, extensions parallèles des systèmes UNIX et programmation en milieu hétérogène ;

–étude de cas de systèmes réels : machines industrielles de Cray, Intel, Thinking Machine Corporation, Fujitsu, NEC, Hitachi, IBM, DEC, MasPar, nCUBE, BBN, KSR, Tera, Stardent et systèmes expérimentaux des universités de Stanford, Caltech, Illinois, Wisconsin, USC, du MIT et de l'ETL.

Le livre est clairement découpé en quatre grandes parties (théorie du parallélisme, technologie des systèmes, architectures parallèles et programmation parallèle) et comprend un grand nombre d'exercices permettant de mettre en pratique ses enseignements ainsi qu'une bibliographie particulièrement complète.

KRAEMER (Eileen) et STASKO (John T.), « The Visualization of Parallel Systems: An OverView », *Journal of Parallel and Distributed Computing*, 1993, t. XVIII, n° 2, p. 105–117.

> Cet article présente un très grand nombre d'outils permettant la visualisation de systèmes parallèles, ces outils allant du plus simple au plus complexe, du gratuit au commercial, de l'outil très spécifique à l'environnement le plus général. Il fait également un bref rappel sur les moyens de prise de traces et d'analyse de données utilisés sur divers systèmes.

MALONY (Allen D.), HAMMERSLAG (David H.) et JABLONOWSKI (David J.), « Traceview: A Trace Visualization Tool », *IEEE Software*, 1991, t. VIII, n° 5, p. 19–28.

> Traceview est un système de visualisation de traces généraliste, i.e. un outil qui ne connait pas la sémantique des traces qu'il manipule. L'utilisateur spécifie les fonctions de filtrage qu'il souhaite utiliser pour obtenir des données de la trace, ces données filtrées consituant alors une vue particulière sur la trace. Les vues sont ensuite connectées à des afficheurs présentant les données sous une forme déterminée.
>
> Il est possible de sauver les vues construites, afin de pouvoir les réutiliser sur d'autres traces présentant la même sémantique et les mêmes enregistrements, même si les afficheurs utilisés diffèrent.

MALONY (Allen D.) et NICHOLS (Kathleen M.), « Standards in Performance Instrumentation and Visualization for Parallel Computer Systems: Working Group Summary », 1991.

> Cet rapport présente les résultats d'un groupe de recherche sur ce que devrait contenir un environnement de visualisation basé sur les traces.
>
> Les composants retenus sont un langage de description de traces permettant la génération de traces auto-descriptives, associé à une bibliothèque de lecture de ces traces et à des modules de visualisation génériques, connectés à la trace par l'utilisateur par l'intermédiaire de composants de filtrage.

MALONY (Allen D.) et REED (Daniel A.), « Performance Analysis, Visualization with Hyperview », dans *IEEE Software* (NICHOLS 1990), 1990, p. 21–30.

> Hyperview est un outil de visualisation orienté objet, qui permet à son utilisateur de spécifier les visualisations qu'il désire en les construisant à partir de composantes graphiques simples. Il lui suffit ensuite de connecter

une trace à ces visualisations, par l'intermédiaire de modules d'extraction de données, pour observer l'exécution de son programme.

À notre connaissance, Hyperview est resté à l'état de prototype et n'a jamais été disponible pour une utilisation publique.

MESSAGE PASSING INTERFACE FORUM, « MPI: A Message-Passing Interface Standard », rapport technique, University of Tennessee, 1994.

MPI (*Message-Passing Interface*) est un standard de programmation parallèle par envoi de messages défini par un groupe de travail constitué de chercheurs et de représentants des grands constructeurs de machines parallèles.

Par rapport à un standard de fait comme PVM il a pour atouts sa modernité, sa plus grande puissance d'expression et une bibliothèque de communications de groupes extrêmement complète. Il offre de plus des garanties d'efficacité sur les machines parallèles actuelles et possède de grandes facilités d'extension. Il permet également l'écriture de bibliothèques d'algorithmes parallèles ayant leurs propres espaces de communication, facilitant la réentrance des codes développés ainsi que leur mise au point.

MPI est le standard qui s'impose, et des implantations gratuites sont déj'a disponibles, facilitant sa diffusion. De nombreux développements sont en cours, ainsi que la spécification et la réalisation de composants de programmation parallèles comme par exemple un système d'entrées-sorties parallèlles pour les applications MPI. Une deuxième version du standard est également en train d'être mise au point pour préciser des points tels que l'interface de gestion des tâches de l'application ou l'intégration dans un environnement de programmation parallèle.

MILLER (Barton P.), « What to Draw? When to Draw? An Essay on Parallel Program Visualization », *Journal of Parallel and Distributed Computing*, 1993, t. XVIII, n° 2, p. 265–269.

Ce bref document fournit deux exemples de l'importance de la visualisation en général, puis donne un certain nombre de raisons pour lesquelles la visualisation de programmes parallèles est difficile (absence de vrai modèle physique, difficulté pour illustrer utilement un comportement). Il définit ensuite des critères pour faire une bonne visualisation (e.g. elle doit guider l'utilisateur, être appropriée au modèle de programmation, être informative et non distrayante, etc…), puis des recommendations pour évaluer la qualité d'une représentation (utiliser de vraies applications, faire utiliser l'outil par de nombreux utilisateurs).

MILLER (Barton P.), CALLAGHAN (Mark D.), CARGILLE (Jonathan M.), HOLLINGSWORTH (Jeffrey K.), IRVIN (R. Bruce), KARAVANIC (Karen L.), KUNCHITHAPADAM (Krishna) et NEWHALL (Tia), « The Paradyn Parallel Performance Measurement Tools », rapport technique, Computer Sciences Department, University of Wisconsin-Madison, 1210 W. Dayton Street, Madison, WI 53706, 1994.

Paradyn est un environnement d'évaluation de performance original. Même s'il est basé sur la prise d'informations au cours de l'exécution d'une application, l'instrumentation de l'application est faite à la demande de l'environnement pendant son exécution. Plutôt que d'avoir de nombreuses visualisations, Paradyn en offre seulement deux, une représentation matricielle et un traceur de courbes, mais il les exploite intelligemment. (Paradyn a également une interface pour appeler des visualisations dans d'autres environnements, mais aucune démonstration de faisabilité n'a encore été faite.)

Le point le plus original de Paradyn est cependant certainement son « consultant de performance ». En s'aidant d'un modèle d'analyse de problèmes de performance développé localement, W^3 (pour *Why, Where, When*), ce consultant est capable d'identifier des problèmes de performance, d'en prouver l'existence (par sélection des indices de performances adaptés et envoi de ces derniers aux visualisations) et même de désigner la partie du code responsable du problème si celui-ci est assez simple (cette capacité est due au fait que Paradyn instrumente directement le code et sait donc exactement à quelle ligne source correspond une instruction donnée).

Le plus grand point faible de Paradyn est le coût de son portage pour l'évaluation d'un système donné, puisqu'il exige certaines caractéristiques du système d'exploitation et de l'environnement d'exécution des programmes parallèles mais nécessite également l'écriture d'un module d'instrumentation dynamique du code binaire de la machine sur laquelle s'exécutent les applications.

MILLER (Barton P.), HOLLINGSWORTH (Jeffrey K.) et CALLAGHAN (Mark D.), *The Paradyn Parallel Performance Tools and PVM*, dans DONGARRA et TOURANCHEAU (1994), chap. 1, 1994.

Ce chapitre rappelle les principes de Paradyn, un environnement d'évaluation de performance original qui permet l'instrumentation des applications au cours de leur exécution et supporte un modèle d'analyse de problèmes de performances permettant de mettre en évidence certains problèmes dans les applications.

Il présente ensuite l'architecture spécifique permettant d'intégrer Paradyn et PVM 3.3, cette intégration se traduisant par le contrôle du mécanisme de gestion de tâches de PVM par Paradyn. Un exemple d'utilisation du système est ensuite donné.

NICHOLS (Kathleen M.), « Performance Tools », *IEEE Software*, 1990, p. 21–30.

Cet article présente un certain nombre d'outils de visualisation, et permet d'effectuer une comparaison entre les différents genres d'outils disponibles.

On notera que certains outils n'étaient à l'époque que des prototypes — voire des outils en cours de développement — et que tous n'ont pas été rendus disponibles publiquement.

PLATEAU (Brigitte), « APACHE : Algorithmique Parallèle et pArtage de CHargE », rapport APACHE 1, IMAG – Équipe APACHE, 46, av. Félix Viallet, 38031 Grenoble CEDEX 9, France, 1994.

Ce rapport présente la problématique du projet de recherche APACHE dont les objectifs sont l'étude, la conception et la mise au point d'un environnement de programmation pour machines parallèles.

Cet environnement doit assurer une exécution efficace des programmes parallèles essentiellement grâce à un mécanisme portable de répartition automatique de la charge de calcul.

L'environnement devra également disposer d'outils de placement statique, d'observation et de mesure et enfin d'évaluation de performances.

REED (Daniel A.), « Scalable Performance Environments for Parallel Systems », dans *Sixth Distributed Memory Computing Conference Proceedings*, sous la direction de STOUT (Quentin) et WOLFE (Michael), p. 562–569, 1991.

Cet article présente des idées sur la façon dont des environnements d'évaluation de performance doivent être construit pour permettre leur utilisation quel que soit le parallélisme des applications observées.

La structure d'environnement retenue est un système reconfigurable, entièrement constructible par l'utilisateur, et disposant d'une bibliothèque de vues. Le support d'un degré de parallélisme quelconque se ferait à traver le filtrage de la trace pour construire des données agrégées qui seront ensuite visualisées. Les traces manipulées par un tel environnement sont autodescriptives.

L'article finit en parlant brièvement de Pablo, un environnement d'évaluation de performance conçu suivant les idées exposées précédemment.

REED (Daniel A.), AYDT (Ruth A.), MADHYASTHA (Tara M.), NOE (Roger J.), SHIELDS (Keith A.) et SCHWARTZ (Bradley W.), « An Overview of the Pablo Performance Analysis Environment », rapport technique, Department of Computer Science, University of Illinois, Urbana, Illinois 61801, 1992.

Ce rapport présente l'environnement d'évaluation de performances Pablo, le premier environnement disponible publiquement offrant un langage de programmation visuelle permettant de spécifier l'analyse à effectuer pour les performances.

Pablo est un environnement très général intégrant des visualisations et l'utilisation du son. Les traces qu'il manipule sont stockées dans un format auto-descriptif et les graphes d'analyses à utilisés sont configurés en utilisant les attributs donnés dans la partie descriptive de la trace.

L'immense inconvénient de Pablo est la difficulté de la configuration de son système de programmation visuelle, et le manque d'assistance dont dispose un utilisateur novice : la phase de configuration initiale de l'environnement, avant de pouvoir commencer à faire des évaluations de performances, est réellement trop complexe.

ROMAN (Gruia-Catalin) et COX (Kenneth C.), « A Taxonomy of Program Visualization Systems », *IEEE Software*, 1993, t. XXVI, n° 12, p. 11–24.

Cet article présente une classification des systèmes de visualisation de programmes basée sur cinq critères : 1/ la portée définit le domaine de la visualisation et les aspects qui sont examinés, 2/ l'abstraction représente le niveau de présentation, 3/ la méthode de spécification indique quels mécanismes sont utilisés pour construire la visualisation, 4/ l'interface donne les facilités fournies par le système pour la présentation et la manipulation de l'information, et 5/ la présentation concerne l'utilisation effective du système et comprend les mécanismes et heuristiques utilisés par le système pour construire les visualisations, i.e. les moyens par lesquels le système aide l'utilisateur à comprendre les visualisations.

La taxonomie présentée est compatible avec celle proposée par d'autres chercheurs, et leurs critères s'y incorporent naturellement.

SARAIYA (Nakul P.), « Simple/Care concurrent-application analyzer », dans *IEEE Software* (NICHOLS 1990), 1990, p. 21–30.

SIMPLE/CARE est un ensemble d'outils permettant de modéliser et de simuler un système parallèle complet, de la machine aux applications.

L'utilisateur dispose d'un langage permettant de décrire les instrumentations qu'il souhaite faire, et l'environnement lui permet de connecter divers

composants graphiques pour représenter les données obtenues par l'instrumentation.

SARAIYA (Nakul P.), NISHIMURA (Sayuri) et DELAGI (Bruce A.), « SIMPLE/CARE—An Instrumented Simulator for Multiprocessor Architectures », rapport technique, Knowledge Systems Laboratory, Stanford University, Stanford CA 94305, 1990.

SIMPLE/CARE est un système de simulation et de visualisation de systèmes parallèles, basé sur un simulateur à événements discrets.

SIMPLE est le simulateur, et il permet de spécifier un système complet en décrivant au niveau le plus bas les composants de ce sytème. Il permet également de définir des procédures d'instrumentation de ces composants afin d'observer leur comportement. Une fois les composants définis, ceux-ci sont assemblés pour construire le système parallèle lui-même.

CARE fournit une bibliothèque de composants tels que des interfaces réseaux, des bus, des processeurs, des processeurs de communication ou des contrôleurs mémoire. Il définit aussi des extensions parallèles aux modèles de programmation orienté objet, à variables partagées et fonctionnel, collectivement appelées LAMINA. Il offre enfin une boîte à outils d'instruments de collecte de données et de visualisation, utilisée pour construire un ensemble de vues du système et des applications l'utilisant.

SCHÄFERS (Lorenz) et SCHEIDLER (Christian), « A Graphical Programming Environment for Parallel Embedded Systems », 1992.

Ce document présente TRAPPER (TRAffonic Parallel Programming EnviRonment), un environnement graphique de programmation pour systèmes parallèles embarqués. La description du système parallèle se fait de façon graphique, et l'environnement dispose d'un outil de visualisation de l'exécution et d'un outil de visualisation pour les performances.

L'outil de visualisation de l'exécution peut fonctionner soit en temps-réel, soit en post mortem, le second cas étant nécessaire lorsque le flot d'événements devient rapide. Il présente une vue animée soit du graphe de processus, chaque processus et chaque lien de communication affichant son état, soit des changements d'états des processus en fonction du temps. Dans la première vue, il est possible de choisir entre plusieurs types d'animations (coloriage, textures, apparition de flèches, affichage de graphes ou histogrammes dans les boîtes correspondant au processus), et dans la seconde l'utilisateur peut sélectionner un ensemble de processus et voir leurs états sous forme de courbe ou de barres colorées, superposées à un diagramme espace-temps.

L'outil de visualisation pour les performances, quant à lui, présente des informations orientées matériel. Les visualisations disponibles sont une vue globale des états sur le graphe matériel, des courbes de charge des processeurs, des schémas d'ordonnancement et de communication.

STOUT (Quentin) et WOLFE (Michael), *The Sixth Distributed Memory Computing Conference Proceedings*, IEEE Computer Society Press, Portland, Oregon, 1991.

TRON (Cécile), PLATEAU (Brigitte), VINCENT (Jean-Marc), KITAJIMA (J P.), PRÉVOST (Joëlle) et RAYNAL (Patrick), « ALPES (ALgorithms, Parallelism and Evaluation of Systems), an Environment for the Performance Evaluation of Parallel Programs Implementations », dans *RenPar4 — 4es Rencontres du Parallélisme*, p. 108–111. Laboratoire d'Informatique Fondamentale de Lille, Villeneuve d'Ascq, France, 1992.

Cet article présente le projet ALPES dont le but est la réalisation d'un environnement complet pour l'évaluation de performance d'applications parallèles. L'objectif d'ALPES est de permettre l'évaluation de différents programmes, différentes machines et différentes stratégies d'implémentation des programmes sur ces machines, avec un outil unique.

Pour atteindre ce but ALPES se fonde sur deux modèles, un modèle de programmes, ANDES, qui permet de décrire quantitativement les tâches d'un programme parallèle (en indiquant par exemple ses besoins mémoire et ses temps de calcul et de communication) et un modèle de machines, MADRE, spécifiant les caractéristiques (performances et fonctionnalités) d' une machine.

À partir de ces deux modèles un programme synthétique émulamnt le comportement d'un programme réel sur la machine choisie est généré. Son exécution est tracée et les traces ainsi obtenues sont exploitées par un environnement de visualisation pour l'évaluation de performance de systèmes parallèles.