Visualisation interactive et extensible de programmes parallèles à base de processus légers

Benhur de Oliveira Stein

UNIVERSITÉ JOSEPH FOURIER – GRENOBLE 1
SCIENCES & GEOGRAPHIE

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER

Discipline : Informatique

Présentée et soutenue publiquement

par

Benhur de Oliveira Stein

Le 21 octobre 1999

Visualisation interactive et extensible de programmes parallèles à base de processus légers

---

Directeur de Thèse

Jacques CHASSIN DE KERGOMMEAUX

Composition du jury

Michel RIVEILL, *Président*
Jean-Marc JÉZÉQUEL, *Rapporteur*
José C. CUNHA, *Rapporteur*
Brigitte PLATEAU
Jacques CHASSIN DE KERGOMMEAUX

Thèse préparée au sein du Laboratoire Informatique et Distribution
(Institut d'Informatique et Mathématiques Appliquées de Grenoble)

Remerciements

Je remercie tout d'abord Jacques Chassin de Kergommeaux pour m'avoir dirigé pendant ces quatre dernières années et pour avoir transformé en français le « texte » que j'ai réussi à produire. C'est cool d'avoir un chef qui est aussi un ami.

Je remercie Brigitte Plateau pour m'avoir accueilli au sein du projet Apache, grâce à qui j'ai échappé au débogage de programmes parallèles et pour avoir participé à mon jury de soutenance.

Je tiens à remercier Michel Riveill pour avoir présidé mon jury de soutenance.

Je remercie Jean Marc Jézéquel et José C. Cunha pour avoir accepté de rapporter ce travail. Leurs remarques m'ont permis d'améliorer la qualité et la clarté de ce manuscrit.

Je remercie la CAPES et le COFECUB pour m'avoir soutenu financièrement pendant le développement de ce travail. Je remercie spécialement Marta Elias de la CAPES pour sa compétence et sa disponibilité. Je remercie aussi les gens de l'Université de Technologie de Compiègne et particulièrement Mila pour mon premier accueil en France et pour les premiers contacts avec la belle et complexe langue française.

Je remercie aussi l'Universidade Federal de Santa Maria et mes collègues du departamento de Eletrônica e Computação pour m'avoir permis de venir et pour m'avoir soutenu tout ce temps.

J'ai eu la chance de travailler dans un laboratoire avec une très bonne atmosphère d'amitié et de travail qui va certainement beaucoup me manquer. Je ne mets pas le nom de tout le monde parce que je ne saurais pas trier la liste. La rédaction de ce manuscrit aurait été énormément plus difficile sans la compagnie de tant de bons collègues. J'ai partagé le bureau avec une des personnes les plus loyales que j'ai déjà connu. Les longues nuits de fin de rédaction ont été presque agréables. Merci, Alexandre.

Je remercie Marcelo pour m'avoir convaincu a m'inscrire au mestrado à l'Universidade Federal do Rio Grande do Sul, ce qui m'a fait suivre le chemin de la recherche.

Je remercie finalement Andréa pour avoir été toujours à mon coté, pour être une grande amie en plus d'une femme merveilleuse, pour avoir été ce que je rêvais depuis que j'ai découvert qu'il y a des différences entre hommes et femmes.

Table des matières

Table des figures

Liste des tableaux

1

Introduction

Si le parallélisme s'est imposé dans le domaine du calcul hautes performances, sa diffusion est freinée par la difficulté d'utilisation des systèmes parallèles et l'insuffisance des environnements de programmation existants. Ce phénomène est aggravé par le fait que les modèles de programmation parallèle ne sont pas stabilisés et que la plupart des modèles utilisés ne restent pas valides suffisamment longtemps pour permettre le développement et l'amélioration des outils d'un environnement. L'étude qui fait l'objet de cette thèse a pour contexte le « débogage pour les performances », activité qui consiste à rechercher toutes les causes d'inefficacité des programmes parallèles afin de pouvoir les éliminer. Cette activité est importante puisque le parallélisme est utilisé afin d'obtenir des performances importantes. Le travail décrit dans cette thèse a permis la réalisation d'un environnement appelé Pajé, mot qui signifie « guérisseur » mais aussi « sorcier » dans la langue Tupi. Le principal mal qu'il s'agit ici de conjurer est l'inefficacité des applications parallèles.

Le modèle de programmation auquel on s'intéresse ici est basé sur la mise en œuvre de réseaux de processus légers communicants ou fils d'exécution, évoluant dynamiquement durant l'exécution des programmes par création et terminaison de nouveaux fils d'exécution. Deux types de communications sont possibles : par message entre nœuds distincts et par mémoire commune, en utilisant des mécanismes de synchronisation, à l'intérieur d'un même nœud. Ce modèle de programmation a été défini pour tirer parti des architectures parallèles émergentes basées sur l'utilisation de grappes (*clusters*) construites en utilisant du matériel produit en grande série. Chacun des nœuds de la grappe est composé d'un multiprocesseur symétrique à mémoire partagée. Le réseau d'interconnection entre les nœuds est un réseau rapide produit en grande série (FDDI, Myrinet, etc.) fournissant éventuellement la

possibilité d'accéder de la mémoire à distance (réseau à capacité d'adressage suivant la norme SCI par exemple). Si le nombre de processeurs de chaque nœud reste modeste (quelques unités), celui du nombre de nœuds est potentiellement important (plusieurs dizaines voire centaines).

Le modèle de programmation est conçu pour exploiter le parallélisme entre les processeurs d'un même nœud d'une part et d'autre part pour masquer la latence des communications ou des accès distants à la mémoire. En effet, en dépit des progrès rapides des débits et latences des réseaux de communication, la progression encore plus rapide des fréquences d'horloge des processeurs de série utilisés dans les nœuds augmente la durée relative des accès distants à la mémoire, relativement aux accès dans le cache interne des processeurs. Pour masquer cette latence, on exécute sur chacun des nœuds du système parallèle plus de fils d'exécution que de processeurs afin qu'un fil d'exécution prêt puisse être activé chaque fois qu'un fil d'exécution se met en attente de communication.

En outre, ce type de modèle de programmation se prête bien à la programmation d'applications « irrégulières » telles que la recherche arborescente et plus généralement de toutes les applications dont le comportement ne peut pas être prédit de façon valable et dont la programmation est facilitée par l'utilisation de fils d'exécution.

Cette étude s'est déroulée au sein du projet[1] APACHE [55] dont l'objectif est l'étude de l'ensemble des aspects liés à la mise en œuvre efficace et portable d'applications irrégulières et dont les études sont concrétisées par l'environnement ATHAPASCAN. ATHAPASCAN comporte une couche basse appelée ATHAPASCAN-0 [7] qui marie les fils d'exécution et les communications. Cette couche assure la portabilité de la couche haute de l'environnement, appelée ATHAPASCAN-1 [24, 12] qui offre un degré de portabilité supplémentaire des applications en assurant automatiquement leur ordonnancement en ligne. ATHAPASCAN-0 permet également de programmer directement des applications en bénéficiant des facilités offertes par les fils d'exécution pour masquer la latence des communications et programmer des applications irrégulières.

Dans l'environnement ATHAPASCAN le « débogage pour les performances » est basé sur le traçage logiciel des applications parallèles suivi de l'analyse des traces et de la visualisation des exécutions tracées. Les travaux présentés dans ce document concerne donc l'étude de la visualisation d'exécutions parallèles, à partir de traces d'exécution, dans le cadre de l'environnement ATHAPASCAN. L'objectif était de fournir aux programmeurs d'applications un outil les aidant à identifier les « erreurs de performances » de leurs programmes. Il ne s'agit donc pas pour l'outil d'identifier automatiquement ces erreurs en analysant les traces mais plutôt de don-

[1] CNRS-INPG-INRIA-UJF

ner au programmeur une représentation aussi claire que possible de l'exécution de ses programmes afin qu'il puisse identifier lui-même ses erreurs aussi facilement que possible.

La visualisation d'exécutions parallèles a pour objectif d'en faciliter la compréhension. De nombreux outils de visualisation ont déjà été réalisés : à partir de l'analyse de traces d'exécution ils montrent graphiquement l'enchevêtrement des communications inter-processus dans le temps (diagramme espace-temps), l'activité des processeurs (diagramme de Gantt), les taux moyens d'activité des processeurs, etc. Certains outils existants proposent un très grand nombre possible de représentations. Pourtant, aucun d'entre eux ne semble être utilisable pour visualiser l'exécution de programmes ATHAPASCAN. En effet, ces outils semblent avoir été conçus pour la visualisation de programmes réguliers, où le nombre de processus est figé durant l'exécution et habituellement égal au nombre de processeurs utilisés. En raison du nombre fixe de processeurs, il n'est pas envisageable d'utiliser ces outils en représentant chaque fil d'exécution comme un processeur. Si l'un des outils offre la possibilité de visualiser des fils d'exécution créés et détruits dynamiquement, ces fils d'exécution partagent le même nœud et communiquent exclusivement par mémoire partagée.

Ayant pris la décision de créer un environnement de visualisation pour ATHAPASCAN, il a été décidé de le doter de toutes les caractéristiques désirables pour un tel environnement. L'aptitude à représenter un nombre potentiellement important de fils d'exécution évoluant dynamiquement est liée à l'aptitude au passage à l'échelle[2] de l'environnement, soit la possibilité de visualiser l'exécution de gros programmes parallèles. En outre, l'objectif étant d'aider le programmeur autant que faire se peut, il a été décidé de lui offrir le maximum de contrôle possible sur la session de visualisation. Cette propriété a pour nom interactivité, qui permet à l'utilisateur d'inspecter le contenu des objets visualisés, de lier la représentation d'un événement au code source qui en est à l'origine, etc. Enfin, pour prendre en compte l'absence de stabilisation des modèles de programmation parallèles et offrir la possibilité d'ajouter des visualisations non encore envisagées, un soin particulier a été apporté à ce que l'environnement soit extensible facilement. Cette possibilité a permis de visualiser l'exécution de programmes ATHAPASCAN-1 sans changer l'environnement qui avait été validé en visualisant des programmes ATHAPASCAN-0.

La thèse apporte des contributions à chacun des trois points cités précédemment : extensibilité, interactivité et « scalabilité ». Différents mécanismes sont prévus pour ajouter facilement des fonctionnalités à l'environnement. La plus originale est la généricité, qui permet de définir un nouveau modèle de programmation par des

[2]Pour simplifier la rédaction, les anglicismes « scalabilité » et « scalable » seront utilisés pour traduire les mots anglais *scalability* et *scalable* dont une traduction plus correcte serait la périphrase qui précède

commandes placées en tête d'une trace d'exécution. Il est ainsi possible de visualiser un nouveau modèle d'exécution parallèle *sans avoir à modifier aucunement l'environnement de visualisation.* Une solution originale a également été trouvée pour combiner l'interactivité avec les propriétés d'extensibilité et de « scalabilité », le principal problème posé par la mise en œuvre de l'interactivité étant la consommation mémoire, dont l'importance doit être limitée autant que possible pour permettre le traitement de « grosses » applications. L'aptitude au passage à l'échelle a été combinée avec l'interactivité en donnant la possibilité aux utilisateurs de passer dynamiquement d'un niveau d'abstraction à un autre, un niveau d'abstraction élevé montrant l'activité d'un plus grand nombre de nœuds du système au détriment de la précision de la représentation, qui ne peut être obtenue qu'en se concentrant sur un sous-ensemble du système. Plus peut-être que des contributions à chacun des trois points cités ci-dessus, la principale contribution de cette thèse est sans doute d'avoir réussi à les combiner au sein d'un même outil.

La thèse à fait objet de quelques publications, [18, 17, 67] et un article est en cours d'évaluation [19].

Le manuscrit de thèse comporte 7 chapitres :

Chapitre 2 : « Visualisation de programmes parallèles »

Ce chapitre présente tout d'abord la problématique de la thèse. La visualisation est placée dans le contexte de la mise au point pour les performances et de ses relations avec la collecte d'informations. Les étapes du traitement des traces d'exécution permettant la visualisation sont ensuite décrites avant la mise en évidence des propriétés que devraient posséder les environnements de visualisation d'exécutions parallèles, à savoir l'extensibilité, l'interactivité et l'aptitude au passage à l'échelle. Le chapitre se termine par une brève description des environnements de visualisation les plus représentatifs existants actuellement.

Chapitre 3 : « Présentation de Pajé »

Le chapitre commence par une description du contexte dans lequel s'est déroulée la thèse, à savoir le projet APACHE et l'environnement ATHAPASCAN. L'interface de programmation du noyau exécutif ATHAPASCAN-0, dont les traces ont servi à valider l'outil Pajé, est brièvement décrit. Les fonctionnalités de Pajé sont ensuite présentées en montrant l'utilisation qui en a été faite pour améliorer les performances d'une application de dynamique moléculaire. La présentation de Pajé se termine par une description de son architecture dont l'objectif est de limiter les références en avant dans le reste de la thèse en définissant l'ensemble des concepts qui seront employés par la suite. En effet, les chapitres qui suivent sont consacrés aux contributions de la thèse dans les domaines de l'extensibilité, l'interactivité et la « scalabilité » ; or l'un des principaux apports de la thèse est d'avoir réussi à combiner ces propriétés

au sein d'un même environnement et il est donc difficile d'étudier chacun des concepts en isolation.

Chapitre 4 : « Extensibilité »

Pour faciliter son extensibilité, l'environnement de visualisation est conçu comme un ensemble de composants (modules) connectés au sein d'un graphe flot de données. De nouvelles fonctionnalités peuvent ainsi être apportées en développant un nouveau composant et en le connectant aux composants existants. Pour faciliter le développement de nouveaux composants et leur connexion au reste de l'environnement, des règles de développement et des protocoles de communication entre composants sont proposés. Pour augmenter encore la flexibilité de l'environnement, un simulateur — ce composant est le plus dépendant de la sémantique du modèle de programmation — générique est proposé. Paramétré par des commandes insérées dans la trace d'exécution, il permet à l'environnement de visualiser de nouveaux modèles de programmation ou des données spécifiques à une application sans nécessiter aucune modification.

Chapitre 5 : « Interactivité »

L'interactivité suppose un accès très rapide aux données de visualisation puisqu'un réaffichage peut être provoqué par le moindre mouvement de souris de l'utilisateur. L'interactivité implique donc le maintien en mémoire des données de la trace ainsi que des données produites par le simulateur. Ce chapitre présente en détail les différentes organisations de données qui ont été conçues pour accélérer la recherche des données et compare la complexité de la recherche des objets à visualiser selon ces organisations. Si dans tous les cas la complexité dépend linéairement du nombre d'objets en mémoire, on peut montrer que l'organisation choisie réduit énormément le facteur constant apparaissant dans la complexité de la solution.

Chapitre 6 : Aptitude au passage à l'échelle, « Scalabilité »

Ce chapitre rappelle tout d'abord que la « scalabilité » de la visualisation ne saurait dépasser celle des autres outils dont elle dépend, telle la prise de traces. La « scalabilité » impose également des contraintes sur les structures de données. La principale contribution dans ce domaine est de donner aux programmeurs de « grosses applications » la possibilité de naviguer selon leurs besoins entre différents niveaux d'abstraction. Cette possibilité est obtenue par la définition de techniques de filtrage agissant sur les données manipulées par l'environnement de visualisation.

Chapitre 7 : Conclusion

Les principales contributions de la thèse sont rappelées et les perspectives qu'elle ouvre sont esquissées.

2

Visualisation de programmes parallèles

La visualisation de l'exécution de programmes parallèles fait partie de la phase de mise au point pour les performances du cycle de développement des programmes parallèles. Il s'agit d'une phase importante puisque l'obtention de rendements élevés est le but principal de l'utilisation des systèmes parallèles. En théorie, la recherche des erreurs de performances devrait être une phase incontournable du cycle de développement des programmes parallèles. Pourtant, en pratique, Reed [57] remarque que la plupart des programmeurs se contentent de performances « acceptables », et ne cherchent à déboguer leurs programmes pour les performances que si ce n'est pas le cas. La visualisation de l'exécution d'un programme parallèle constitue une aide importante pour la compréhension de son comportement et par conséquent pour l'amélioration de ses performances.

La mesure de performances peut être utilisée dans au moins deux contextes différents. Le premier se produit quand on veut un contrôle dynamique de l'exécution d'un programme parallèle. Dans ce cas, les données collectées sont analysées en ligne parce qu'une réaction rapide est nécessaire. C'est la cas de quelques outils surveillant des activités de systèmes d'exploitation, comme *xload, perfmeter, top*, etc. des systèmes Unix [68], utilisés par les ingénieurs système pour contrôler les ressources informatiques. C'est également le cas des outils de surveillance employés dans les systèmes temps réel. Pour une analyse globale du comportement des programmes parallèles, pendant un cycle de mise au point pour les performances, les données de performance sont le plus suivant analysées *post mortem*.

La visualisation d'exécutions de programmes parallèles dont il est question dans cette thèse correspond à la deuxième situation :

- Elle a pour principal objectif d'aider les programmeurs à améliorer les performances de leur programme parallèle ou, dit autrement, à en éliminer les « erreurs de performances ».
- Elle est basée sur l'analyse de traces d'exécution.

La visualisation de l'exécution d'un programme implique donc les phases suivantes : collecte de données, analyse de ces données pour les réduire et pour calculer des indices de performance et présentation des indices de performance au programmeur. Dans le cadre de cette thèse, nous nous intéressons essentiellement aux deux derniers aspects. Néanmoins, en raison de la complémentarité entre la collecte des données et leur analyse, on aborde dans ce chapitre les différentes techniques de collectes de données et les raisons qui ont conduit, dans le projet APACHE, à choisir un traceur logiciel.

L'analyse de données transforme les traces d'exécution en indices de performance. Afin d'aider à la mise au point pour les performances, un grand nombre d'indices de performance doivent être fournis pour que les programmeurs puissent détecter et réduire les problèmes de leurs programmes. La variété des façons de les présenter est encore plus riche. Il y a des données statistiques et des données comportementales ; des données constantes et des données variables ; des données unidimensionnelles et multidimensionnelles, etc. Des utilisateurs différents trouveront une visualisation plus claire qu'une autre, pour visualiser les mêmes données. Un outil sera plus apprécié si son utilisateur peut choisir comment présenter chaque donnée. Ce choix est aussi important en raison de la variété de la quantité des données. Une visualisation adaptée à la présentation de dix valeurs ne l'est pas forcement pour en présenter dix mille. Le contraire est aussi vrai. La quantité de données pouvant être très importante, les outils de visualisation doivent être capables de supporter ou de s'adapter à des tels volumes de données. Il est souvent nécessaire de réduire les données pour pouvoir les visualiser. Il est important dans ce cas que l'utilisateur puisse visualiser les données plus détaillées correspondantes aux parties du programme révélées suspectes par l'analyse des données globales. La facilité de naviguer parmi les différents niveaux d'abstraction des données constitue une partie essentielle de la puissance de l'outil.

De ce qui précède découlent deux caractéristiques importantes pour l'outil de visualisation : l'interactivité et la « scalabilité » . La scalabilité est la possibilité de représenter un grand nombre de données. Une des solutions possibles pour la réaliser, et qui sera développée dans cette thèse, est la possibilité de grouper certaines données dans des représentations plus abstraites — lorsque l'utilisateur désire une vision d'ensemble — sans perdre la possibilité d'« éclater » certaines données agrégées pour en connaître l'origine. L'interactivité est la propriété de l'environnement

de visualisation qui permet à son utilisateur d'effectuer ce type de manipulation — mais également de nombreuses autres telles que le déplacement d'avant en arrière dans le temps d'exécution et l'identification des relations entre les objets affichés — à partir de la représentation visuelle qui lui est donnée.

Une autre caractéristique importante des environnements de visualisation est l'extensibilité. Cette propriété est rendue nécessaire par la conjonction de plusieurs facteurs. Le principal est l'importance de la quantité de travail qui doit être investie pour construire un environnement de visualisation et qui constitue une importante motivation pour que cet environnement ne tombe pas rapidement en obsolescence. Or, les modèles de programmation parallèles n'étant pas figés, tout outil qui ne serait lié qu'à un modèle particulier de programmation parallèle risquerait de devenir obsolète aussi rapidement que ce modèle. Un autre facteur provient de l'absence de maturité des outils de visualisation qui conduit à de nombreuses expérimentations. Il convient donc de pouvoir ajouter facilement de nouvelles fonctionnalités ou visualisations à un environnement.

Dans ce chapitre, nous nous intéresserons tout d'abord aux indices de performances, dont la présentation intelligible constitue l'objectif des outils de visualisation qui nous intéressent. Nous aborderons ensuite la collecte d'informations et en particulier le traçage logiciel, les problèmes qu'il pose et les solutions qui ont été proposées à ces problèmes. Les différentes phases de l'analyse des données collectées, en vue de leur visualisation seront ensuite présentées. Le chapitre se termine par la présentation de quelques outils de visualisation de programmes parallèles existants, parmi les plus représentatifs des outils existants. Les lacunes de ces outils, relativement aux critères énoncés, feront apparaître les motivations qui ont présidé le travail présenté dans cette thèse et à la réalisation de l'environnement Pajé.

2.1 Indices de performance

De très nombreux indices de performance sont susceptibles d'intéresser le programmeur d'une application parallèle. Ces indices peuvent être divisés en deux classes principales [9] :

Temps d'exécution qui doit être réduit autant que possible. L'objectif de la mise au point pour les performances est le plus souvent de réduire le temps d'exécution d'un programme. Cet indice peut être décomposé en plusieurs mesures, le temps utilisé pour l'exécution des diverses parties des programmes —procédures, protocoles de communication, etc. Le temps d'exécution total d'un programme parallèle dépend souvent non seulement du temps d'exécution sur chaque processeur mais aussi des synchronisations et communi-

cations entre les plusieurs processeurs. Dans ce cas, il peut être important d'identifier le chemin critique pour ne pas optimiser inutilement certaines parties du programme.

Taux d'utilisation des ressources indiquent si le programme a bien utilisé les ressources mises à sa disposition. Si nous considérons des taux d'utilisation des processeurs, les programmeurs doivent savoir quel pourcentage de temps est utilisé par divers surcoûts, tels que l'exécution de codes de synchronisation, la création, l'arrêt ou l'ordonnancement de tâches, les communications ou l'inactivité. Les taux globaux d'utilisation des ressources peuvent indiquer un problème tel qu'une basse utilisation des processeurs par l'application ou une inactivité importante. Cependant, pour identifier l'origine de tels problèmes il est souvent nécessaire d'employer des données plus détaillées. Par exemple, une inactivité importante pourrait indiquer la présence d'un goulot d'étranglement, dont l'origine pourrait être un manque de parallélisme dans le programme, une faible performance de l'ordonnanceur de tâches, une utilisation excessive de synchronisations, etc.

Donc, les indices globaux seuls ne sont souvent pas suffisants pour découvrir les causes d'une mauvaise performance. Des données plus détaillées sont nécessaires, qui peuvent être des valeurs partielles des mêmes indices, prises à intervalles de temps réguliers ou entre phases de l'application, afin de mieux identifier temporellement la partie du programme qui présente des problèmes. Souvent il faut faire appel à des données encore plus détaillées, avec lesquelles on peut déceler le comportement du programme dans le temps, avec des détails de l'exécution tels que la séquence des états des composants du programme, les échanges de messages, les diverses synchronisations, etc.

L'exécution d'un programme parallèle peut être surveillée à plusieurs niveaux possibles d'abstraction : matériel, système d'exploitation, environnement de programmation et application. Intuitivement, le niveau application est le plus significatif pour les programmeurs, puisque c'est le seul endroit où ils peuvent contrôler ou ajuster des paramètres. Cependant, il se peut que les faibles performances d'une application ne puissent être expliquées qu'en observant l'impact des choix de programmation sur l'environnement de programmation ou sur le système d'exploitation, pendant l'exécution de l'application.

2.2 Collecte d'informations

Afin de pouvoir visualiser et analyser l'exécution d'un programme parallèle, des données caractérisant cette exécution doivent être collectées pendant son exécution. Plusieurs techniques peuvent être utilisées pour observer les exécutions de

programmes parallèles afin d'en mesurer les performances [57] : l'échantillonnage, le comptage, le chronométrage ou le traçage. Chaque technique représente un compromis différent entre le niveau de détail des données obtenues et le niveau de perturbation introduite dans le programme.

2.2.1 Techniques d'observation

L'**échantillonnage** consiste à vérifier la valeur du compteur ordinal du processeur pendant l'exécution du programme observé, à intervalles de temps fixes. Le temps passé dans une partie du programme (procédure) est supposé proportionnel à la quantité de fois que le compteur ordinal se trouve dans cette procédure lors d'un échantillonnage. Les outils basés sur l'échantillonnage sont très utilisés pour la mesure de performances de programmes séquentiels [29]. Ce type d'outil est en revanche peu adapté à la mise en évidence des obstacles à l'efficacité des programmes parallèles : les mesures rendent mal compte des goulots d'étranglement et par ailleurs, les techniques d'échantillonnage ne permettent pas toujours l'évaluation des temps de communication, tels que l'attente d'un message (qui ne serait détectable que dans le cas d'une attente active). L'échantillonnage pose aussi des problèmes pour l'observation de phénomènes de faible durée, à cause de la fréquence peu élevé de prise des échantillons.

Le comptage, le chronométrage et le traçage sont basés sur l'occurrence d'*événements*. On suppose que les processus des applications émettent des événements observables dont l'enregistrement constitue une trace d'exécution. Les événements auxquels on s'intéresse dépendent du type d'observation effectuée mais comportent le plus souvent des émissions et réceptions de messages par les processus, le début et fin des actions « intéressantes » réalisées par les processus ainsi que des événements « définis par les utilisateurs ».

Le **comptage** consiste donc à enregistrer le nombre d'événements de chaque type qui ont eu lieu pendant l'exécution du programme. Les informations de comptage sont en général associées à d'autres. La plupart des outils qui génèrent un profil d'exécution [29] produisent le nombre de fois que chaque fonction observée a été exécutée en plus du temps pris par ces exécutions.

La plupart des processeurs possède des compteurs matériels, qui comptent des événements tels que les instructions exécutées, défauts de page, interruptions, etc. Ces compteurs peuvent aussi être enregistrés pendant l'exécution du programme. De bas niveau et difficilement portables d'une architecture à autre, ils sont néanmoins parfois utiles pour la compréhension des défauts de performance.

Le **chronométrage** consiste à mesurer le temps passé entre deux événements et à enregistrer la somme de temps passé dans chaque état « intéressant » du pro-

gramme. Par exemple, pour obtenir le temps cumulé passé dans chaque procédure du programme, la date de l'événement « fin de procédure », soustrait à la date de l'événement « début de procédure » correspondant, est accumulé dans un accumulateur associé à chaque procédure. À la fin de l'exécution du programme, ou périodiquement pendant l'exécution, l'ensemble de ces accumulateurs est enregistré.

Le **traçage** consiste à enregistrer chacun des événements d'une exécution parallèle. Le traçage est la technique d'observation de l'exécution de programmes la plus générale. À partir d'une trace d'exécution, il est possible de déduire toute information susceptible d'être obtenue avec les autres techniques, pas seulement sur l'exécution totale du programme, mais aussi sur une partie quelconque de cette exécution. Si on désire obtenir des informations sur le temps passé dans les différentes parties des programmes (procédures), il suffit d'associer des événements aux débuts et fins de ces procédures. Pour détecter la durée d'une communication, il suffit d'enregistrer les émissions et réceptions de messages. De plus, le traçage permet d'obtenir un historique détaillé du comportement du programme observé et non pas simplement un résumé de son exécution. Pour ces raisons, la plupart des outils de visualisation de programmes parallèles sont basés sur la collecte et l'analyse de traces d'exécution [60, 72, 13, 33].

Cependant, le traçage souffre de plusieurs inconvénients. Tout d'abord, il peut être très intrusif si on collecte une information trop détaillée, ce qui peut affecter non seulement le temps d'exécution mais aussi le comportement du programme observé. Un autre problème, provient de ce que la validité des données enregistrées peut être corrompue par l'interaction avec le système d'exploitation. Par exemple, le temps écoulé dans une procédure correspond à la différence entre la date de la fin et du début de l'exécution de la procédure *seulement si* le processus qui exécute la procédure n'a pas été suspendu pendant son exécution. Par conséquent, si les outils basés sur le traçage (et le chronométrage) conviennent bien à la mesure des exécutions parallèles sur des systèmes peu chargés, ils peuvent échouer à obtenir des données précises relatives à des systèmes multi-utilisateurs chargés. Le traçage est parfois couplé au comptage ou au chronométrage au niveau système pour obtenir des données plus précises.

Un autre problème important posé par le traçage est le volume des données générées, pour les programmes de grande taille (en nombre de processeurs et en temps d'exécution). La capture de données détaillées relatives à l'exécution d'un programme sur un millier de processeurs peut facilement atteindre 25 giga octets à l'heure [57]. Non seulement la capture d'une telle quantité de données est problématique mais sa présentation ne l'est pas moins.

Une façon d'attaquer le problème est de réduire les données à la génération. Plusieurs méthodes sont possibles, comme diminuer dynamiquement le niveau de détail de l'enregistrement lorsque la fréquence d'enregistrement s'avère trop impor-

tante [60], l'identification de groupes de processeurs avec un comportement assez proche pour enregistrer le comportement d'un seul [57, 53], l'identification automatique de problèmes de performance avec la prise de traces seulement dans les endroits considérés « suspects » [50], la reconnaissance de motifs de comportement [74] et enfin le raffinement du traçage pratiqué sur plusieurs exécutions, un enregistrement plus détaillé étant pratiqué dans les endroits où une exécution précédente moins détaillée a révélé un problème de performance.

2.2.2 Collecte de traces

Comme défini ci-dessus, le traçage est l'enregistrement des événements d'une exécution dans un fichier de trace. Comme c'est le cas pour toutes les techniques d'observation, le traçage peut être réalisé à différents niveaux d'abstraction. La qualité des traces indique la fiabilité des informations enregistrées. Elle est affectée principalement par le manque d'horloges globales dans les systèmes répartis — ce qui rend difficile de trouver l'ordre des événements se produisant sur différents nœuds— et par l'effet de sonde — qui peut changer le comportement des exécutions tracées relativement aux exécutions non tracées. La qualité des traces dépend de la technique de traçage. Cette section examine les techniques de traçage ainsi que les facteurs qui affectent la qualité de l'information tracée.

2.2.2.1 Techniques de traçage

Il existe plusieurs techniques de collecte de traces [37] : matériel, logiciel et hybride. Le traçage logiciel est la technique de traçage la plus répandue parce qu'elle est bon marché et assez facile à mettre en œuvre. Cependant il rend difficile l'obtention des traces de haute qualité en raison de l'absence d'horloges globales dans la plupart des systèmes répartis et en raison de l'intrusion provoquée par l'enregistrement et le transport de la trace d'exécution par le système parallèle, simultanément à l'exécution du programme observé.

2.2.2.2 Qualité de l'information tracée

Dans le cas idéal, les événements enregistrés seraient datés avec une horloge globale de précision infinie et il n'y aurait aucune intrusion due au traçage. Cependant ce n'est pas le cas en général et particulièrement dans le cas du traçage logiciel. L'inexistence d'horloge globale dans les systèmes parallèles répartis peut produire des événements incohérents entre eux. L'intrusion due au traçage peut changer le comportement de l'exécution du programme observée.

Manque d'horloge globale Le manque d'horloge globale dans un système parallèle à mémoire répartie peut avoir comme conséquence des incohérences entre les événements enregistrés s'ils sont datés en utilisant les horloges locales des processeurs, non synchronisées. Par exemple, la date de réception d'un message entre processeurs différents peut être inférieure à sa date d'émission. De telles incohérences rendent difficile ou impossible l'analyse des traces d'exécution par des outils d'évaluation de performances. Sur des traceurs matériels ou hybrides, ce problème est résolu en utilisant le matériel dédié [37]. Sur des traceurs logiciels, ce problème peut être résolu par la mise en place d'un algorithme logiciel de correction d'horloge [39, 46].

Intrusion due au traçage Comme n'importe quelle technique d'observation, le traçage perturbe l'exécution des programmes parallèles observés. Il est difficile d'estimer l'intrusion du traceur puisqu'elle dépend du programme tracé et du nombre d'événements tracés. En cas de traçage hybride ou matériel, on suppose que l'intrusion reste limitée à une fraction dérisoire du temps d'exécution et qu'elle ne change pas assez le comportement de l'exécution du programme observé pour empêcher l'utilisation des traces d'exécution par les outils d'évaluation de performance [37].

L'intrusion du traceur ne peut pas être négligée dans le cas du traçage logiciel. Plusieurs propositions ont été faites pour modéliser et compenser l'intrusion des traceurs logiciels [48, 73, 45, 46]. L'objectif de telles compensations est de produire, à partir de la trace d'exécution correspondant au comportement perturbé de l'application, une trace d'exécution qui soit le plus proche possible de la trace d'exécution « idéale », qui serait obtenue par une instrumentation non-intrusive.

Lorsque le programme tracé possède des fils d'exécution multiples, l'intrusion est encore accrue à cause des synchronisations entre les fils d'exécution, nécessaires pour garantir l'intégrité des données générées.

2.2.2.3 Tamponnage et extraction de données

La quantité de données tracées dépend du nombre d'événements tracés : elle peut être limitée quand le traçage est restreint aux événements de communication ; une quantité énorme de données peut être produite si des mesures plus détaillées sont nécessaires ou si l'on fait une mauvaise utilisation de l'outil de traçage. Pour faire face à ce problème, quelques systèmes d'observation, tels que Pablo [58], ajustent dynamiquement la fréquence d'enregistrement des traces et peuvent remplacer le traçage d'événements par un comptage quand la fréquence d'occurrence des événements tracés devient trop haute. De toutes façons, une quantité potentiellement grande de données de trace doit être enregistrée et extraite du système parallèle. Di-

vers compromis peuvent être considérés entre le surcoût en mémoire, résultant de l'allocation de grandes tampons de trace, et le surcoût temporel, résultant du temps dépensé pour le transfert des données au disque ou de l'utilisation d'algorithmes de compression de données [46].

Le problème de tamponnage est augmenté par la multiprogrammation légère. Si les fils d'exécution partagent des tampons, il faut payer un fort surcoût pour en synchroniser l'accès. Avec des tampons individuels par fil d'exécution, il faut gérer une quantité peut-être importante de tampons, à des taux d'utilisation peut-être très différents, mais où des synchronisations n'apparaissent que lorsqu'un tampon est plein.

2.2.2.4 Format du fichier de traces

Le format du fichier de traces décrit les données qui sont enregistrées dans le fichier, dans quel ordre et sous quelle forme (binaire, textuelle). Pour pouvoir lire le fichier de traces et en extraire les informations, il faut en connaître le format. De nombreux formats de trace ont été développés à ce jour, presque autant que d'outils d'observation. Cette prolifération de formats différents et incompatibles vient de la disparité des environnements d'exécution. Une autre raison pour le développement d'un format de traces spécifique est d'enregistrer le minimum d'informations nécessaires pour exprimer les actions du programme dans son environnement, résultant dans des fichiers de traces aussi économiques que possible.

Quelques formats de trace ont au contraire été développés avec pour objectif la généralité : être à même d'exprimer les différents paradigmes des environnements de programmation. Pour que cette généralité soit possible, le format des traces lui même ne peut contenir aucune sémantique des informations qu'il transporte. Seule la façon de représenter les données doit être définie par le format, pas sa signification. Ces formats sont généralement auto-descriptifs : la structure des événements est définie dans les en-têtes des fichiers de trace. Le format auto-descriptif qui semble avoir le plus de succès est SDDF [3], développé pour l'outil Pablo [60].

L'avantage principal d'avoir un format de traces unifié réside dans la possibilité d'utilisation de différents outils d'analyse et visualisation à partir d'un même fichier de traces, évitant des conversions de format, qui en plus d'être fastidieuses et de consommer de l'espace disque, ne sont pas toujours sans pertes : certains concepts présents dans un format peuvent être absents dans un autre format et donc supprimés par une tentative de traduction. L'utilisation de différents outils reste néanmoins limitée à des analyses simples, qui ne nécessitent pas la reconstitution des états du programme analysé. Pour faire une telle reconstitution, l'outil aurait besoin de bien connaître le modèle de programmation utilisé par le programme ou encore d'être conçu pour cela : c'est le cas de Pajé dont la conception générique

permet l'utilisation pour différents modèles de programmation sans nécessiter de conversion de traces (voir chapitre 4).

2.3 Analyse des traces et visualisation

Pour être utile, l'information collectée pendant l'exécution d'un programme parallèle doit être présentée au programmeur sous une forme qui lui soit compréhensible. Dans la plupart des cas, cette information n'est pas prête à être visualisée telle qu'elle a été collectée, et doit subir un certain traitement. Ce traitement vise d'une part, à reconstituer la séquence des états du programme observé et à en calculer des indices de performance, et d'autre part à réduire les données et à sélectionner celles qui seront effectivement visualisées à chaque instant. Les traitements des données peuvent donc être décomposés en plusieurs étapes : lecture des traces d'exécution, qui extrait l'information du fichier de traces, simulation, qui reproduit les états du programme à partir des événements de la trace, analyse de données, qui produit des indices de performance et fait la réduction des données, filtrage, qui sélectionne les informations à visualiser et visualisation, qui présente ces informations à l'utilisateur. Toutes ces étapes n'existent pas dans tous les outils de visualisation ou ne sont pas facilement distinguables. Leur présence dépend des données produites pendant l'exécution et du type de visualisation voulue ; leur distinction dépend de l'architecture interne de construction de l'outil. Ces étapes sont décrites dans les sections qui suivent.

2.3.1 Lecture des traces

Avant de pouvoir faire quoi que ce soit avec les données collectées lors de l'exécution du programme, l'outil de visualisation doit accéder à ces données. Dans la plupart des cas, ces données sont mises à disposition de l'outil de visualisation dans un fichier (ou plusieurs), dans un ordre qui correspond en général à l'ordre chronologique de génération des données.

Le principal problème que pose la lecture de traces est la connaissance du format du fichier de traces (voir section 2.2.2.4). La plupart des outils sont capables de lire un ou quelques formats de traces et il faut donc que l'environnement d'exécution utilisé pour le développement du programme parallèle fournisse une trace dans le format accepté par l'outil ou bien que la trace fournie soit convertie à ce format, avant de pouvoir être analysée.

La lecture de traces dépend du format des fichiers de traces. Cependant, elle n'est pas forcément dépendante de la sémantique des données lues. Plus cette séparation est marquée et la dépendance de la sémantique est confinée, plus l'outil sera

portable facilement à des environnements d'exécution différents.

2.3.2 Simulation de l'exécution

La simulation a pour but de reconstituer le comportement des divers objets du programme parallèle dont les actions ont été enregistrées. Les objets considérés dépendent de l'environnement de programmation utilisé par le programme parallèle, et représentent en général les processeurs, les processus et les liens de communication, mais peuvent être aussi des fils d'exécution, des objets de synchronisation ou des objets significatifs pour l'application, définis par le programmeur.

La simulation n'est pas toujours présente dans les outils de visualisation. Elle existe quand on veut restituer la séquence des états du programme observé à partir de l'enregistrement des événements tracés. En revanche, si les données collectées ne sont pas des événements mais des compteurs ou des valeurs d'indices de performance, ou bien si on ne veut pas visualiser les changements d'états, la simulation n'a pas de raison d'être.

Pour restituer le comportement des objets du programme parallèle à partir des événements disponibles dans la trace d'exécution, il faut connaître la sémantique de ces événements. Cette sémantique est souvent spécifique de l'environnement de programmation utilisé et est en général codée dans l'outil d'observation, restreignant la possibilité de réutilisation de l'outil pour des environnements de programmation sémantiquement différents. Ce problème de portabilité est plus important quand la simulation est plus détaillée. Une façon de limiter ce problème est le développement d'un simulateur générique, configurable avec une description de la sémantique des données de la trace. De la même façon qu'un format de traces autodescriptif peut contenir différentes données, une trace contenant la description de la sémantique de ses événements peut contenir des données qui ont des sémantiques différentes. Si en plus on permet l'ajout, par l'application, de descriptions sémantiques des données à visualiser qu'elle produise, l'outil de visualisation peut présenter des données de l'application sur mesure. Une définition complète de la sémantique des événements dans le fichier de traces permettrait de réaliser un outil vraiment portable et capable de représenter les données spécifiques de l'environnement de programmation et des applications, augmentant l'expressivité de l'outil et ses possibilités de survie.

Un exemple limité de description sémantique existe dans le format de traces Alog et l'outil de visualisation upshot [35]. Dans cet outil, on peut définir des nouveaux états des processus (pour distinguer les différentes phases du programme, par exemple). La définition d'un nouvel état est entièrement contenue dans le fichier de trace, l'outil de visualisation n'ayant aucune connaissance à priori de son existence.

Le concept n'a malheureusement pas été poussé plus loin, restant limité à la simple définition de nouveaux états des processus.

La généricité a été mieux prise en compte dans Pajé, où il est possible de décrire la sémantique de toute une famille de modèles de programmation par un ensemble de commandes insérées dans des fichiers de traces par le programme à analyser. La généricité dans Pajé est abordée dans le chapitre 4.

Cette fonctionnalité n'est pas sans coût. Un simulateur générique est généralement moins performant qu'un simulateur spécifique, et la quantité de données nécessaires à son fonctionnement peut aussi être supérieure. Ces surcoûts sont cependant justifiables par l'expressivité conquise, et peuvent être diminuées avec un simulateur hybride, capable de simuler les événements spécifiques d'un environnement de programmation donnée et aussi de comprendre et de simuler une description sémantique et les événements associés.

2.3.3 Analyse de données

L'analyse de données a pour but de diminuer la quantité d'information devant être visualisée et analysée par le programmeur. Cette diminution est réalisée en produisant des indices de performance qui synthétisent les données brutes produites par la simulation ou enregistrées pendant l'exécution du programme. De nombreux indices de performance peuvent être obtenus en analysant ces données brutes, tels que le volume de données échangées durant l'exécution du programme, le degré d'utilisation des processeurs, l'évolution des files de messages, divers comptages, des calculs statistiques, etc. Cette analyse des données est en général simple, la difficulté réside dans le choix des indices à produire [49], pour qu'ils soient représentatifs de l'exécution observée et susceptibles d'aider le programmeur à mieux comprendre son programme et à en améliorer les performances.

Dans certain outils, l'analyse de données est plus poussée, essayant de trouver automatiquement des problèmes de performance, et même ses causes. C'est le cas de l'outil Paradyn qui ajuste dynamiquement la collecte de données en fonction d'une analyse en ligne (voir section 2.4.4).

La recherche de problèmes de performance est plus efficace quand on a d'abord une vision globale de l'exécution qui peut être raffinée sélectivement au fur et à mesure que la recherche avance. De cette façon, on n'est pas submergé par trop de détails impossibles à comprendre. L'utilisateur n'accède qu'aux détails nécessaires à la compréhension de ce qu'une visualisation plus abstraite lui a présenté. Afin de permettre ce type de recherche « descendante », il est important de garder une liaison entre les données abstraites produites par l'analyse des données et les données brutes qu'elles synthétisent. L'existence de telles liaisons simplifie la construction

de visualisations interactives, capables de présenter la correspondance entre objets présentés dans des visualisations à niveaux d'abstraction différents.

2.3.4 Filtrage des données

Le filtrage vise à diminuer la quantité de données a visualiser. Au contraire de l'analyse, qui produit des nouvelles données à partir des données existantes, le filtrage ne fait qu'occulter certaines données, empêchant leur visualisation. L'idée est d'enlever les informations qui ne sont pas intéressantes à un instant donné et de laisser celles que le sont. Ce choix n'est pas facile, et est en général laissé au programmeur, moyennant une offre de filtres permettant d'effectuer ce choix selon différents critères.

Comme la production des données, le filtrage peut se faire à plusieurs moments : pendant la collecte des données, pendant la lecture des traces, pendant la simulation, pendant l'analyse ou bien juste avant l'affichage. Plus on filtre tôt, moins on aura de données à traiter et à visualiser et plus rapide sera leur traitement. En revanche, le choix de ce qu'il faut filtrer est plus difficile, vu qu'on ne sait pas toujours d'avance ce que l'utilisateur va vouloir visualiser. Les conséquences du filtrage en amont du processus d'analyse sont donc plus lourdes puisque irrécupérables : si on découvre après coup qu'un filtrage réalisé pendant la collecte des traces a été trop « filtrant » et qu'on a besoin de données éliminées, il faut réexécuter le programme avec une configuration des filtres plus « perméable ».

Le filtrage tardif, au contraire, permet un choix plus sélectif, au prix d'un surcoût en temps de traitement et en espace de stockage. Le grande avantage du filtrage tardif (couplé à la production de données abstraites par l'analyse de données) est de permettre la construction d'un outil qui, à partir d'une vision simple, facilement compréhensible, des données abstraites de l'exécution, permet à son utilisateur d'explorer des données de plus en plus détaillées, là où il le juge nécessaire. De cette façon, l'utilisateur n'est pas forcé d'examiner des données qui ne sont pas nécessaires ou voulues, une grande source d'insatisfaction avec les outils de visualisation [57].

Il faut donc trouver un bon équilibre entre ne pas avoir trop de données à traiter et en avoir suffisamment pour une visualisation de qualité.

2.3.5 Présentation des données

La présentation des données est l'interface entre le programmeur et les données qui caractérisent son programme. La qualité des visualisations offertes par un environnement et la facilité d'interaction avec ces visualisations sont cruciales pour

leurs capacités à aider le programmeur à comprendre, trouver et corriger les problèmes de son programme.

Reed propose une classification des présentations de performances basée sur trois axes [62] : dynamique, cardinalité et ordinalité (voir figure 2.1).

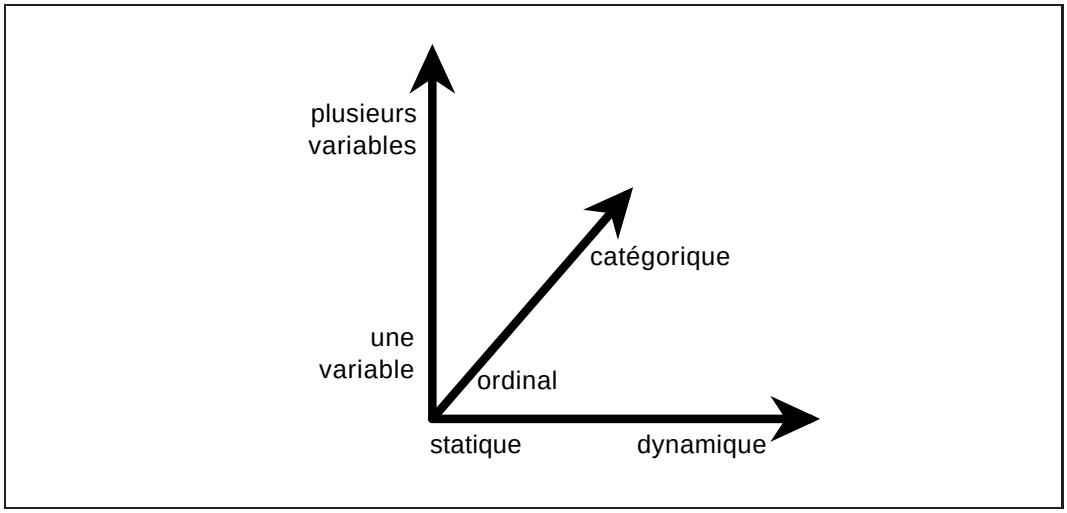

Figure 2.1 *Classification des présentations de données de performance [62]*

À un extrême de l'axe de la dynamique, on a des visualisations statiques, qui montrent une vision unique, inchangeable de la performance d'une application. Un profil d'exécution tel que peut générer l'outil `gprof` [29] en est un exemple. Une visualisation dynamique, par contre, met sa représentation à jour pour refléter un changement dans les données. Un exemple de visualisation dynamique est un graphe de l'utilisation moyenne des processeurs pendant un intervalle donnée de l'exécution, qui est réactualisé quand l'utilisateur change l'intervalle de temps sélectionné.

Suivant l'axe de la cardinalité, les présentations se distinguent selon le nombre de variables présentées simultanément. D' un côté on trouve les techniques de présentation des valeurs simples, comme les jauges ou les cadrans. Les techniques pour montrer les relations entre plusieurs variables, comme par exemple les histogrammes ou les nuages de points (« scatterplots »), sont à l'autre extrême de cet axe.

Le troisième axe, ordinalité, représente d'un côté les visualisations de données numériques, ordonnées, tels que le nombre de messages échangés ou le taux d'utilisation d'un processeur, et de l'autre côté les données non-numériques et sans ordre total, qui représentent des catégories, comme les états d'un processus (actif, bloqué, etc.) ou l'identification des procédures exécutées [65, 15].

Un cas particulier et très commun de visualisation à plusieurs variables se présente quand une de ces variables est le temps. Ces visualisations montrent l'évolution des valeurs d'autres variables pendant l'exécution du programme observé. Ces visualisations sont appelées temporelles ou historiques, en opposition aux visualisations instantanées, qui montrent les valeurs des indices de performance à un instant quelconque dans le temps d'exécution. Un exemple de visualisation temporelle est le diagramme espace-temps, trouvé dans beaucoup d'outils de visualisation. Dans un tel diagramme, en général en deux dimensions, un axe représente le temps et l'autre représente l'« espace », l'évolution des valeurs des variables visualisées dans le temps.

Une visualisation temporelle peut encore être animée, quand les données sont affichées au fur et à mesure qu'elles sont produites par le simulateur ou par le lecteur de traces [72, 33, 1]. La visualisation représentant l'exécution du programme défile devant l'utilisateur, qui peut, en général, en contrôler la vitesse de défilement, mais pas la direction (le défilement ne peut pas revenir en arrière dans le temps). Dans le contexte de cette thèse et à la différence de l'environnement Pajé, une telle visualisation ne sera pas considérée comme interactive. D'autres visualisations temporelles présentent une vision statique de l'exécution [14, 75, 44], permettant à l'utilisateur de se déplacer dans cette représentation en avant et parfois en arrière, et d'en changer l'échelle du temps, pour se concentrer sur une partie de l'exécution ou pour avoir une vision d'ensemble. Ces visualisations nécessitent le maintien d'un plus grand nombre de données en mémoire, et en général acceptent plus d'interaction avec l'utilisateur qu'une visualisation animée.

Une visualisation doit être claire, l'utilisateur doit comprendre facilement les données qui lui sont présentées. L'outil de visualisation peut faciliter cette compréhension, en étant interactif. L'interaction peut par exemple aider l'utilisateur à trouver les relations entre les données présentées, entre vues différentes de la même donnée ou entre les données présentées et le code source du programme parallèle qui a généré ces données. Par exemple, une visualisation qui montre le taux d'utilisation d'un processeur, avec sa valeur maximum et minimum, serait plus utile si elle présentait aussi l'endroit du programme (ou à l'endroit dans une autre visualisation) correspondant à l'utilisation minimale.

Il existe aussi des outils qui stimulent d'autres sens que la vision, comme la sonorisation de traces d'exécution [22] ou l'immersion dans un univers virtuel [63].

2.4 Exemples d'outils de visualisation

Un grand nombre d'outils ont déjà été développés pour visualiser l'exécution des programmes parallèles. Cette prolifération est due en part à la complexité de

la tâche et à la varieté de domaines et de perspectives à considérer, mais aussi au manque de possibilités d'extension des outils. Nous présentons ici quelques uns de ces outils. Des présentations plus exhaustives peuvent être trouvées dans la bibliographie [51, 8, 40, 31].

2.4.1 Paragraph

ParaGraph [34, 33, 32] est un des premiers outils de visualisation à avoir été disponible gratuitement. Il a popularisé l'utilisation de la visualisation pour l'évaluation des performances des applications parallèles. ParaGraph emploie des traces d'exécution produites par la bibliothèque de programmation parallèle PICL [26, 27].

ParaGraph propose un grand nombre de visualisations ; quelques unes sont montrées dans la figure 2.2. En ce sens, il est encore un des outils les plus complets

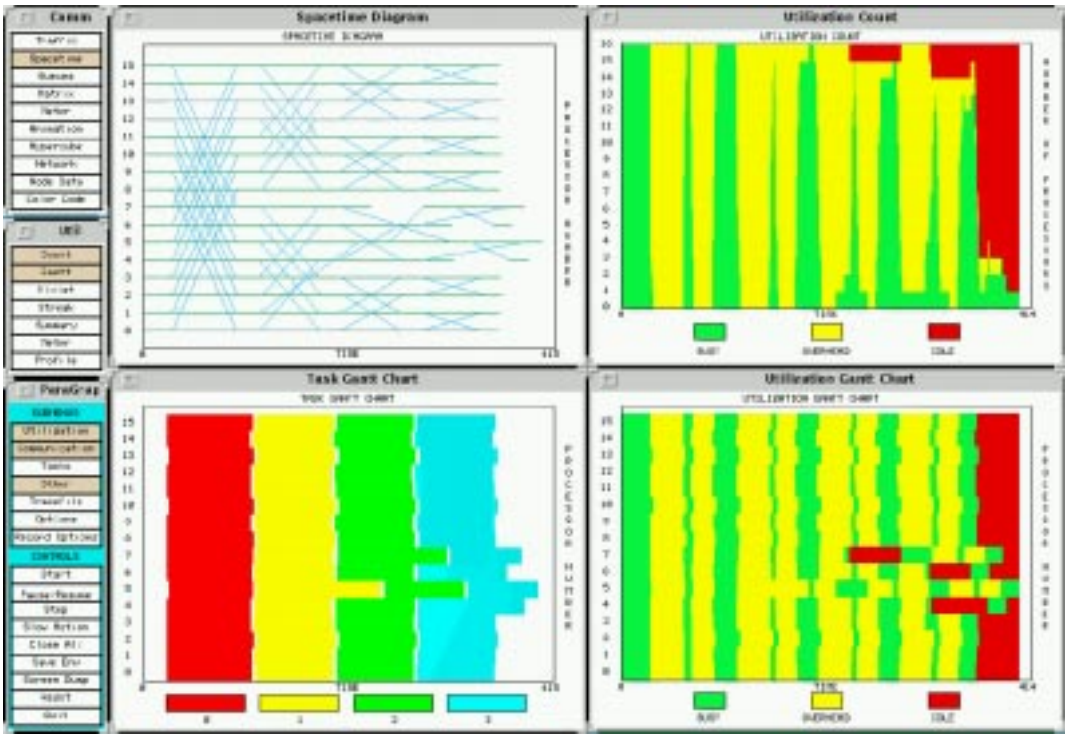

Figure 2.2 *Quelques visualisations de ParaGraph*

existants actuellement. Ses visualisations sont dynamiques : elles sont actualisées au fur et à mesure que le fichier de traces est lu et simulé. Ses visualisations peuvent être classées en quatre catégories :

- Les visualisations d'**utilisation** montrent les degrés d'utilisation des processeurs de la machine, individuellement et dans leur ensemble. Elles servent à déterminer l'efficacité de l'utilisation des processeurs et la distribution de la charge de calcul.
- Les visualisations de **communication** montrent le flux des messages, sur toute la machine, sur les processeurs individuels, entre paires de processeurs ou encore les messages individuels. Son utilité est de déterminer la fréquence, le volume, la distribution des communications et l'existence de congestions dans les tampons et les liens de communication.
- Les visualisations de **tâches** offrent des possibilités de personnaliser quelques visualisations de ParaGraph. Une tâche est une unité de calcul définie par le programmeur en plaçant dans son programme des appels à des fonctions qui identifient son début et sa fin. On peut visualiser le nombre de tâches exécutées, l'utilisation des processeurs par les tâches, entre autres.
- Les visualisations **diverses**, qui ne rentrent pas dans les autres catégories ou qui appartiennent à plus d'une, comme le chemin critique, une représentation textuelle de la trace ou des sommaires statistiques.

La plupart des visualisations de ParaGraph sont instantanées, et montrent l'état du programme dans l'instant courant de la simulation (et éventuellement les valeurs maximales ou minimales des indices montrées). Quelques visualisations sont temporelles, et montrent l'évolution des états du programme jusqu'à l'état courant de la simulation, pendant un temps qui dépend de la taille de la fenêtre utilisée pour la représentation et de l'échelle temporelle utilisée. Ces visualisations temporelles défilent automatiquement pour permettre l'affichage des nouvelles données au fur et à mesure que la simulation progresse.

Paragraph n'est pas très « scalable », puisque la plupart des visualisations deviennent difficiles à comprendre quand le nombre de processus est grand (et ce nombre ne peut pas dépasser 512). L'outil n'offre pas la possibilité de visualiser un sous-ensemble des processeurs. Il n'a pas non plus de filtres, pour limiter la quantité des données à présenter.

Les possibilités d'extension de Paragraph restent limitées. Il est possible d'ajouter une visualisation. L'extension de son modèle de programmation, ou son adaptation à un modèle différent de celui de PICL n'est pas une tâche simple, l'outil est construit d'une façon monolithique et il n'existe pas de division nette entre la lecture, la simulation et la visualisation des données. L'outil ne supporte donc pas de concepts étrangers à PICL, comme les fils d'exécution et primitives de synchronisation autres que les échanges de messages. L'existence de tâches définies par l'utilisateur et le nombre important de visualisations offertes diminue un peu la nécessité de l'étendre.

Les visualisations ne sont pas interactives, l'outil n'offre aucune aide pour trouver une correspondance entre données présentées sur fenêtres différentes ou entre les données présentées et le code source.

2.4.2 Pablo

Pablo [59, 60] est un environnement de visualisation puissant et flexible. L'architecture de Pablo est basée sur un graphe extensible de composants qui peuvent être connectés par l'utilisateur pour produire un outil de visualisation donné. L'utilisateur contrôle totalement la sélection des informations du fichier de traces à traiter, la sélection du traitement à réaliser sur ces informations et comment l'information résultante sera affichée. Cette liberté a un prix : l'apprentissage de Pablo est bien plus difficile que celui de la plupart des autres outils.

Un grand nombre de composants ont été développés, pour transformer et pour afficher des données. Les modules de transformation permettent des opérations telles que sélection, opérations arithmétiques, logarithme, fonctions statistiques, création de vecteurs ou d'autres collections de données à partir de données scalaires. Parmi les modules de visualisation on trouve graphique en colonne, nuage de points, ligne de contour, camembert, diagramme de Kiviat. Une des bases de l'extensibilité de Pablo est l'absence de connaissances sémantiques des données par ses modules. Un module arithmétique, par exemple, peut transformer n'importe quelle donnée numérique, indépendamment de sa signification. Le choix des opérations à réaliser et sur quelles données sont de la responsabilité de l'utilisateur, qui connaît la signification de ses données et les opérations capables de produire des résultats significatifs.

La construction d'un schéma de visualisation consiste à sélectionner, connecter et configurer les modules dans un graphe flot de données sans cycles. La figure 2.3 montre un graphe d'analyse simple. Chaque module reçoit des données du module qui le précède, les transforme ou les affiche et éventuellement les envoie au(x) module(s) qui lui succède(nt). Même ce graphe d'analyse simple est assez compliqué à mettre en œuvre ; un grand nombre de composants doit être connecté et configuré. La configuration d'un module est, par exemple, la sélection des champs des registres d'entrée qui seront traités et de la façon dont seront appelés les champs des registres de sortie.

Pablo manipule des fichiers de traces dans un format auto-descriptif appelé SDDF (*Self Defining Data Format*) [3]. Ce même format est utilisé pour la communication entre modules. Il permet l'extension de Pablo par le développement et intégration de nouveaux composants. Cette facilité d'extension a permis le développement d'un certain nombre de visualisations expérimentales [56, 61], comme

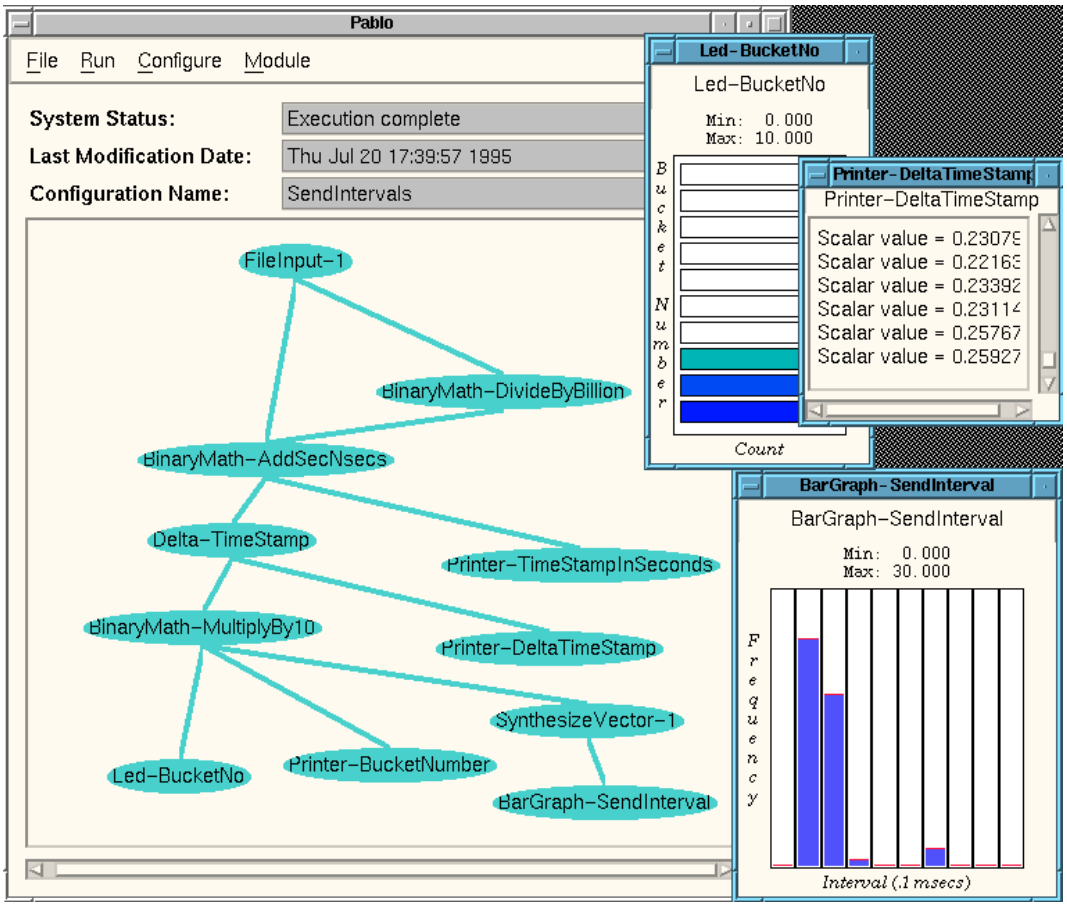

Figure 2.3 *L'environnement Pablo*
La fenêtre principale montre un graphe d'analyse qui calcule l'intervalle entre envois de messages. Les petites fenêtres à la droite montrent le résultat, trois visualisations différentes des intervalles d'une exécution.

des projections tridimensionnelles d'un espace de données multi-dimensionnelle et même l'immersion dans un tel espace à l'aide d'outils de « réalité virtuelle ». Des composants de sonorisation de traces ont aussi été développés, comme complément à la visualisation.

Pablo ne contient pas de simulateur, un de ses buts étant l'indépendance de l'environnement de programmation, la possibilité de traiter les données produites dans n'importe quel environnement de développement parallèle. Il ne peut donc pas restituer la séquence d'états de l'application, se limitant a montrer des données plutôt statistiques. En utilisant Pablo, on visualise des indices de performance, et non le comportement du programme. Ses visualisations ne sont pas interactives, et il n'y a pas de liaison avec le code source ou même entre visualisations.

L'extension de Pablo par ajout de composants est assez simple, les modules étant

indépendants les uns des autres et communiquant exclusivement par des données dans le format SDDF. Malheureusement l'intégration d'un nouveau composant à Pablo nécessite une modification dans le programme pour que le composant soit reconnu, puisque tous les modules doivent faire partie du module exécutable de Pablo.

La bibliothèque d'instrumentation de programmes de Pablo possède des mécanismes de régulation de la quantité de données enregistrées ; l'enregistrement d'événements dans la trace (traçage) est commuté vers un enregistrement de compteurs (comptage) quand le volume de traces dépasse un certain seuil.

2.4.3 SvPablo

La nouvelle version de Pablo, appelée SvPablo [20] a une philosophie un peu différente de la philosophie originale de Pablo. Un seul composant combine l'instrumentation, l'analyse et l'organisation du code source de l'application et des fichiers de trace. Les indices de performance produits durant l'exécution du programme instrumenté sont accumulés par fonction et par ligne du programme source (les fonctions et les lignes concernées, ainsi que les indices pouvant être choisis par le programmeur). Ces indices sont : le nombre de fois que la ligne ou la fonction a été exécutée, le temps d'exécution cumulé, le nombre de messages envoyés ou reçus, le nombre d'octets envoyés ou reçus, les dates d'émission et de réception des messages. Si l'architecture de la machine le permet[1], des compteurs matériels peuvent aussi être enregistrées, ce qui permet de calculer des données tels que la performance en mégaflops, le nombre de défauts de cache, etc. Les données collectées ne sont enregistrées qu'à la fin de l'exécution du programme. Les fichiers générés sont donc plus petits qu'un fichier de traces, et leur taille est indépendante du temps d'exécution du programme.

Un autre objectif de SvPablo est d'être indépendant du langage et de l'architecture cibles du programme analysé. Les informations capturées sur chaque processeur pendant l'exécution du programme sont ensuite assemblées et corrélées au code source de l'application, générant un fichier de performances. Ce fichier représente une exécution du programme et sera utilisé comme entrée pour l'interface graphique de SvPablo.

Les données ainsi recueillies sont visualisées dans l'interface graphique de SvPablo, qui les présente selon un code de couleurs (voir figure 2.4). À côté du nom de chaque fonction du programme (à droite dans la figure, indiqué dans la fenêtre « Routines in Performance Data ») il y a deux carrés colorés qui correspondent au nombre de fois que la fonction a été exécutée et le temps cumulé de ces exécutions.

[1] Qui a été implanté pour l'architecture MIPS R10000

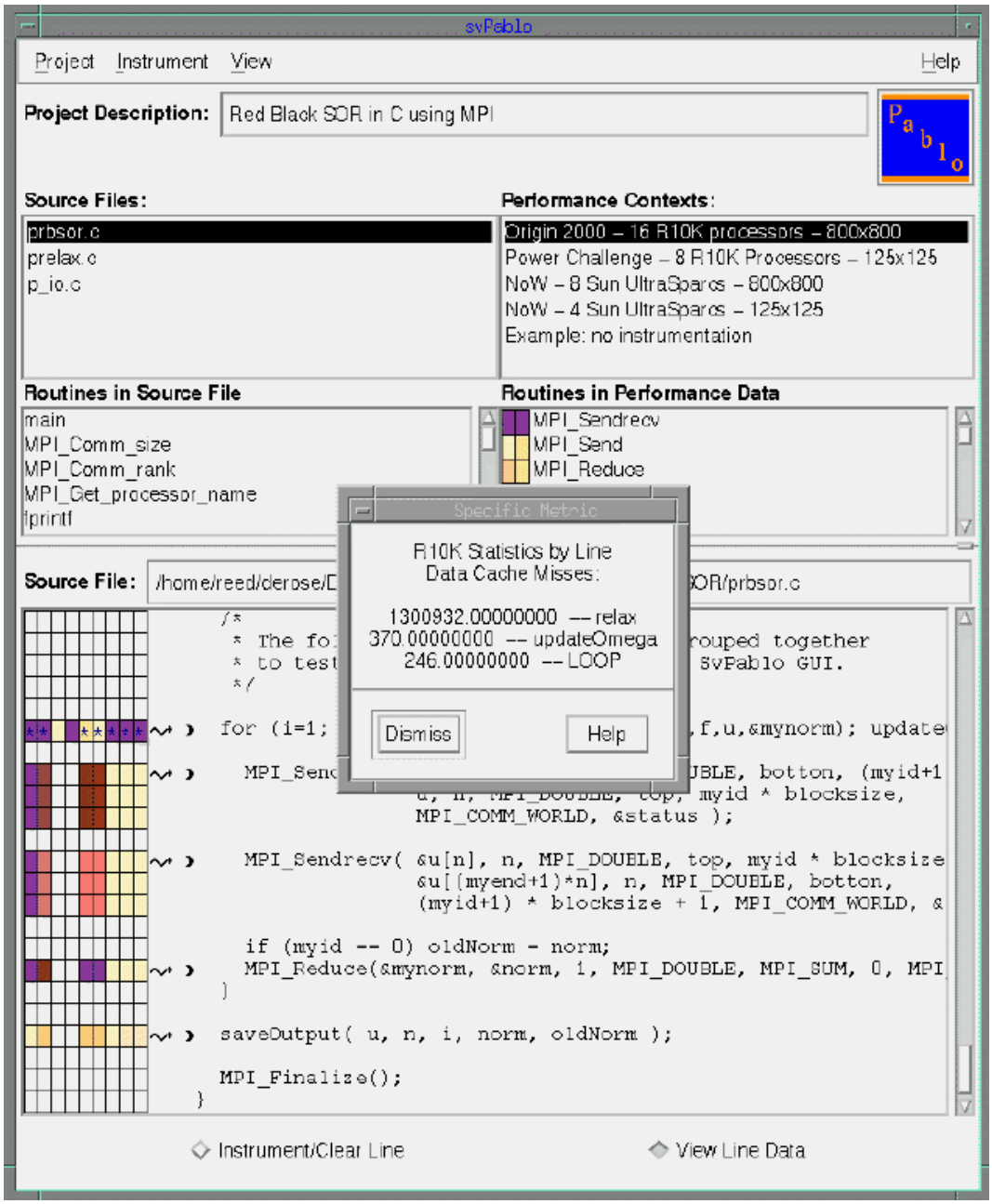

Figure 2.4 *Visualisation de performances avec SvPablo*

En choisissant une fonction, l'utilisateur peut visualiser le code source de la fonction, avec les carrés colorés qui représentent les indices de performance enregistrées pour chaque ligne du code source. Les valeurs correspondants à chaque carré peuvent être obtenues en sélectionnant le carré, ainsi que les valeurs individuelles par

processeur et quelques données statistiques (moyenne, déviation, maximum, minimum).

Comme il n'y a pas d'enregistrement de traces d'événements temporels, l'outil ne possède pas de visualisations temporelles, comportant un suivi détaillé de la vie du programme.

2.4.4 Paradyn

L'environnement Paradyn [50] a été conçu pour identifier les erreurs de performance automatiquement, pendant l'exécution d'un programme parallèle. Paradyn est constitué par des modules de collecte de données, connectés au programme parallèle à observer, et par un module principal, qui contrôle les modules de collecte et présente l'interface graphique à l'utilisateur.

Les modules de collecte d'indices de performance peuvent se connecter dynamiquement à un programme en train d'être exécuté. Ils peuvent ajouter et enlever des primitives d'instrumentation de points du programme (début et fin de procédures, appel de primitives de communication, etc.), et envoyer les données collectées au module principal. Les données collectées par Paradyn ne sont pas de traces, mais des échantillons des valeurs de compteurs et de chronomètres. Ces indices sont rassemblés à intervalles réguliers par le module principal de Paradyn, qui les utilise pour chercher des problèmes de performance dans le programme.

Paradyn recherche les problèmes de performance dans un espace appelé W^3 (pour *Why, Where, When*). D'abord, il teste des hypothèses dans l'axe *Why*, pour chercher l'existence de goulots d'étranglement, comme excès de temps de blocage en synchronisations, excès de primitives de synchronisations, taux d'utilisation des processeurs insuffisants, etc. Ces tests sont réalisés avec des primitives simples, et ne nécessitent que d'une petite quantité de données. Si un problème est détecté, Paradyn augmente le niveau d'instrumentation du programme, pour tenter de trouver la localisation du problème (le processeur, la procédure, l'objet de synchronisation, etc. — axe *Where*). L'axe *When* situe le problème dans le temps.

Les données capturées peuvent aussi être visualisées dans des histogrammes où dans une matrice d'indicateurs. L'environnement offre un mécanisme pour simplifier l'accès aux données, pour aider le développement d'autres visualisations.

Paradyn a été conçu pour supporter des grands programmes, qui s'exécutent sur un grand nombre de processeurs pendant longtemps. Le volume de données capturé est assez petit, et il est adapté dynamiquement, pour n'augmenter la quantité de données qu'à des endroits où un problème de performance est détecté. Dû à la façon dont il recherche les problèmes de performance, il ne peut détecter que des problèmes qui durent suffisamment longtemps. Comme les indices de performances

générés ne comportent pas d'événements, il ne peut pas reconstruire la succession d'états du programme observé.

2.4.5 Annai

Annai [13, 71] est un environnement intégré pour le développement et la mise au point des programmes parallèles. Il comporte, entre autres, un débogueur symbolique étendu et un outil d'analyse et visualisation de traces d'exécution. Les extensions du débogueur permettent la représentation graphique de structures de données réparties, la détection d'inter-blocages et la réexécution déterministe.

La bibliothèque de traçage d'Annai peut être configurée pour générer des traces détaillées de l'exécution ou des profils plus succincts de parties du programme, telles que procédures ou boucles. Un profil contient le temps et nombre d'appels de la partie du programme, le nombre et volume de ses communications et sa consommation mémoire. Les profils sont enregistrés à la fin de l'exécution du programme, ou peuvent être demandés à la volée, pendant l'exécution.

Les visualisations proposées par Annai sont montrées dans la figure 2.5. Il pos-

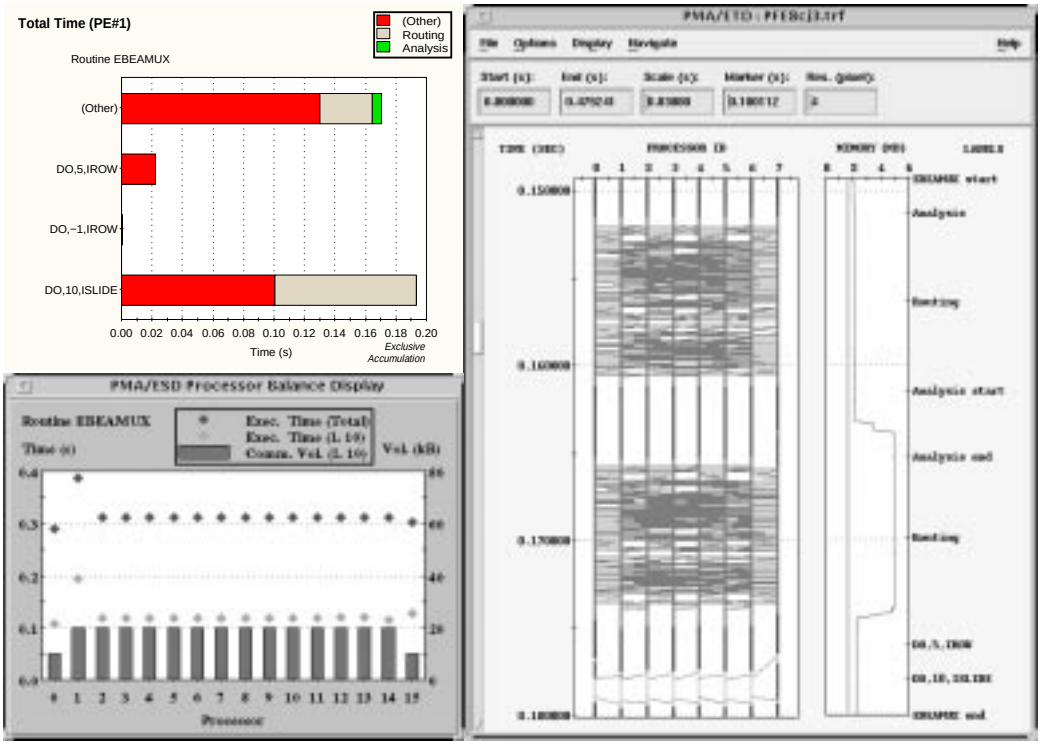

Figure 2.5 *Visualisations d'Annai*

sède deux types de graphes où l'espace est divisé en colonnes pour afficher des statistiques sur les profils d'exécution (à gauche dans la figure) : un pour montrer la distribution d'un indice de performance parmi les parties du programme, sur un processeur ou sur l'ensemble des processeurs, et l'autre qui montre la distribution de charge de calcul et communication, d'une partie ou de tout le programme, entre les processeurs. Les données des profils d'exécution peuvent aussi être utilisées par un outil qui montre la structure du programme parallèle, qui les affichera à côté du code source, d'une façon semblable à SvPablo.

Une troisième visualisation d'Annai est un diagramme espace-temps qui peut montrer l'évolution dans le temps des états de divers processeurs, des échanges de messages, et de la consommation mémoire. Le diagramme affiche aussi les instants où les parties du programme commencent ou terminent. Ce diagramme permet le déplacement en avant et en arrière dans le temps, ainsi que le changement de l'échelle de temps.

2.4.6 Scope

Scope [1] est le premier outil de visualisation développé dans le cadre du projet APACHE. Les objectifs étaient de réaliser un outil extensible, générique, personnalisable, interactif, puissant et convivial. Pour être extensible, l'outil est construit comme un graphe de composants fonctionnant en mode flot de données pur. La généricité de Scope est la capacité de certains composants comme un composant de visualisation de représenter plusieurs architectures telles que l'hypercube, le réseau maillé, etc. L'utilisateur peut personnaliser la visualisation en choisissant les couleurs et les formes des objets visualisés. L'interactivité est la possibilité d'inspecter certains objets visualisés, essentiellement les processeurs et les canaux de communication.

Les visualisations de Scope sont animées : elles sont rafraîchies chaque fois qu'un événement est lu à partir du fichier de traces. Il ne possède pas de visualisations historiques : tous les objets représentés par Scope à un instant donné de la visualisation correspondent à l'état des composants du programme parallèle observé à un instant donné de son exécution. Scope comporte une visualisation tridimensionnelle unique, permettant de représenter l'état des processeurs et des liens de communication. Scope offre une possibilité de retour en arrière à des états sauvegardés précédemment, la sauvegarde se faisant à l'initiative de l'utilisateur ou automatiquement. En outre, l'interface de Scope devrait permettre le retour arrière pas à pas, par relecture de la trace à l'envers : ce mode est cependant incompatible avec la technique de simulation et n'a donc pas été implémenté.

L'architecture interne de Scope est fortement basée sur Eve [2], un environne-

ment de programmation visuelle destiné à la construction graphique de programmes sous forme de graphe de composants de manipulation de données. Cette architecture donne à Scope ses capacités d'extensibilité via le développement de composants qu'on peut ajouter assez facilement à son graphe de composants. La technique de simulation n'étant pas générique, deux lecteurs de trace et deux simulateurs ont été développés pour visualiser l'exécution de programmes C-parallèle — modèle de programmation basé sur Occam/CSP [36] —, ainsi que de programmes PVM [69]. Scope a été testé sur quelques applications jouets, essentiellement écrites en utilisant le premier de ces deux modèles de programmation. Le retour en arrière à un état enregistré au préalable ne fonctionne pas « pour des raisons techniques ». Même s'il permet de se déplacer en arrière dans le temps, il est difficile de suivre l'évolution de l'exécution d'un programme comportant quelques dizaines d'objets tracés juste en observant la succession de leurs états instantanés.

L'analyse de Scope a fait apparaître un certain nombre de limitations qui ne permettaient pas d'envisager son utilisation pour le développement de Pajé sans une restructuration complète. Deux des objectifs de Pajé étaient en effet de combiner un haut niveau d'interactivité — permettant à la fois d'obtenir des informations additionnelles concernant certains des objets représentés et de changer dynamiquement des paramètres d'affichage tels que l'intervalle de temps visualisé ou le niveau d'abstraction des données visualisées — avec une visualisation historique — représentant l'évolution, pendant un intervalle de temps donné, des états du programme observé. L'utilisation par Scope d'un graphe flot de données *pur* n'est pas adapté, pour des raisons qui seront abordées dans la section 3.4.1.3, au développement de telles visualisations.

2.4.7 Gthread

Gthread [76], est un des seuls outils, qui permet la visualisation de programmes parallèles contenant plusieurs fils d'exécution. Les visualisations que Gthread propose sont représentées dans la figure 2.6. Elles présentent l'état instantané de chaque fil d'exécution, le graphe des appels de fonctions, les verrous d'exclusion mutuelle, les barrières de synchronisation et un graphe avec le déroulement de l'exécution dans le temps. Toutes les visualisations sont animées : un appel de fonction, par exemple, est représenté par une animation du déplacement du cercle qui représente le fil d'exécution qui a réalisé l'appel de la fonction appelante à la fonction appelée, dans le graphe d'appels.

La seule liaison avec le code source est l'identification de la fonction exécutée par chaque fil d'exécution. L'outil est limité à des petits programmes, ne comportant que quelques dizaines de fils d'exécution maximum, s'exécutant sur une seule machine. Gthread ne gère donc pas la notion de nœuds ni de communication par

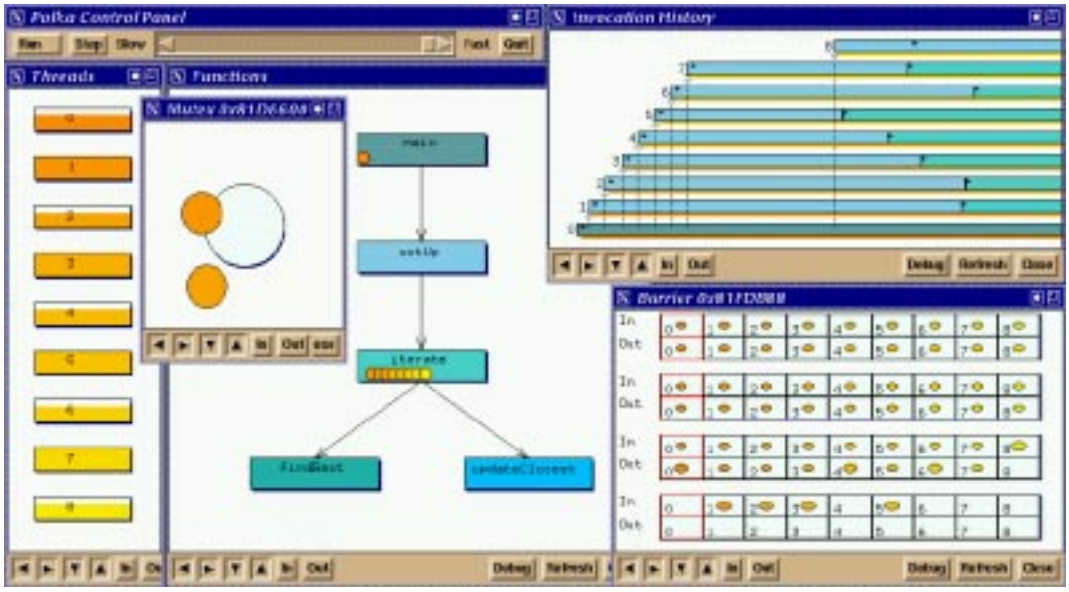

Figure 2.6 *Les visualisations de Gthread*

échanges de messages. Les visualisations sont animées, mais elles ne sont pas interactives : la seule interaction possible est le contrôle de la vitesse de simulation.

2.5 Conclusion

À partir de données collectées durant l'exécution de programmes parallèles, les environnements de visualisation calculent les indices de performances susceptibles d'aider les utilisateurs à identifier leurs « erreurs de performances » avant de les présenter de façon aussi intelligible que possible. Nous avons identifié trois propriétés dont il est souhaitable qu'un environnement de visualisation dispose : l'extensibilité, l'interactivité et la « scalabilité ». Aucun outil existant ne propose l'ensemble de ces propriétés, ce qui a motivé le travail présenté dans cette thèse. L'architecture de Pajé, qui fait l'objet du chapitre suivant, a été très influencée par la volonté de les combiner au sein d'un même environnement.

3

Présentation de Pajé

Ce chapitre a pour objectif de décrire le contexte de la thèse, à savoir le projet APACHE et l'environnement logiciel ATHAPASCAN et de donner une vision d'ensemble de Pajé, tant du point de vue de ses fonctionnalités et de son utilisation que de son architecture, avant de décrire plus en détail les aspects les plus originaux de la thèse dans les chapitres suivants.

3.1 Le projet APACHE et l'environnement ATHAPASCAN-0

3.1.1 Présentation et objectifs du projet APACHE

Le projet APACHE[1], dans le cadre duquel se développe cette thèse, propose une approche originale de la programmation des machines parallèles pour le calcul haute performance qui permette d'atteindre un bon compromis performance-portabilité. La démarche suivie pour la mise en œuvre de cette approche consiste à construire un environnement de programmation, appelé ATHAPASCAN[2].

L'environnement ATHAPASCAN est constitué d'une interface de programmation, d'un noyau exécutif et d'un environnement de prise de traces et visualisation. L'interface de programmation, appelée ATHAPASCAN-1 privilégie un modèle de pa-

[1] APACHE est un acronyme pour « Algorithmique Parallèle et Partage de Charge »
[2] ATHAPASCAN est le nom de la langue parlée par les Apaches

rallélisme de tâches asynchrones assorti de règles de synchronisation pour l'accès aux données partagées. Elle permet le calcul dynamique d'une représentation abstraite du programme (graphe *macro-dataflow*) et une répartition automatique (en utilisant ce graphe) de la charge de calcul et des données. Le noyau exécutif, appelé ATHAPASCAN-0, implante un réseau dynamique de fils d'exécution communicants qui permet l'exploitation du parallélisme offert par la machine parallèle utilisée, tout en restant portable. Enfin, l'environnement de prise de traces et visualisation permet l'observation, l'évaluation et la visualisation d'ATHAPASCAN et de ses applications. Cet environnement est constitué de deux parties, un dispositif logiciel de prise de traces, qui permet l'enregistrement des événements permettant de reconstituer le comportement du programme parallèle et d'un programme qui permet la visualisation du comportement du programme à partir de ces traces. Ce dernier programme, nommé Pajé est l'objet de ce manuscrit.

3.1.2 L'environnement de programmation ATHAPASCAN

ATHAPASCAN est un environnement de programmation parallèle conçu pour être à la fois portable et efficace, et permettre de façon aisée le développement de programmes parallèles [55]. Pour atteindre ces objectifs l'environnement Athapascan est organisé en deux niveaux : ATHAPASCAN-1 et ATHAPASCAN-0 (voir figure 3.1). Les primitives d'un niveau sont bâties en utilisant l'interface offerte par le niveau immédiatement inférieur. Cette organisation permet aussi le développement d'applications parallèles sur chacun des niveaux séparément.

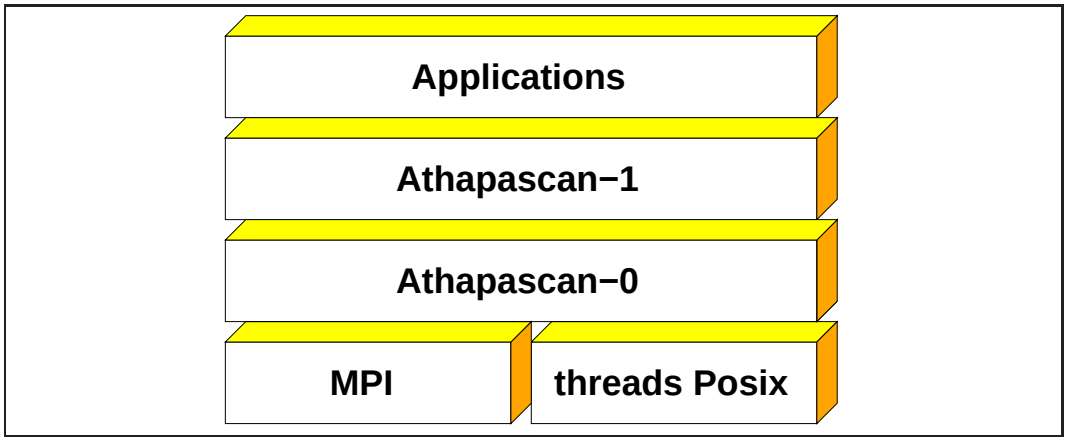

Figure 3.1 *L'architecture multi-niveau de l'environnement* ATHAPASCAN

Le niveau supérieur, ATHAPASCAN-1 [24], est une librairie C++ basée sur le paradigme de programmation parallèle explicite par tâches concurrentes similaires à celles fournies par des langages comme Cilk [5] et Jade [42]. ATHAPASCAN-1

abstrait des détails liés à la machine cible, comme de communications et les synchronisations. De plus, ATHAPASCAN-1 permet l'emploi de différentes stratégies d'ordonnancement pour mieux répartir la régulation de charge d'une application à une architecture cible spécifique.

Une application ATHAPASCAN-1 est vue comme un graphe de tâches, créé dynamiquement pendant l'exécution du programme parallèle. Dans l'abstraction ATHAPASCAN-1 il n'y a aucune communication, tous les échanges de données sont faits de manière implicite à partir d'une relation de dépendance de données décrite au moment de création de chaque tâche.

La couche inférieure ATHAPASCAN-0 [28, 6] propose des mécanismes de création de processus légers localement et à distance accompagnés de fonctions élémentaires de communication entre eux. Le modèle de programmation offerte par ATHAPASCAN-0 est celui d'*un réseau dynamique de processus légers communicants*. Ce niveau a été initialement développé pour la programmation directe d'applications scientifiques et comme support à l'exécutif d'ATHAPASCAN-1. Néanmoins il peut être utilisé en tant que support exécutif pour d'autres langages de programmation et/ou des bibliothèques parallèles (e.g. GIVARO[25]).

Un des aspects le plus important considéré pendant le développement d'ATHAPASCAN est celui de la portabilité. Ce soucis a conduit à la construction d'ATHAPASCAN-0 au dessus des standards de bibliothèques de processus légers (POSIX threads) et de communication (MPI).

3.1.3 L'interface de programmation d'ATHAPASCAN-0

Les programmes ATHAPASCAN-0 sont exécutés par un ensemble de fils d'exécution créés et détruit dynamiquement. Un fil d'exécution peut être créé localement ou dans un nœud distant, et une fois créé, il ne peut pas être migré sur un autre nœud. Les fils d'exécution d'un même nœud se communiquent par mémoire partagée et primitives de synchronisation. Des fils d'exécution appartenant à des nœuds distincts communiquent par échange de messages à travers des ports de communication (figure 3.2).

L'interface de programmation d'ATHAPASCAN-0 comporte un grand nombre de fonctions :

Initialisation : déclaration de ports globaux de communication et de services. Chaque requête de création de fil d'exécution à distance inclut le nom d'un service comme paramètre, le service étant la première fonction exécutée par le nouveau fil d'exécution lors de sa création.

Création d'un fil d'exécution : création avec ou sans attente de fils d'exécution locaux ou distants pour exécuter des services déclarés au préalable.

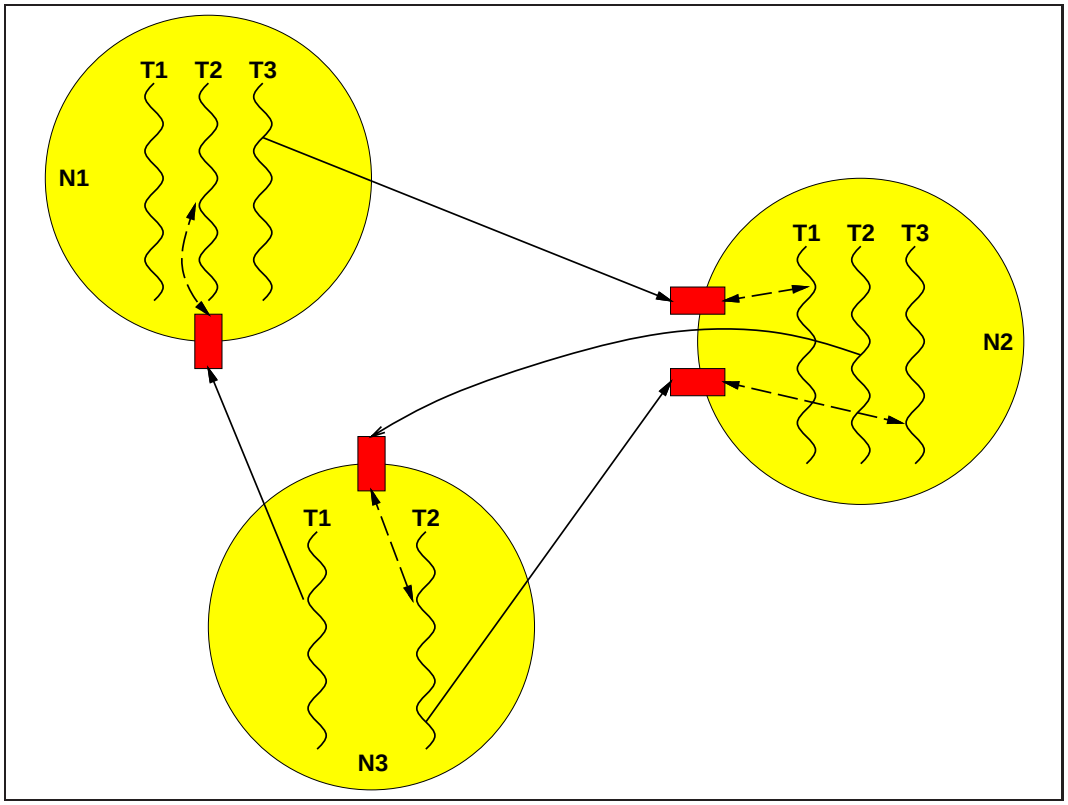

Figure 3.2 *Échange de messages utilisant des ports en* ATHAPASCAN-*0*

Synchronisation : fonctions classiques de synchronisation utilisant des sémaphores, des verrous d'exclusion mutuelle ou des variables de conditions. Il est aussi possible de se synchroniser sur l'événement de terminaison d'un fil d'exécution.

Communications : échange bloquant ou non bloquant de messages à travers des ports de communication.

Primitives de test : Étant donné que beaucoup de requêtes d'ATHAPASCAN-0 ne sont pas bloquantes, il est possible de tester la terminaison des requêtes. Les tests peuvent, eux aussi, être bloquants ou non bloquants.

3.1.4 Traces et performance

L'environnement ATHAPASCAN comporte un traceur logiciel, capable d'enregistrer les événements d'une exécution d'un programme ATHAPASCAN-0 à un niveau de détail suffisant pour la reconstitution, *post mortem*, des états des divers composants du programme ATHAPASCAN-0. Un traceur logiciel, dans notre con-

texte de processus légers communicants doit être à même d'identifier tous les objets clés (et les événements qui leur sont liés) d'un programme parallèles (les processeurs, les fils d'exécution, les ports de communications, les messages, les variables de synchronisation, etc.).

La prise de trace concerne essentiellement l'activité des fils d'exécution, avec pour chaque processus léger son identité, le début, la fin et la cause des périodes de suspension de son exécution. S'y ajoutent des informations permettant de reconstituer l'historique des communications. Il faut résoudre divers problèmes d'identification des fils d'exécution, d'observabilité de leur ordonnancement, d'atomicité des événements et de gestion des tampons de trace. Le traceur doit gérer l'absence de référence temporelle globale [47].

Le traceur n'enregistre que les événements au niveau de l'interface de programmation ATHAPASCAN-0. Cette décision a permis de développer un traceur ATHAPASCAN-0 d'une façon assez indépendante de la machine cible et donc portable sur différentes machines. Le prix à payer est une trace — et par conséquent une visualisation — qui ne contient pas des détails de l'exécution du programme parallèle à un niveau inférieur à l'interface de programmation, notamment l'action de l'ordonnanceur des fils d'exécution au niveau système d'exploitation. L'analyse des traces et la représentation des exécutions tracées par l'outil de visualisation Pajé font l'objet du travail décrit dans cette thèse.

3.2 Principales fonctionnalités de Pajé

Pajé est un environnement de visualisation interactif, « scalable » et extensible, conçu pour la visualisation « post mortem » de traces d'exécution.

L'interactivité est à la base de l'interface utilisateur de Pajé : la visualisation de l'exécution tracé ne défile pas, à vitesse plus ou moins rapide, sur l'écran de l'utilisateur. Au contraire, la visualisation se déplace dans le temps, en avant mais aussi en arrière, en fonction des mouvements de souris de l'utilisateur. Par ailleurs, il est possible d'inspecter les entités affichées, comme par exemple les communications dont on peut connaître l'émetteur, le destinataire ainsi que la taille du message communiqué. Un autre exemple d'inspection est la possibilité d'afficher le commentaire associé par le programmeur à un événement « utilisateur ». Une possibilité d'inspection réduite est attachée à tous les mouvements de la souris : les entités désignées par la souris sont mises en relief, par surbrillance par exemple, tandis qu'une information textuelle relative à l'entité est donnée ; de même tous les autres objets représentés et ayant un lien avec l'objet désigné sont mis en évidence : par exemple la désignation d'un sémaphore aura pour conséquence de mettre en évidence tous les états bloqués des fils d'exécution en attente sur ce sémaphore. Il est également

possible de faire la liaison entre la visualisation et le programme source correspondant.

La « scalabilité » a été une des motivations essentielles pour le développement de Pajé, puisqu'aucun environnement existant ne permettait la visualisation d'un modèle de programmation à base de processus légers communiquants créés dynamiquement et dont le nombre, potentiellement élevé, varie sans cesse. Pour représenter ce type de modèle de programmation, en tenant compte des limites matérielles des capacités des écrans ainsi que des limites dans la perception visuelle des utilisateurs, Pajé comporte des capacités sophistiquées de filtrage. Le filtrage permet de regrouper ou d'éclater dynamiquement des données, en fonction des commandes de l'utilisateur. L'utilisation du filtrage permet donc de simuler des possibilités de zoom : il est par exemple possible de représenter d'abord une exécution parallèle à un haut niveau d'abstraction où on ne voit plus les fils d'exécution mais seulement les nœuds du système parallèle ; si besoin est, il est alors possible de concentrer la visualisation sur un ou plusieurs nœuds, en éclatant la représentation synthétique en une représentation détaillée.

L'extensibilité a été une contrainte forte dans la conception et la réalisation de Pajé. L'objectif est de ne pas figer l'outil pour un modèle de programmation donné ou un ensemble de visualisations données. Afin d'être plus facilement extensible, Pajé a été conçu comme un ensemble de composants interconnectable selon un graphe d'analyse donné. Un soin particulier a été apporté à faciliter le développement de nouveaux composants. Par ailleurs le développement de composants génériques permet l'utilisation de Pajé pour des visualisations pour lesquelles il n'était pas prévu au départ, sans avoir à modifier l'environnement.

Pajé offre plusieurs visualisations : soit une visualisation détaillée de tout ce qui est susceptible de varier dans le temps, combinant diagramme espace-temps et diagramme de Gantt avec une représentation de l'état des objets de synchronisations (verrous, variables conditionnelles et sémaphores), soit des visualisations synthétiques. Pour ces dernières il est possible à l'utilisateur de choisir les données à visualiser : occupation des nœuds de calcul, un comptage des messages, le trafic de données, etc. ainsi que la forme de cette visualisation : graphe de colonnes, camembert, etc.

3.3 Mise au point d'une application en utilisant Pajé

Pour présenter l'utilisation de Pajé et ses fonctionnalités, un exemple d'utilisation pour une application de grande taille en dynamique moléculaire est expliqué

en détail. Pour cette application, la programmation en ATHAPASCAN-0 a facilité la décomposition en plusieurs fils d'exécution des calculs affectés à chaque nœud, laquelle a permis de recouvrir les communications par du calcul d'une façon très naturelle. L'utilisation de Pajé pour la mise au point de cette application s'est montré très utile pour améliorer l´équilibrage de charge et permettre le recouvrement des temps de communication par du calcul.

3.3.1 Visualisation de fils d'exécution dans Pajé

La visualisation de l'activité des fils d'exécution dans les nœuds se fait à l'aide d'un diagramme espace-temps. Ce diagramme (voir figure 3.3) combine, dans une seule représentation, les états de chaque fil d'exécution et les communications. L'axe horizontal représente le temps. Les fils d'exécution sont groupés suivant l'axe

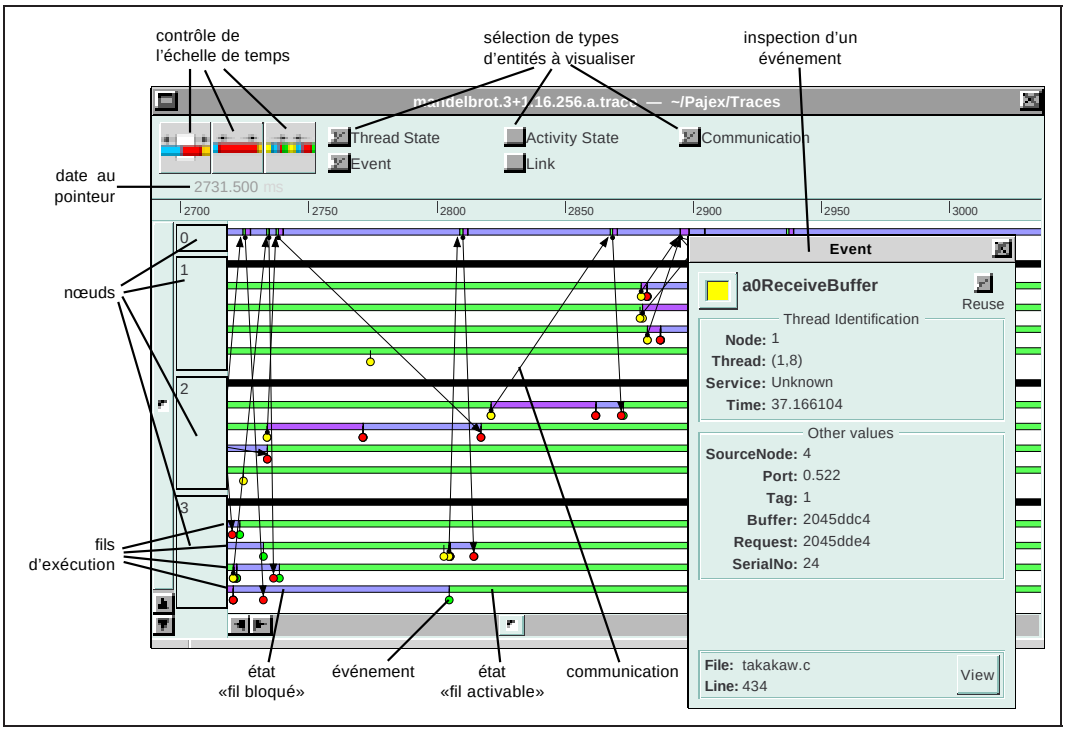

Figure 3.3 *Exemple d'un diagramme espace-temps d'un programme* ATHAPASCAN-*0 Chaque trait horizontal représente le déroulement des états d'un fil d'exécution du programme. Les traits plus foncés indiquent les intervalles de temps où le fil d'exécution est bloqué (en attente de communication). Les traits plus clairs correspondent aux intervalles où le fil d'exécution est « activable ». La fenêtre à droite montre l'inspection d'un événement de réception.*

vertical par nœud. L'espace réservé à chaque nœud du système parallèle est dynamiquement ajusté en raison du nombre de fils d'exécution existants sur ce nœud.

Les communications sont représentées par des flèches, tandis que les états des fils d'exécution sont représentés par des rectangles. Le diagramme utilise des couleurs pour indiquer le type de communication ou l'activité d'un fil d'exécution. Cette représentation n'est pas la plus compacte ni la plus « scalable », mais elle s'avère très pratique pour analyser les détails des interactions entre fils d'exécution, la répartition de charge et le recouvrement des latences de communication. Pajé traite le problème de la « scalabilité » des visualisations à l'aide de filtres, présentés dans la section 3.3.8 et détaillés dans le chapitre 6.

L'utilisateur peut se déplacer en avant ou en arrière dans le temps. Ce déplacement est « rapide » dans les limites temporelles des données gérées en mémoire par Pajé et qui définissent ce que nous appellerons ultérieurement « fenêtre d'observation ». Une tentative de déplacement au delà de la limite de la fenêtre d'observation provoque la lecture de nouveaux événements dans la trace et leur analyse par Pajé, ainsi qu'un glissement des limites de la fenêtre d'observation. De façon analogue, un déplacement en deça du début de la fenêtre d'observation courante provoque la restauration d'un état enregistré précédemment est restauré et une réexécution de l'analyse jusqu'à la date sélectionnée. Ce mécanisme sera mieux décrit dans la section 3.4.2.5 et dans le chapitre 5.

3.3.2 Application en dynamique moléculaire

Cette application simule le mouvement d'atomes de protéines [4]. L'algorithme général consiste à calculer la suite, au cours du temps, des positions des atomes d'un système, étant données leurs positions et vitesses initiales. Cette application est capable de traiter des très grandes structures moléculaires. Elle a permis de simuler le mouvement de la plus grande structure de protéine se trouvant dans la base de données « Brookhaven Protein Data Bank » ; il s'agit d'une β-galactosidase [38]. Cette protéine d'environs 65 240 atomes est immergée dans une sphère d'eau, de rayon égal à 100 Å ; ce procédé donne lieu à un système de 430 000 atomes. La dimension de ce système est environ quatre fois plus grande que les dimensions ayant été traitées par d'autres codes de dynamique moléculaire.

Le calcul des forces entre chaque paire d'atomes est le principal goulot d'étranglement durant une itération de dynamique moléculaire. Afin de diminuer le volume de calculs, on ne considère que les interactions entre chaque atome et ses voisins, i.e., les atomes inclus dans une sphère de coupure de rayon donné. Cette approximation rend le problème irrégulier du point de vue du parallélisme : la sélection des paires d'atomes pour lesquels il est nécessaire de calculer les forces dépend des positions des atomes dans le système. Cependant, le problème présente une bonne localité de données.

3.3.3 Parallélisation de l'application

Une technique traditionnelle pour paralléliser la simulation d'un grand nombre d'atomes consiste à affecter une partie de l'espace simulé sur chaque nœud. Chaque nœud s'occupe des mouvements des atomes appartenant à cette partie de l'espace. Pour calculer les forces exercées sur ces atomes, les nœuds traitant des sous-espaces voisins échangent les positions des atomes proches des frontières des domaines qui leur sont affectés.

Chaque nœud calcule les forces qui s'exercent entre les atomes du domaine qui lui est affecté. Il calcule également, en collaboration avec les nœuds responsables des domaines voisins, les forces qui s'exercent entre les atomes de son domaine et les atomes de la frontière entre son domaine avec les domaines voisins. Les nœuds s'échangent alors les forces qui s'exercent entre les atomes situés à la frontière de leurs domaines.

L'utilisation de fils d'exécution permet l'exploitation du parallélisme de grain fin ainsi que le recouvrement automatique du surcoût de communication par des calculs. Dans l'application traitée, un fil d'exécution est affecté au calcul des forces s'exerçant entre les atomes d'un domaine. Il n'utilise que les données locales au nœud. Seules les synchronisations entre les fils d'exécution d'un nœud doivent être décrites. Ces synchronisations dépendent des accès aux données partagées. Durant l'exécution, l'ordonnanceur du système recouvrira automatiquement le surcoût dû aux communications en activant des fils d'exécution de calcul. Le paragraphe suivant décrit plus précisément le rôle de chaque fil d'exécution durant une itération de l'application de dynamique moléculaire.

3.3.4 Solution parallèle utilisant des fils d'exécution

Sur un nœud p, la simulation est effectuée par les fils d'exécution suivants :

Fil d'exécution principal. Ce fil d'exécution gère les accès aux données partagées avec les autres fils d'exécution locaux. Plus précisément, il gère les synchronisations des fils d'exécution entre les différentes phases de lecture/écriture des données relatives aux atomes du domaine local. Il calcule également une partie des forces d'interaction relatives à la géométrie du système. Finalement, il intègre les équations de mouvement des atomes du nœud. Sur les visualisations des figures 3.4(a) et 3.4(b), il est le fil d'exécution le plus haut de chaque nœud.

Fils d'exécution de communication. Sur chaque nœud, il y a un fil d'exécution de ce type pour la communication avec chacun des nœuds voisins (les nœuds voisins étant ceux qui gèrent les domaines adjacents au domaine géré localement). Ces fils d'exécution envoient les coordonnées des atomes locaux

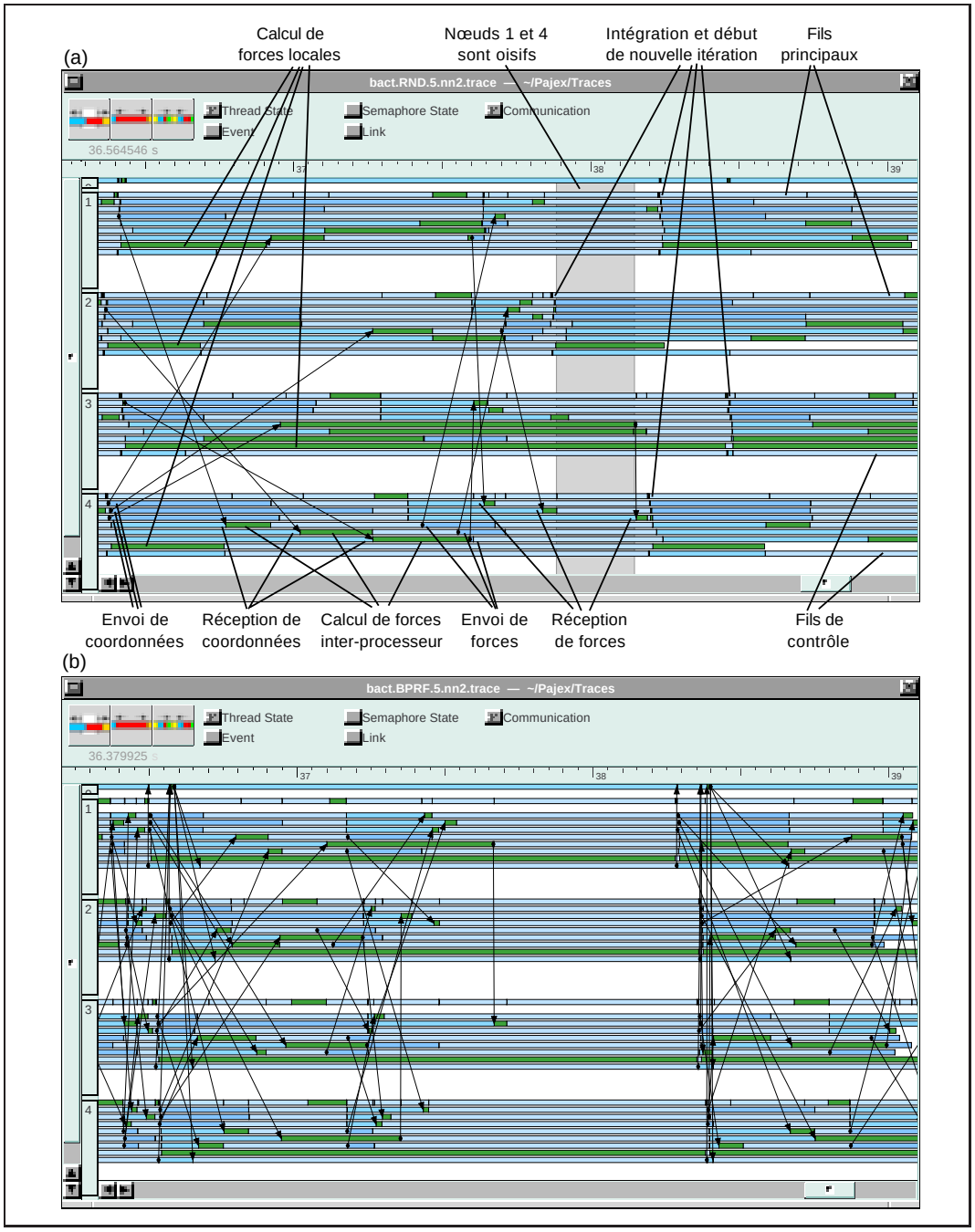

Figure 3.4 *Visulalisation d'une itération du programme de dynamique moléculaire (a) en utilisant un placement aléatoire des domaines—les nœuds 1 et 4 attendent longuement les forces calculées par le nœud 3 (seules les communications impliquant le nœud 4 sont montrées) ; (b) avec un meilleur placement des domaines, tous les nœuds restent actifs en permanence.*

et reçoivent les forces calculées sur les nœuds voisins correspondants. Les second, troisième et quatrième fils d'exécution (en partant du haut) de chacun des nœuds des visualisations des figures 3.4(a) et 3.4(b) sont de ce type.

Fils d'exécution de calcul des forces inter-nœuds. Chaque nœud possède également un exemplaire de ce type de fil d'exécution pour chacun de ses voisins. Ces fils d'exécution ont pour rôle de recevoir les coordonnées des atomes, de calculer les forces entre atomes de deux nœuds différents et de renvoyer les forces calculées. Les cinquième, sixième et septième fils d'exécution (en partant du haut) de chaque nœud des visualisations des figures 3.4(a) et 3.4(b) sont de ce type.

Fil d'exécution de calcul des forces locales. Son rôle est de calculer les forces qui s'exercent entre les atomes du nœud local. C'est le fil d'exécution situé en deuxième position en partant du bas pour chacun des nœuds représenté sur les figures 3.4(a) et 3.4(b).

Fil d'exécution de contrôle : il transmet l'information de contrôle de la simulation entre le nœud local et le nœud principal. Il s'agit du fil d'exécution le plus bas de chaque nœud sur les figures 3.4(a) et 3.4(b).

3.3.5 Visualisation de l'application

Une visualisation détaillée des fils d'exécution d'une application est d'une valeur inestimable pour le programmeur. Une telle visualisation lui permet de :

- Vérifier la cohérence des synchronisations entre les fils d'exécution partageant la mémoire dans un nœud.
- Vérifier la répartition de charge entre les nœuds, ce qui permet de développer d'autres heuristiques pour l'affectation des atomes aux nœuds.
- Choisir les priorités des fils d'exécution de façon à obtenir un bon recouvrement des communications par du calcul.

Les figures 3.4(a) et 3.4(b) représentent deux traces d'exécution d'une itération de dynamique moléculaire, obtenues en utilisant différents schémas de placement des charges de calcul sur les nœuds. Dans ces traces, le nœud 0 sert uniquement à contrôler la simulation. La figure 3.4(a) représente la trace d'une exécution où le placement initial des tâches est fait de façon aléatoire. La charge de calcul est deséquilibrée. Il est possible de vérifier, en analysant la portion de la trace mise en évidence, que les nœuds les moins chargés (nœuds 1 et 4) sont en attente du nœud le plus chargé (nœud 3).

La figure 3.4(b) représente une trace d'exécution utilisant une heuristique de placement initial qui équilibre la charge de calcul et minimise le volume des com-

munications. Dans cet exemple, il est possible de vérifier que, au cours d'une itération, les nœuds travaillent tout le temps ; ils finissent leur itérations tous en même temps. On observe aussi le recouvrement des fils d'exécution qui s'occupent des communications par les fils d'exécution qui calculent les forces locales.

3.3.6 Visualisation des synchronisations locales

ATHAPASCAN comporte des primitives qui permettent la synchronisation entre fils d'exécution d'un même nœud. Ces primitives sont les classiques sémaphores et verrous d'exclusion mutuelle. Dans le diagramme espace-temps de Pajé, les états des sémaphores et des verrous sont representés de façon analogues aux états des fils d'exécution : à chaque état possible est associée une couleur, et un rectangle de cette couleur est représenté à la position correspondant à la période de temps durant laquelle le sémaphore (ou le verrou) se trouvait dans cet état. Pajé reconnaît trois états différents pour un sémaphore : quand un fil d'exécution est bloqué dedans, quand un fil d'exécution se bloquera dedans s'il effectue une opération « P », et quand il est possible à un fil d'exécution d'effectuer une opération « P » sur ce sémaphore sans se bloquer. Un exemple de visualisation de quelques opérations sur les sémaphores est représenté dans la figure 3.5.

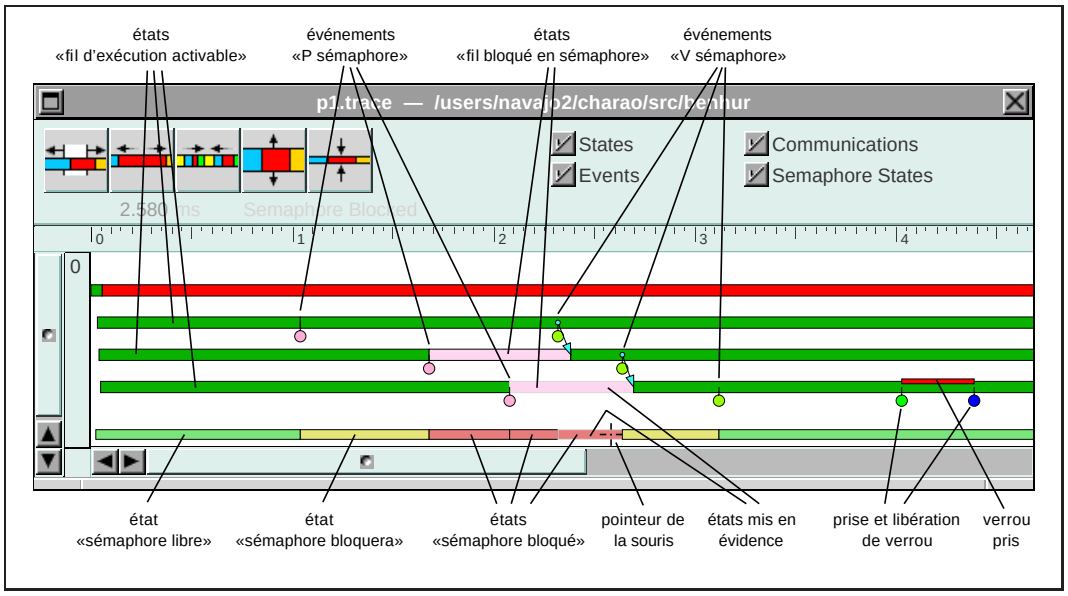

Figure 3.5 *Visualisation de sémaphores*
On peut remarquer la surbrillance d'un état bloqué de fil d'exécution en raison de la position du pointeur de souris sur un état bloqué d'un sémaphore, et les flèches qui indiquent le lien entre une opération « V » dans un sémaphore et le déblocage correspondant d'un fil d'exécution.

À la différence des sémaphores, les verrous ont la notion de propriétaire, c'est à dire que un seul fil d'exécution au plus peut posséder un verrou. Les autres fils d'exécution se bloquent s'ils essaient de le verrouiller, jusqu'à ce que le verrou soit libéré par le fil d'exécution qui en a la propriété. Cette propriété est utilisée pour représenter les verrous dans les visualisations : un rectangle est dessiné à proximité du fil d'exécution durant la période durant laquelle il possède un verrou (voir sur la partie droite de la figure 3.5). En outre, pour faciliter l'identification des fils d'exécution bloqués en attente d'un verrou, un autre rectangle d'une couleur différente est représenté à proximité de chacun de ces fils d'exécution.

3.3.7 Interactivité

Dans les outils de visualisation non interactifs, l'utilisateur ne peut que contrôler la vitesse de simulation. Pajé, en revanche, offre la possibilité de déplacement dans le temps. La simulation est entièrement contrôlée par le déplacement de l'utilisateur dans le temps. Les déplacements au tour de la période représentée à l'écran sont rapides, tandis que le déplacement à des instants distants peut durer plus longtemps (voir section 3.4.2.5). Par ailleurs, Pajé offre plusieurs possibilités d'interaction aux programmeurs : les données visualisées peuvent être inspectées pour en obtenir plus de détails ou des informations qui ne sont représentées graphiquement, identifier les relations entre objets, voir le code source correspondant aux objets visualisés et enfin changer le niveau d'abstraction des visualisations.

Toute l'information qui peut être déduite à partir d'un fichier de traces n'est pas directement représentable (ou il peut ne pas être désirable de la représenter intégralement) dans le diagramme espace-temps. Des informations supplémentaires peuvent être obtenues sur demande de l'utilisateur. L'intégralité des informations connues sur un objet représenté peut être montrée dans une fenêtre d'inspection (voir figure 3.3), obtenue en sélectionnant la représentation graphique de l'objet voulu.

Une autre information très utile est la liaison entre objets du diagramme. Par exemple, la couleur d'un état d'un fil d'exécution indique que ce fil d'exécution est bloqué en attente d'un sémaphore pendant un certain temps, mais, l'identification du sémaphore concerné n'est pas immédiate. La représentation de cette information de façon permanente sur l'écran risquerait de trop encombrer la visualisation. Dans Pajé, le simple fait de placer le pointeur de la souris sur la représentation graphique de l'état d'un fil d'exécution met en évidence, par surbrillance, l'état correspondant du sémaphore, de façon à permettre une reconnaissance immédiate (voir figure 3.5). De façon similaire, tous les états des fils d'exécution bloqués en attente d'un sémaphore sont mis en évidence lorsque le pointeur est positionné sur l'état correspondant du sémaphore. Le nom de l'objet placé sous le pointeur est égale-

ment affiché en haut de la fenêtre, à côté de la date correspondant à la position du pointeur. Plusieurs relations entre objets sont montrées de cette façon dans Pajé.

Pajé maintient optionnellement une relation entre les objets visuels et le code source du programme parallèle : à partir de la représentation visuelle d'un événement, il est possible d'afficher la ligne du code source qui a générée cet événement. De la même façon, la sélection d'une ligne du code source met en évidence tous les événements affichés qui ont été générés par l'exécution de cette ligne.

3.3.8 Filtrage d'information et capacités de « zooming »

La limitation de l'espace écran n'est pas la seule raison pour ne pas représenter toute l'information qui peut être déduite des traces d'exécution : une partie de l'information peut ne pas être nécessaire tout le temps ou peut ne pas être représentable d'une façon graphique ou peut avoir diverses représentations visuelles possibles. En donnant à l'utilisateur une vision simplifiée, abstraite des données et les moyens d'accéder à plus de détails de façon facile et intuitive, là où il lui semble nécessaire, semble être une bonne manière de l'aider à trouver les causes des problèmes de performance de son programme.

Pajé offre plusieurs fonctionnalités de filtrage et de « zooming » pour aider les programmeurs à faire face à cette grande quantité d'information. Le filtrage dans Pajé peut être de plusieurs types :

Réduction : fournit des représentations plus abstraites de l'information. La figure 3.6 montre la même exécution que la figure 3.4(b), comportant seulement une ligne par noeud, et dont les états représentent le nombre de fils d'exécution actifs à chaque instant. La figure montre également un graphe « camembert » de l'activité des nœuds, dans l'intervalle de temps sélectionné dans le diagramme espace-temps. Une autre possibilité de réduction est le regroupement de plusieurs nœuds, fils d'exécution, sémaphores, etc.

Sélection : permet l'enlèvement d'objets d'une visualisation. Cette sélection peut être basée sur le type d'un objet visuel (ne pas montrer les événements ou les états des fils d'exécution, ou les communications, etc.), sur son sous-type (de tous les états possibles des fils d'exécution, montrer seulement ceux qui représentent un fil d'exécution bloqué) ou sur une certaine instance spécifique (choisir quels nœuds, fils d'exécution, sémaphores, verrous d'exclusion mutuelle ou groupes seront visualisés).

Repositionnement : permet aux utilisateurs de choisir l'ordre suivant lequel les objets sont affichés, de sorte que, par exemple, des nœuds relatifs puissent être affichés côte à côte. Pajé dispose d'un filtre de repositionnement qui réutilise l'espace écran rendu disponible par la terminaison d'objets tels que les fils

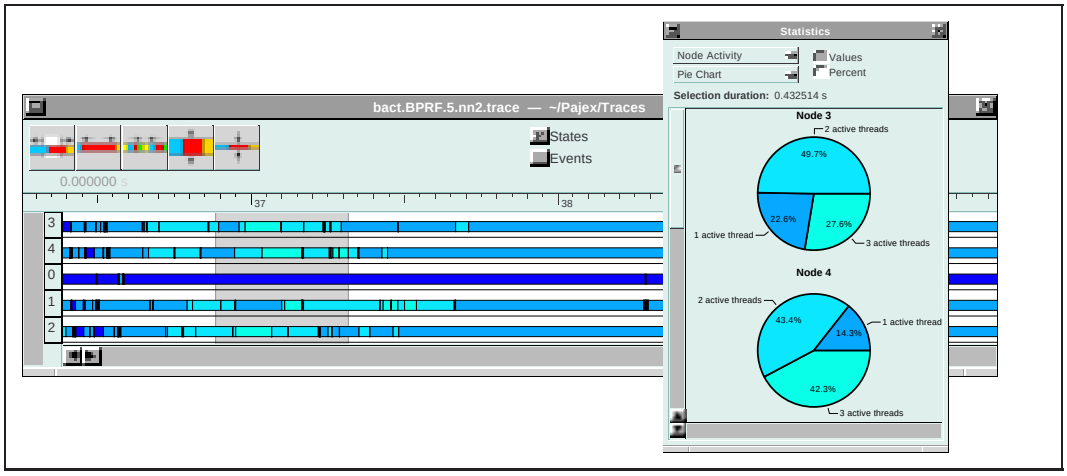

Figure 3.6 *Utilisation des processeurs*
Groupement des fils d'exécution de chaque nœud pour montrer l'évolution de l'état global du système (les couleurs plus claires correspondent à un plus grand nombre de fils d'exécution activables). Le camembert montre le nombre de fils d'exécution activables en pourcentage de l'intervalle de temps sélectionné.

d'exécution de courte durée (voir figures 3.7(a) et 3.7(b)).

Changements visuels : la correspondance entre le type d'une entité et la couleur, forme et taille de sa représentation graphique peut être personnalisé par l'utilisateur.

La possibilité de passer d'une visualisation abstraite des données à des visualisations plus détaillées donne à l'utilisateur les capacités de naviguer ses données entre différents niveaux d'abstraction.

3.4 Architecture de Pajé

Les principaux apports de Pajé concernent l'extensibilité, l'interactivité et la « scalabilité », notions qui sont définies dans le chapitre précédent. Ces trois aspects étant fortement corrélés, il est difficile de les présenter séparément. C'est pour cette raison que nous avons cru bon de regrouper ici les définitions des principaux concepts utilisés dans la suite de la thèse. Ces concepts concernent l'organisation interne de l'environnement — modulaire, structurée en graphe flot de données de composants logiciels indépendants —, l'organisation des données — hiérarchique — ainsi que l'accès aux données utilisées pour la visualisation — dont le stockage est centralisé.

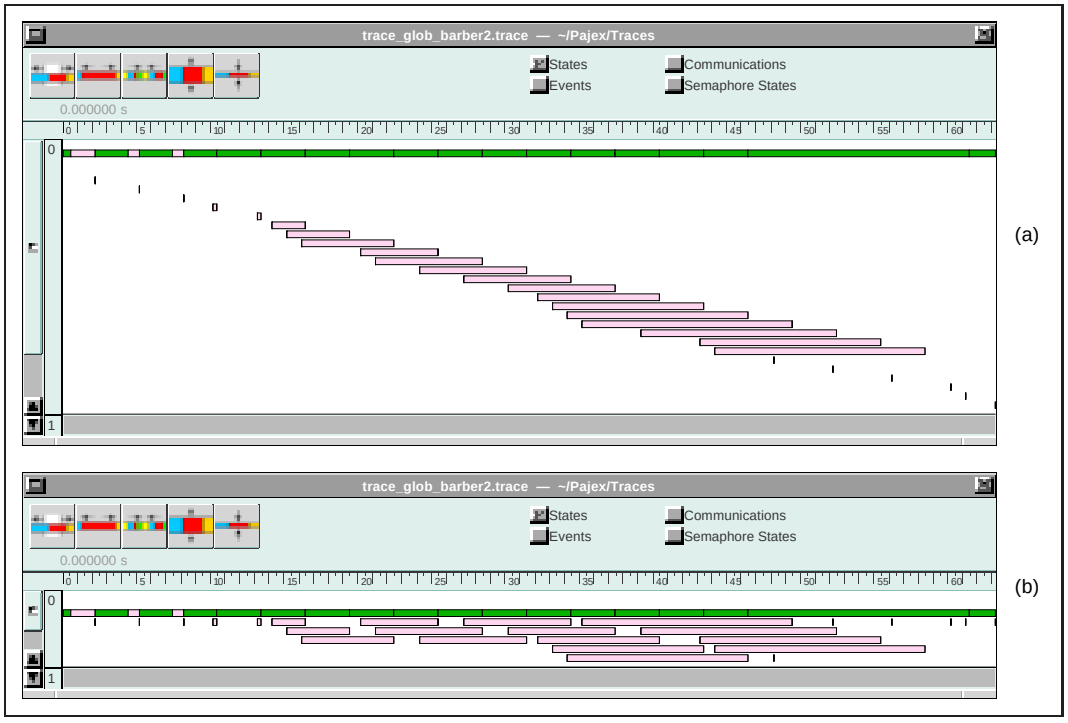

Figure 3.7 *Réutilisation de l'espace des fils d'exécution de courte durée*
(a) visualisation d'un programme avec des fils d'exécution de courte durée ;
(b) le même programme, réutilisant l'espace des fils d'exécution terminés.

3.4.1 Graphe de composants

3.4.1.1 Définition

Pour faciliter l'extensibilité de l'environnement, Pajé a été développé de façon très modulaire. Il est construit comme un graphe flot de données à gros grain, composé d'objets indépendants, qui communiquent entre eux à travers des liens de communication, de façon analogue à Pablo [60]. Les sommets du graphe sont les composants d'analyse tandis que les arêtes sont des liens de communication. Les données qui parcourent les liens sont des objets représentant les entités —événements, états des fils d'exécution, communications, etc.— du programme analysé (voir figure 3.8). Un environnement de visualisation donné est réalisé en connectant les composants souhaités pour constituer un graphe qui représente la façon dont les données seront analysées.

Contrairement à Pablo, le graphe des composants de Pajé n'est pas un graphe flot de données pur : quelques composants sont reliés par des liens bidirectionnels pour l'échange des signaux de contrôle. Le graphe de contrôle est principalement nécessaire à la mise en place de l'interactivité dans Pajé. Le flot de contrôle et son

interaction avec le flot des données sont décrits dans la section 3.4.1.3.

La figure 3.8 représente un exemple d'un graphe flot de données simple, comprenant un lecteur de traces, un simulateur, deux modules de visualisation et un module de filtrage. Le fichier de traces est lu à partir du disque par le lecteur de

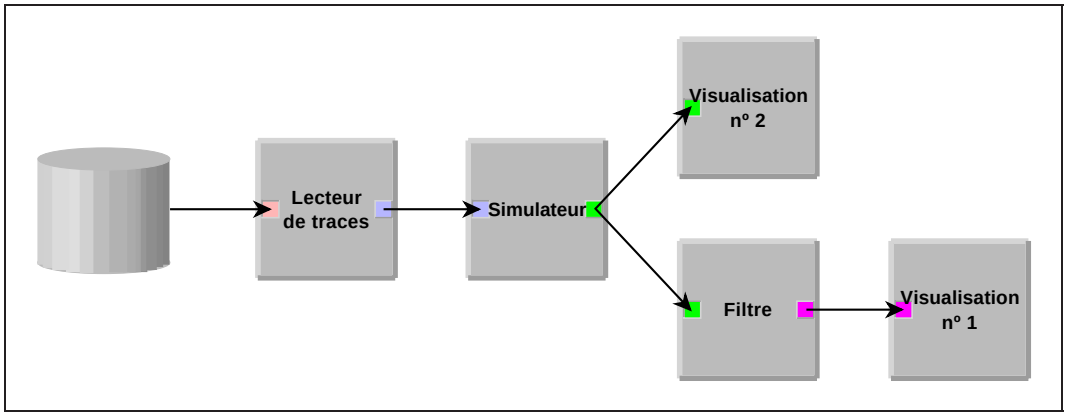

Figure 3.8 *Exemple de graphe flot de données*
Le lecteur de traces produit des objets représentant des événements à partir des données lus du disque. Ces événements sont employés par le simulateur pour produire des objets plus abstraits, comme les états des fils d'exécution, communications, etc. Les données sont filtrées avant d'être visualisées par la visualisation nº 1, tandis que la visualisation nº 2 les utilise tels quels.

traces qui génère des objets qui représentent les événements produits lors de l'exécution du programme analysé. Le simulateur utilise ces événements pour produire des objets représentant des entités plus abstraites telles que les états successifs des fils d'exécution, les communications, etc. —en simulant l'activité du programme analysé. Ces objets sont ensuite envoyés à un filtre, qui choisit quelles données seront envoyées à la visualisation nº 1. La visualisation nº 2 reçoit les données à visualiser sans qu'elles soient filtrées.

Les modules partagent une interface d'interconnexion et un protocole d'accès aux données (voir la section 3.4.2.3). Ces deux caractéristiques permettent l'extension de l'environnement par ajout de nouveaux composants au graphe. Dans la mesure du possible, les composants ont été conçus sans connaissance sémantique des données qu'ils traitent. Cette indépendance relative aux données d'entrée, combinée avec un protocole d'accès aux données bien défini, rend les composants facilement réutilisables pour traiter différents types d'entités produites par le programme parallèle (même ceux qui sont définis par l'utilisateur et qui ne sont pas connus à priori).

3.4.1.2 Composants « classiques »

Quelques composants de notre environnement peuvent être trouvés dans d'autres environnements de visualisation existants [72, 60, 41] : contrôleur, lecteur de traces, simulateur de programmes parallèles, etc.

Contrôleur : ce module est toujours présent. Il n'est pas inséré dans le graphe flot de données et n'est donc pas visible sur la figure 3.8. C'est le premier module à exécuter. Il réalise le chargement des autres modules dans la mémoire et la création du graphe. Après l'initialisation, ce module est responsable du contrôle global de l'environnement et particulièrement de l'utilisation de la mémoire.

Lecteurs de traces : des lecteurs pour deux versions d'ATHAPASCAN-0 et pour le format de trace Alog employé par l'outil upshot [35] et par le traceur de l'IBM SP/1 [70] ont été développés. Le format de traces actuel d'ATHAPASCAN-0 est auto-défini ce qui permet de rajouter simplement de nouveaux types d'événements.

Simulateur : nous avons développé un simulateur pour les événements ATHAPASCAN-0. En analysant les événements, le simulateur reproduit les états des fils d'exécution, les communications, les états des sémaphores, etc. utilisés par le programme parallèle lors de son exécution. En plus du simulateur ATHAPASCAN-0, nous avons aussi développé un simulateur générique, qui permet la définition de nouveaux types de données par l'utilisateur. Ce mécanisme puissant permet d'étendre facilement les capacités de Pajé, pour visualiser des données spécifiques à l'application ou pour s'adapter à d'autres modèles de programmation parallèle. Il est décrit dans le chapitre 4.

3.4.1.3 Limitations du graphe flot de données

Structurer l'environnement comme un graphe flot de données est bien approprié à la mise en place des composants ayant besoin d'un accès unique à chaque information dérivée de la trace. En revanche, cette solution pose problème lorsque l'environnement est interactif et doit réagir aux actions de l'utilisateur sur les représentations graphiques des données, puisqu'il faut en général accéder à la donnée correspondant à la représentation graphique pour répondre à l'action de l'utilisateur.

Comme exemple de module qui n'accède qu'une seule fois aux données, considérons un composant qui calcule les taux d'utilisation des nœuds à partir des états des fils d'exécution. Le taux d'utilisation correspond à la somme des temps occupés par les fils d'exécution divisée par le temps total d'exécution[3]. Ce module

[3]Ce calcul peut être plus complexe si on considère des nœuds avec plusieurs processeurs ou si

vérifie chaque entité du type « état d'un fil d'exécution » produite par le simulateur. Si l'entité correspond à un état occupé, sa durée est ajoutée à un accumulateur associé au nœud du fil d'exécution considéré. L'objet qui représente cet état n'a ensuite plus besoin d'être consulté. Un autre exemple est un module passif de visualisation, qui affiche une représentation visuelle correspondant à chaque entité, qui ne peut être ni interrogée ni changée. Après avoir été affichée, l'entité n'est plus consultée.

Cependant, les modules de visualisation interactifs ou qui permettent d'être réaffichés doivent accéder aux entités à chaque interaction ou réaffichage. Dans un graphe flot de données « pur », chaque entité traverse le graphe une seule fois, après avoir été produite. Dans ce cas, pour pouvoir accéder plusieurs fois aux mêmes entités, un module interactif devrait, soit stocker ces données, soit les recevoir à nouveau du graphe flot de données chaque fois qu'elles sont nécessaires. La première solution aurait pour conséquence l'augmentation de la complexité de tels modules, pour contrôler un grand volume de données, aussi bien que la réplication de données si plusieurs modules de ce type étaient utilisés. La deuxième solution aurait comme conséquence des coûts additionnels de calcul pour lire et simuler la trace plusieurs fois. Nous avons utilisé dans Pajé un hybride de ces deux solutions, qui consiste à avoir un stockage centralisé des données (appelé « fenêtre d'observation ») et à utiliser des messages de contrôle dans le graphe de composants, pour accéder à ces données.

3.4.2 La fenêtre d'observation et l'encapsuleur

3.4.2.1 Définition

Pour surmonter la contradiction entre le besoin de structurer l'environnement comme un graphe flot de données pour en faciliter l'extensibilité, et la nécessité d'accéder plusieurs fois aux mêmes données pour permettre l'interactivité, les données produites par le simulateur et utilisées pour la visualisation sont stockées dans une structure de données appelée *fenêtre d'observation*. La fenêtre d'observation contient les données correspondantes à un certain intervalle de l'exécution du programme visualisé. Cet intervalle correspond aux données avec lesquelles l'utilisateur peut interagir ; la fenêtre d'observation « glisse » sur l'échelle de temps, de façon transparente pour l'utilisateur, toujours qu'il déplace une visualisation temporelle en dehors de la fenêtre courante (voir section 3.4.2.5). L'objectif de la fenêtre d'observation est de permettre l'interactivité tout en conservant les capacités d'extensibilité d'un outil modulaire et en limitant l'utilisation mémoire, pour ne pas empêcher la « scalabilité » de l'outil. L'organisation des données dans la fenêtre d'observation est assez complexe, en raison des contraintes de performance pour

les changements de contexte entre fils d'exécution ne sont pas tracés.

accéder les données, imposés par l'interactivité (cette organisation est présentée dans la section 5.2).

La fenêtre d'observation est produite par un composant appelé *« encapsuleur »*, à partir des entités produites par le simulateur. L'encapsuleur est placé juste après le simulateur dans le graphe de composants. Les successeurs de l'encapsuleur dans le graphe de composants utilisent la fenêtre d'observation comme moyen d'accès aux entités. L'encapsuleur casse l'aspect flot de données pur du graphe de composants : chaque entité reçue par l'encapsuleur est liée à la fenêtre d'observation. Avant l'encapsuleur, le flot des informations sur le graphe est déclenché par la disponibilité des données. Après l'encapsuleur, ce flot est explicitement lancé par des *messages de contrôle* sur demande des composants qui en ont besoin. Les messages de contrôle peuvent aller dans les deux directions (voir figure 3.9). L'architecture

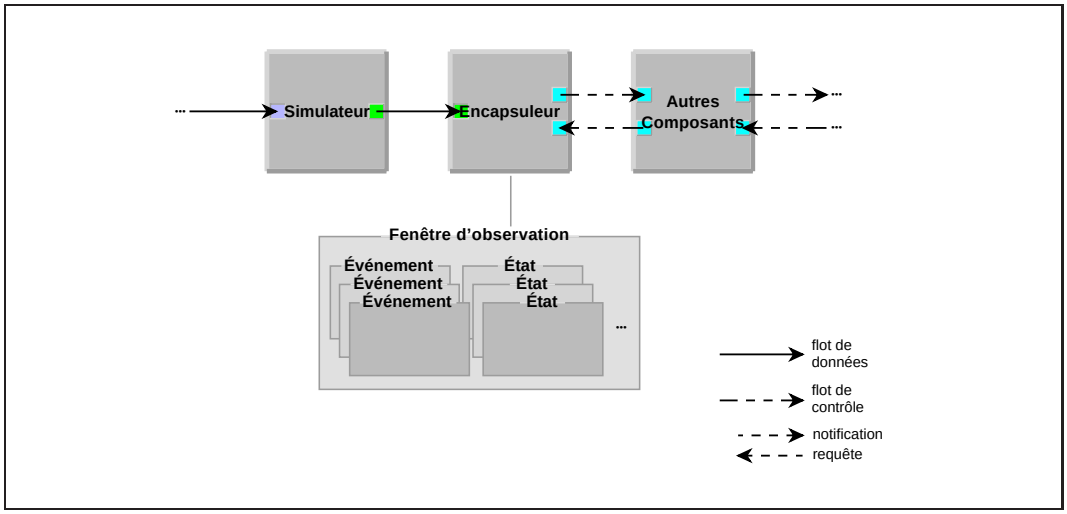

Figure 3.9 *Encapsuleur et fenêtre d'observation*
L'encapsuleur organise l'accès par d'autres composants à toutes les entités produites par le simulateur, en utilisant de la fenêtre d'observation. Après l'encapsuleur, les composants sont liés par des liens de contrôle, au lieu des liens de données utilisés avant lui.

d'interconnexion de composants de Pajé ne force pas l'existence de l'encapsuleur ni d'un graphe flot de contrôle. On peut avoir un graphe flot de données pur avec Pajé, sans utiliser l'encapsuleur et en utilisant des modules qui ne dépendent pas de son existence. L'utilisation de la fenêtre d'observation a néanmoins des avantages qui justifient largement cet abandon d'un graphe flot de données, tels que la centralisation du contrôle de la mémoire (voir session 3.4.2.5) et la construction de visualisations interactives (voir chapitre 5).

3.4.2.2 Organisation hiérarchique des données de la fenêtre d'observation

Dans Pajé il a été décidé d'organiser les données hiérarchiquement pour refléter la hiérarchie du modèle de programmation. La seule contrainte imposée par les composants de Pajé sur l'organisation des données est leur structuration hiérarchique. Cependant cette hiérarchie n'est pas figée ce qui facilite l'extension de Pajé : les composants peuvent en effet traiter des données non prévues à l'origine, pourvu qu'ils puissent accéder à une description de la hiérarchie de ces données.

Pour éviter la confusion que l'emploi du mot « objet » pourrait introduire, nous appelons dans ce qui suit entités et conteneurs les objets manipulés par Pajé. Une **entité** est donc une donnée élémentaire telle qu'un événement particulier dans l'exécution d'un processus, un certain état d'un fil d'exécution, une communication, etc. Les entités sont représentables graphiquement, et sont habituellement présentes en grande quantité. Les entités sont en général créées par le lecteur de traces à partir d'un fichier de traces, ou par le simulateur, mais peuvent aussi être produites par d'autres modules : par exemple, un module d'analyse peut produire les états des nœuds à partir des états des fils d'exécution produits par le simulateur.

Les **conteneurs** sont des objets de plus haut niveau qui rassemblent des entités élémentaires ou d'autres conteneurs. Par exemple : tous les événements qui ont eu lieu dans le fil d'exécution 1 du nœud 0 appartiennent au conteneur « fil d'exécution 1 du nœud 0 » ; par ailleurs, le conteneur « fil d'exécution 1 du nœud 0 » appartient au conteneur « nœud 0 ». Cette relation d'appartenance entre entités et conteneurs définit un graphe hiérarchique de forme arborescente.

Les entités et les conteneurs possèdent un **type**. Par exemple :

- Entités de type « événement » : envoi de message, appel de fonction, etc.
- Entités de type « état d'un fil d'exécution » : actif, bloqué en réception, etc.
- Conteneurs de type « nœud » : nœud 0, nœud 1, etc.

La hiérarchie des types d'entités et de conteneurs construite par le simulateur d'ATHAPASCAN-0 est représentée dans la figure 3.10. Les feuilles de ce graphe, qui sont dessinées en gris dans la figure, représentent les **types d'entités**. Les nœuds internes du graphe, dessinés en clair, représentent les **types de conteneurs**.

Une exécution d'un programme est représentée par Pajé par un arbre comportant les conteneurs correspondants aux objets mis en jeu durant cette exécution et les entités correspondant à la séquence d'événements, états, etc., vécue par ces objets pendant l'exécution. Cet arbre est construit à partir de l'arbre de types, et possède une structure similaire. Un exemple d'arbre simple, représentant l'exécution d'un programme ATHAPASCAN-0 sur deux nœuds, avec deux fils d'exécution par nœud et avec un verrou d'exclusion mutuelle sur le deuxième nœud est montré dans la

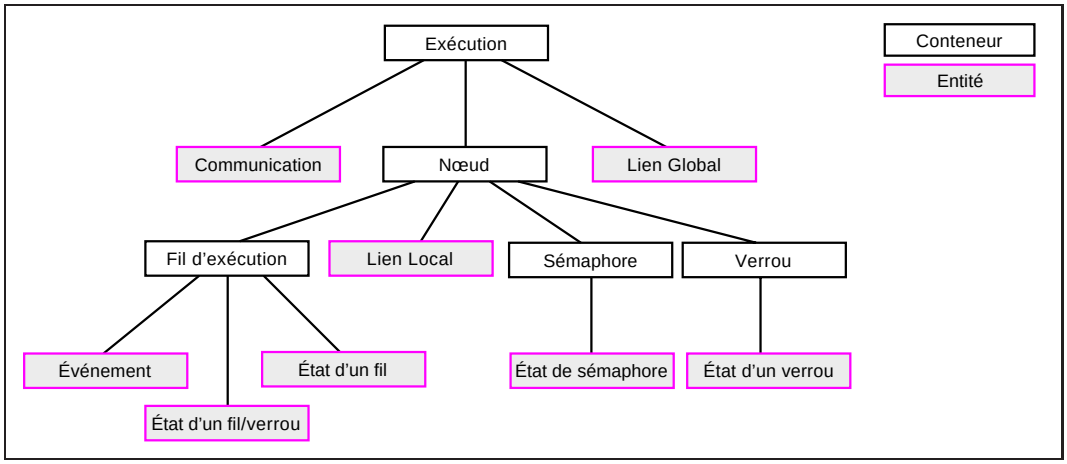

Figure 3.10 *Hiérarchie des types des entités et conteneurs d'*ATHAPASCAN*-0*
Les nœuds du graphe qui ont une couleur plus foncée sont les types des entités (les autres sont les types des conteneurs). Un « lien local » est une entité capable de montrer une liaison entre deux autres entités d'un même nœud, comme entre un événement de libération d'un verrou et le déblocage correspondant d'un fil d'exécution. Un « lien global » représente une liaison entre entités de nœuds différents, comme entre un événement de création d'un fil d'exécution à distance et le début de l'exécution du fil créé.

figure 3.11. Les entités montrées sur cette figure sont, pour chaque fil d'exécution,

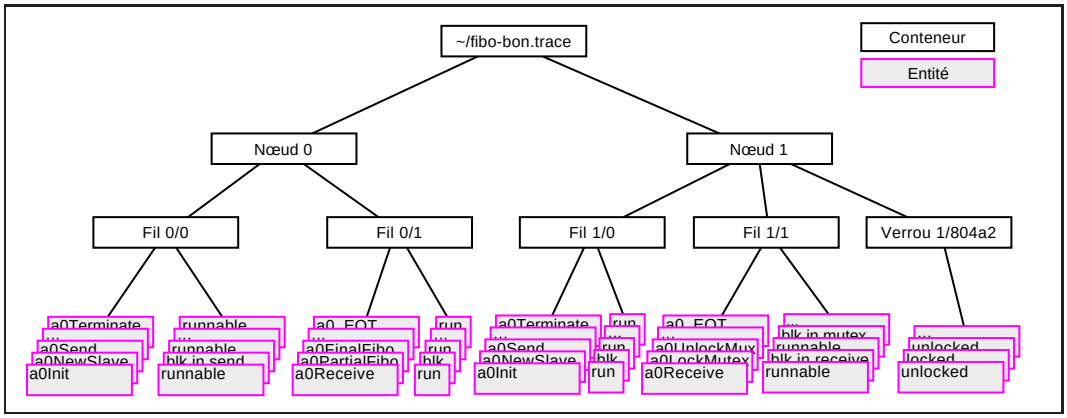

Figure 3.11 *Arbre répresentant une exécution* ATHAPASCAN*-0*

les événements qu'il a produit et les divers états pris au cours de son exécution. Pour le verrou d'exclusion mutuelle, les entités montrées dans la figure représentent sa séquence d'états.

3.4.2.3 Messages de contrôle

Il existe deux types de messages de contrôle dans Pajé : les messages envoyés vers les composants successeurs dans le graphe (de gauche à droite dans la figure 3.9), appelés *notifications* et les messages envoyés dans la direction opposée, appelés *requêtes*.

Les notifications sont utilisées pour informer les autres composants que les données ont changé. Ces changements peuvent être causés par l'ajout de données dans la fenêtre d'observation suite à la lecture d'une nouvelle partie de la trace ou par un changement de configuration d'un module de filtrage, par exemple. Des notifications différentes permettent d'annoncer les différents changements possibles :

- dans la hiérarchie des types de conteneurs et d'entités ;
- dans la hiérarchie de conteneurs et entités ;
- dans la position de la fenêtre d'observation ;
- dans les couleurs des entités.

Les requêtes sont employées par les composants « consommateurs de données » (par exemple un composant de visualisation) pour obtenir les données dont ils ont besoin. Les requêtes sont envoyées au composant précédent, qui peut répondre directement, s'il dispose de l'information demandée, ou transmettre la requête au module qui le précède. Si la requête arrive jusqu'à l'encapsuleur et les données requises ne sont pas dans la fenêtre d'observation, le contrôleur est informé, pour qu'il puisse redémarrer la lecture de la trace. Les types de requêtes existantes permettent :

- D'obtenir des informations globales sur l'exécution ou sur la fenêtre d'observation (structure hiérarchique des types d'entités et de conteneurs, durée de l'exécution).
- D'identifier les conteneurs apartenant à un niveau donnée de la hiérarchie (les nœuds, les fils d'exécution du nœud 1, etc., dans le cas d'ATHAPASCAN-0).
- D'accéder aux entités, choisies par date et par type : par exemple, tous les événements du fil d'exécution 1 du nœud 7 entre 3,2 et 4,3 secondes d'exécution.
- D'obtenir plus d'informations à propos d'une entité, comme le conteneur auquel elle appartient, la date qui lui est associée, sa forme, sa couleur, les objets qui lui sont liés, les noms des attributs de l'entité, la valeur d'un attribut à partir de son nom. Les 2 derniers permettent le développement de modules de filtrage génériques, qui filtrent les entités à partir de la valeur de ces attributs.
- De demander l'inspection d'une entité. Cette fonctionnalité est utilisée par des composants de visualisation quand l'utilisateur sélectionne une représentation graphique d'une entité ; le composant de visualisation n'a pas besoin de connaître tous les détails de l'objet examiné.

Toute la complexité de stockage et d'accès à la grande quantité de données produites à partir de la trace est isolée dans la fenêtre d'observation. En plus de simplifier la construction des modules « consommateurs de données », l'accès centralisé aux données a d'autres avantages, notamment pour la construction des filtres (voir la section 6.3.1) et pour la gestion mémoire par le composant contrôleur. Chaque fois qu'un composant émet une requête pour obtenir des données qui ne sont pas dans la fenêtre d'observation courante, l'encapsuleur informe le composant contrôleur, qui peut relancer la lecture des traces jusqu'à ce que les données requises soient produites et stockées dans la fenêtre d'observation.

3.4.2.4 Exemple : fonctionnement d'un module de visualisation

Pour expliquer l'interaction entre un module de visualisation et le reste de Pajé à travers les messages de contrôle, prenons comme exemple le module de visualisation qui réalise un diagramme espace-temps. Une image du diagramme produit par ce module peut être vue dans la figure 3.3. Dans ce diagramme, l'axe horizontal représente le temps, tandis que l'axe vertical représente la hiérarchie des conteneurs et entités. Par exemple, pour la hiérarchie d'ATHAPASCAN-0 représentée dans la figure 3.11, l'espace vertical est divisé entre les nœuds ; l'espace de chaque nœud est divisé entre ses fils d'exécution, ses sémaphores et ses verrous d'exclusion mutuelle ; l'espace de chaque fil d'exécution, à son tour, est divisé entre les événements et les états de ce fil d'exécution. Un exemple d'une division de l'espace est montrée dans la figure 3.12.

Le diagramme espace-temps est produit en deux phases : la construction de la structure qui définit la distribuition spaciale des entités et l'obtention et l'affichage des entités. Dans ces deux phases, le module de visualisaton obtien les données necéssaires des autres modules, à travers des requêtes, comme expliqué ci-dessous.

Construction de la représentation spatiale L'espace alloué à chaque conteneur n'est pas constant, car la quantité de types d'entités et de sous-conteneurs est différente d'un conteneur à un autre (dans la figure 3.12, le nœud 0 possède trois fils d'exécution, tandis que le nœud 1 en a deux). La quantité de conteneurs ne reste pas fixe pendant une exécution : des conteneurs (fils d'exécution, sémaphores, verrous d'exclusion mutuelle) sont créés et détruits dynamiquement. Le module de visualisation doit donc adapter l'espace alloué à chaque conteneur au fur et à mesure que la simulation du programme progresse. Il se peut aussi que des nouveaux types de conteneurs soient créés ou détruits dynamiquement pendant l'exécution du programme (voir section 4.3 sur l'extensibilité de Pajé et la section 6.3.1 sur le filtrage des données), ce qui demande aussi une adaptation de l'espace alloué.

Des notifications sont envoyées par l'encapsuleur (ou par les modules de fil-

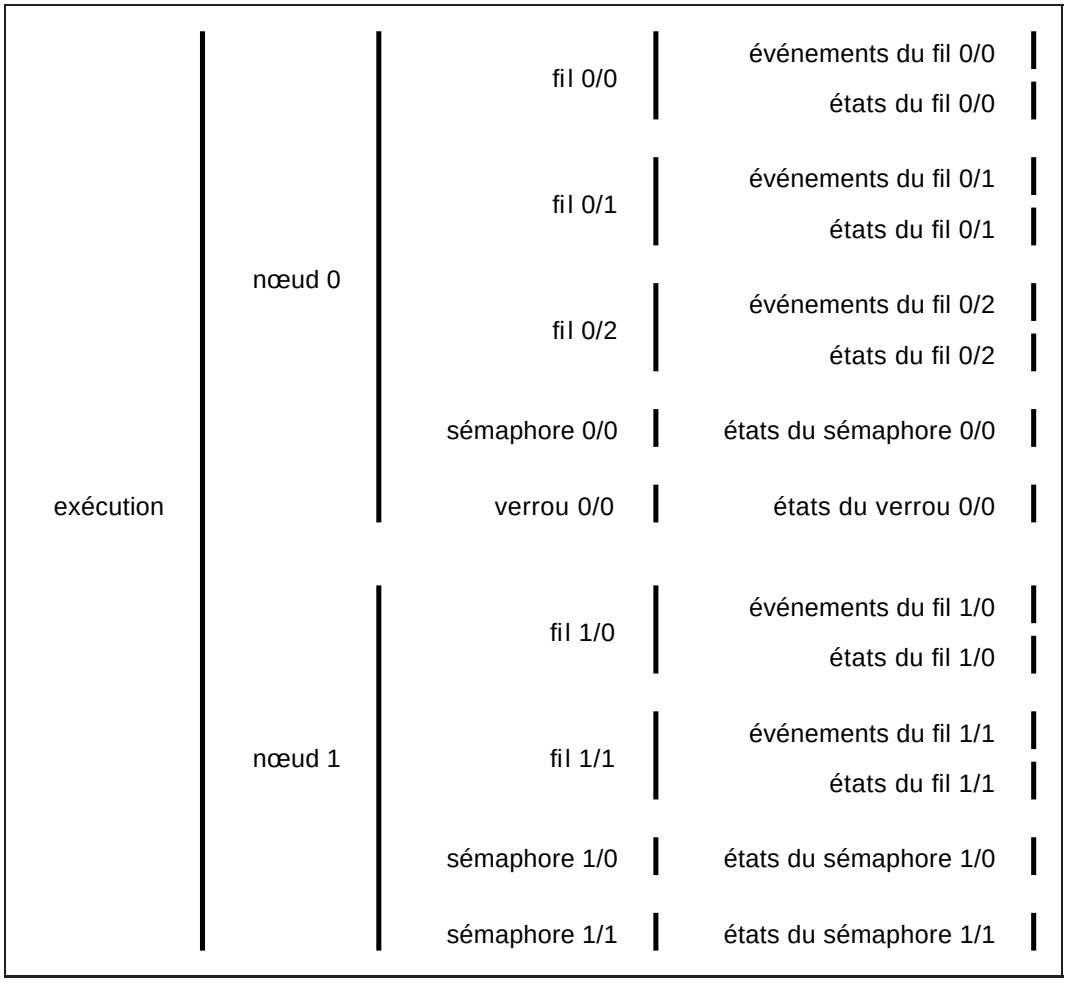

Figure 3.12 *Représentation spatiale de la hiérarchie d'entités d'*ATHAPASCAN*-0*

trage) chaque fois que la hiérarchie des types ou la hiérarchie des entités est changée. En recevant ces notifications, le module de visualisation recalcule l'espace et la position des représentations des conteneurs sur l'écran. Pour ce faire, il envoye une requête demandant la hiérarchie de types. Après, il parcourt la hiérarchie de types et construit la hiérarchie des conteneurs et entités. Cela est fait par des requêtes successives demandant les conteneurs existants dans chaque niveau de la hiérarchie. À cette hiérarchie d'entités le module ajoute aussi l'espace nécessaire pour représenter chaque type d'entité, pour arriver enfin à la structure qui représente la division spatiale de l'écran, comme celle de la figure 3.12. La construction de cette structure utilise donc deux types de requêtes différentes : le premier pour obtenir la hiérarchie de types et le deuxième pour obtenir les conteneurs à chaque niveau de la hiérarchie.

Obtention et affichage des entités Supposons que le composant de visualisation doive redessiner une partie de l'écran. Tout d'abord, il doit découvrir quel est l'intervalle de temps et les types des entités nécessaires pour redessiner cette partie de l'écran. Puis, pour chaque type d'entité, il doit envoyer une requête au composant qui le précède dans le graphe de composants, pour obtenir les entités de ce type qui appartiennent à l'intervalle de temps considéré. Pour dessiner chaque entité, il doit savoir sa forme, couleur, position et taille. Ces informations sont également obtenues par des requêtes envoyées au composant précédent.

N'importe laquelle des informations obtenues à partir d'une requête peut être changée par des filtres, qui peuvent ainsi réaliser des changements profonds dans l'affichage (voir section 6.3.1).

3.4.2.5 Déplacement de la fenêtre d'observation

Les données contenues dans la fenêtre d'observation sont directement accessibles par les modules de l'environnement, aussi souvent qu'ils en ont besoin. Il est donc possible de réaliser des modules de visualisation qui permettent à l'utilisateur d'interagir avec les objets visualisés et de déplacer la visualisation en avant et en arrière dans le temps — les données nécessaires pour le réaffichage étant accessibles à partir de la fenêtre d'observation.

Cependant, puisque la fenêtre d'observation est stockée dans la mémoire de l'ordinateur, il se peut que sa capacité ne soit pas suffisante pour contenir toutes les données lues dans le fichier de traces ou produites par le simulateur. Dans ce cas, la fenêtre d'observation ne contient qu'une partie des données, et c'est seulement sur cette partie que l'accès peut se faire librement.

En limitant la consommation mémoire, la fenêtre d'observation joue un rôle important dans les capacités de « scalabilité » de Pajé. D'un côté, cette limitation lui permet de traiter des fichiers qui dépassent les capacités de la mémoire, sans pour autant empêcher l'interactivité. D'autre côté, l'opération très fréquente de rechercher une donnée spécifique est plus facilement optimisable, vu que la quantité de données est limitée.

Sous l'action de l'utilisateur, lorsqu'il essaie de visualiser au delà de la limite courante de la fenêtre d'observation, l'encapsuleur actionne le contrôleur pour qu'il commande la lecture et la simulation de nouvelles données. Ces nouvelles données sont incluses dans la fenêtre d'observation tandis que les objets les plus anciens sont enlevés. On dit que la fenêtre d'observation « glisse » dans le temps. Lors d'un glissement de la fenêtre d'observation, l'encapsuleur envoie une notification aux autres modules pour qu'ils puissent s'adapter aux nouvelles limites.

Un problème plus intéressant se pose lorsque l'utilisateur essaie de déplacer

une visualisation en deçà de la limite inférieure de la fenêtre d'observation. Dans ce cas, pour permettre le glissement de la fenêtre de visualisation, il faut récupérer les données correspondantes aux états du système avant l'état de début de la fenêtre d'observation courante.

L'implantation d'un simulateur capable de générer les états d'un programme dans l'ordre inverse est très difficile, dans la plupart des cas impossible. La réinitialisation du simulateur et relecture du fichier de traces à partir de son début peut représenter la simulation de presque toute la trace, si la fenêtre d'observation est placée à la fin de l'exécution. Cette opération est en général trop coûteuse car dans ce cas la trace a une taille importante puisqu'elle dépasse la capacité de la fenêtre d'observation. La simulation à partir d'un endroit quelconque de la trace n'est pas non plus possible, puisque la simulation d'un événement dépend, dans la plupart des cas, de l'état des objets simulés.

La solution adoptée dans Pajé consiste à remettre le simulateur dans l'état correspondant au temps à partir duquel on veut les données, positionner le lecteur de traces dans l'événement correspondant et reprendre la simulation à partir de ce point. Cette solution est similaire à celle proposée par Scope [1]. Il faut donc enregistrer l'état du simulateur et du lecteur de traces de façon à pouvoir les reprendre. L'état du simulateur correspond à l'état de tous les objets qu'il simule, ce qui peut être assez important. La fréquence des enregistrements ne doit donc pas être trop élevée, sous peine d'utiliser trop d'espace disque pour ces enregistrements et de consommer trop de temps en les enregistrant. Nous avons utilisé dans Pajé un algorithme simple, qui consiste à enregistrer l'état de la simulation à intervalles de temps de simulation réguliers[4]. Si nécessaire, la simulation est relancée à partir de l'état sauvegardé le plus proche, avant la date requise.

Une extension envisagée de Pajé consiste à utiliser les enregistrements des états de la simulation d'une session de visualisation à l'autre. De cette façon, l'utilisateur va pouvoir reprendre la visualisation à partir d'un instant quelconque déjà visualisé, sans devoir attendre le déroulement de la simulation depuis le début de la trace.

L'existence de la fenêtre d'observation et le déclenchement automatique de la lecture des traces quand un module accède à des données qui ne sont pas en mémoire simplifie la construction des modules. Ils n'ont pas à se préocuper des problèmes de gestion mémoire : ils peuvent accéder aux données comme si elles étaient toutes en mémoire principale. Il existe néanmoins des motifs d'accès aux données à éviter, parce que, même si les données voulues sont obtenues, les performances de l'outil peuvent s'écrouler. Par exemple, l'accès à une seule donnée au début de la trace, qui ne poserait aucun problème si toute la trace était en mémoire, peut se traduire par

[4]Cet intervalle correspondent en fait à la largeur de la tranche de temps employée dans l'algorithme de recherche des données, voir section 5.2.3.5.

une relecture complète de la fenêtre d'observation, si le début de la trace n'est plus dans la mémoire. Ce problème se pose surtout pour les filtres, et il est traité dans la section 6.3.3.

3.5 Conclusion

Cette thèse s'est déroulée dans le cadre du projet APACHE, qui développe un environnement de programmation parallèle pour le calcul haute performance. Cet environnement, appelé ATHAPASCAN permet d'atteindre un bon compromis performance portabilité. Il possède une interface de programmation basée sur des tâches asynchrones et des données partagées (ATHAPASCAN-1) s'appuyant sur un noyau exécutif qui implante un réseau dynamique de fils d'exécution communiquants (ATHAPASCAN-0). Ce sont les traces produites par ce noyau exécutif qui ont permis de valider l'outil de visualisation Pajé.

Pajé a été utilisé dans le projet APACHE pour améliorer les performances de « grosses » applications et en particulier d'une application de dynamique moléculaire.

L'architecture de Pajé a été conçue pour combiner les propriétés d'extensibilité, d'interactivité et de « scalabilité ». L'extensibilité a conduit à structurer Pajé en graphe de composants indépendants et génériques, susceptibles de traiter toute structure de données hiérarchique dont la description leur est fournie. La combinaison entre les contraintes d'interactivité et de « scalabilité » — qui impose de limiter autant que possible la consommation mémoire — ont mené à centraliser le stockage et l'accès aux données. L'interactivité imposant un accès très efficace aux données, leur structuration a fait l'objet d'études approfondies qui sont présentées dans un chapitre ultérieur. L'impératif de « scalabilité » a en outre conduit à la définition d'un mécanisme de filtrage qui sera lui aussi décrit dans un autre chapitre.

Un mot enfin sur l'implémentation. Afin de faciliter la structuration de l'outil en modules aussi indépendants que possibles, Pajé a adopté une conception orientée objet. Sa mise en œuvre a été facilitée en adoptant l'environnement de développement OpenStep[52] — qui fournit une bibliothèque puissante d'objets prédéfinis — et le langage Objective-C [43] qui lui est associé.

4

Extensibilité

Le domaine du calcul parallèle étant loin d'être figé, il est indispensable de tenir compte de l'évolution des modèles de programmation parallèle. Si l'outil de visualisation n'est pas capable de suivre ces évolutions, il risque de ne plus être utilisable rapidement, et de voir son cycle de vie arriver à une fin prématurée. Pour ne pas être rapidement dépassé par les progrès de la programmation parallèle, un environnement de visualisation doit donc être facilement adaptable. Les changements auxquels l'outil doit s'adapter comportent les évolutions des paradigmes de programmation parallèle, le développement de nouvelles techniques d'analyse, filtrage et visualisation des données ainsi que l'apparition de besoins particuliers de certains utilisateurs pour certaines applications. Il est vraisemblable que la prolifération d'outils de visualisation soit due pour l'essentiel à l'incapacité ou à la difficulté d'adapter les outils existants à de nouveaux besoins.

Nous définissons l'extensibilité comme la capacité offerte par l'environnement de visualisation à être adapté facilement à ces changements. Nous considérons dans ce chapitre trois formes d'extensibilité : la capacité de configurer les composants de l'outil pour les adapter à des situations diverses ; la facilité de construire et d'intégrer de nouveaux composants à l'environnement ; la dernière forme d'extensibilité qui a été considérée et qui est aussi la plus importante est la possibilité d'adapter Pajé sans avoir à le modifier. En effet, quel que soit le soin apporté à rendre l'architecture de Pajé aussi modulaire que possible et le développement de nouveaux modules aussi facile que possible, son extension suppose une connaissance assez poussée de l'implantation de l'environnement. En revanche, la possibilité qui est offerte ici permet, dans certaines limites qui seront exposées plus loin, d'adapter l'outil à différents modèles de programmation ou aux besoins particuliers, et sou-

vent éphémères, de certaines applications, sans avoir à le toucher aucunement. La principale difficulté du traitement de plusieurs modèles de programmation par un outil d'analyse et de visualisation de traces d'exécution tel que Pajé provient de la dépendance du simulateur de l'environnement relativement à la sémantique des traces. La solution adoptée dans Pajé pour s'affranchir de cette difficulté a été de concevoir un simulateur générique. Ce simulateur est configuré par des événements spéciaux insérés dans la trace d'exécution par des commandes que le programmeur ajoute à son programme : Pajé comporte ainsi un petit langage de commande destiné à parametrer le simulateur pour une analyse ou un modèle de programmation particulier.

Dans ce chapitre nous présentons successivement les trois formes d'extensibilité considérés, à savoir configuration de l'environnement à partir de composants existants, développement de nouveaux composants et enfin adaptation de l'environnement en utilisant les possibilités du simulateur générique.

4.1 Configuration du schéma d'analyse

Nous appelons « schéma d'analyse » l'ensemble des traitements que les données de la trace subissent avant d'être présentées à l'utilisateur. La possibilité de configurer le schéma d'analyse d'un outil de visualisation permet l'utilisation d'un seul outil dans plusieurs situations différentes. Plus un outil de visualisation offre à ses utilisateurs de possibilités de changements du schéma d'analyse, plus grandes seront les possibilités d'extension de la gamme de problèmes traitables par l'outil sans avoir à développer des nouveaux composants. Ces changements du schéma d'analyse se traduisent par le choix des traitements à effectuer sur les données, l'ordre d'application de ces traitements et la configuration de chaque traitement.

4.1.1 Définition d'un schéma d'analyse

La contrainte d'extensibilité peut influencer fortement la conception d'un outil de visualisation.

Les environnements de visualisation monolitiques n'offrent pas en général la possibilité de choisir le schéma d'analyse qui sera appliqué mais seulement la capacité de configurer les traitements qui seront effectués sur les données. En revanche, un outil modulaire, où chaque module réalise un traitement spécifique sur les données, indépendamment des autres modules, se prête bien à la définition d'un schéma d'analyse.

La possibilité de changer l'ordre d'application des traitements implique qu'un

module puisse traiter les données fournies par n'importe quel autre module de traitement. Cela exige une définition bien précise de la façon dont un module accède les données à traiter et la façon dont il les rend disponibles aux autres modules.

Comme ces traitements peuvent transformer les données, un module doit être prêt à traiter des données potentiellement différentes. L'indépendance des modules relativement aux types des données traitées et à leur sémantique est donc indispensable à la flexibilité de la configuration du schéma d'analyse.

Cette indépendance relativement à la sémantique des données est aussi importante pour permettre le traitement de différents types de données : un module qui calcule la taille moyenne des messages échangés durant une exécution tracée doit être utilisable pour calculer la moyenne des temps d'exécution d'une procédure. De nouveaux types de données peuvent être créés par ajout de fonctionnalités au traceur (qui générera alors ces nouveaux types de données) ou par le développement de nouveaux modules de traitements de données.

Certains outils permettant la réalisation d'un schéma d'analyse graphiquement, en utilisant un environnement de programmation visuelle [60, 1]. D'autres utilisent des fichiers de configuration comportant une description textuelle, modifiable par l'utilisateur. Dans d'autres outils, la définition du graphe d'analyse est faite dans le propre outil de visualisation, qu'il faut changer et recompiler pour changer la définition.

4.1.2 Configuration des composants

Certains composants d'un schéma d'analyse doivent être configurés, avant de pouvoir traiter des données. Ces configurations servent par exemple à :

- Séléctionner les données à traiter. On peut choisir de traiter seulement les données d'un certain type ou qui sont contenues dans un certain intervalle de temps ou encore qui sont relatives à un certain sous-ensemble de nœuds.
- Sélectionner les traitements à effectuer, si le composant le permet. Dans un composant qui fait des statistiques, par exemple, choisir l'opération à effectuer, ou paramétrer un composant qui fait des réductions ou filtrages des données ;
- Choisir des caractéristiques visuelles des réprésentations graphiques, telles que la couleur, la forme, la taille, etc., si les modules de visualisation permettent ce genre de configuration.

Ces configurations peuvent en général être enregistrées, permettant ainsi à l'utilisateur de choisir la configuration la plus adaptée au traitement qu'il veut réaliser sur ses données. Quelques outils permettent que les configurations (ou quelques

unes des configurations possibles) soient faites pendant le déroulement d'une section de visualisation, avec un résultat immédiat sur les données présentées. Cela permet de tester plus aisément les possibilités de l'outil.

4.1.3 Graphe de composants dans Pajé

Le schéma d'analyse de Pajé est chargé dynamiquement, à partir d'un fichier de configuration constitué à partir des préférences indiquées par l'utilisateur, lors du chargement d'un fichier de traces. Ce fichier contient l'identification des composants à charger (pour être trouvé, le composant doit être dans un des répertoires prévus à cet effet) et les liaisons entre composants. Après le chargement, chaque composant doit se configurer. Les informations de configuration des composants sont maintenues dans un fichier correspondant au schéma d'analyse utilisé, de telle sorte que chaque schéma possède une configuration indépendante pour ses composants. L'utilisateur peut changer la configuration des composants pendant la session de visualisation. Ces changements sont sauvés dans le fichier de configuration du schéma. C'est d'ailleurs la méthode utilisée pour configurer les schémas.

4.2 Développement de nouveaux composants

Quand les composants offerts par un environnement de visualisation ne permettent pas de réaliser les traitements ou visualisations requises par son utilisateur, il peut s'avérer nécessaire de développer de nouveaux composants. Plusieurs facteurs peuvent être à l'origine de cette nécessité :

- Le developpement et validation de nouvelles techniques d'analyse et de représentation de données.
- L'incorporation de techniques d'analyse et de représentation de données à l'environnement de visualisation.
- Des changements dans l'environnement d'exécution utilisé, qui peuvent exiger des changements dans l'environnement de visualisation.
- Visualisation de traces provenant d'un environnement d'exécution différent et d'un format différent de ceux connus par l'outil.
- Réalisation de traitements de données ou des visualisations spécifiques à une application.

Ce dernier point sera discuté dans la section 4.3, tandis que la lecture de traces de format étranger et le développement et l'intégration de nouveaux composants seront traités ci-dessous.

4.2.1 Lecture de fichiers de traces de format différent

Il est parfois intéressant d'utiliser un environnement de visualisation pour représenter des données générées par plusieurs environnements d'exécution. Les motivations sont, par exemple, la comparaison des performances d'un programme exécutée dans des environnements d'exécution différents ou l'utilisation d'un environnement de visualisation plus performant que celui prévu pour l'environnement d'exécution utilisé. L'outil Pablo [60], par exemple, est fourni avec des programmes pour réaliser la conversion entre son format SDDF et le format de PICL, utilisé par l'outil ParaGraph [33]. Dans la plupart des cas, deux environnements d'exécution généreront des fichiers de traces dans des format différents et incompatibles. La visualisation de traces d'exécution produites dans un format différent de celui utilisé par l'environnement de visualisation pose deux problèmes, le premier c'est interpréter la syntaxe du format des données ; le deuxième est la compréhension de la sémantique de ces données.

Le problème de la syntaxe est simple à résoudre ; il suffit d'écrire soit un traducteur du format de traces voulu vers le (ou un des) format(s) de traces accepté(s) par l'environnement de visualisation, soit d'écrire un nouveau module de lecture de traces et de l'intégrer à l'environnement, qui sera alors capable de lire ce nouveau format.

La construction d'un traducteur est généralement plus facile que la construction d'un module pour un environnement de visualisation, et c'est la solution adoptée dans la plupart des cas. Elle pose néanmoins le problème d'augmenter la consommation d'espace disque, puisque deux copies de la trace existeront, en plus du temps nécessaire à la conversion.

Le problème de la sémantique peut s'avérer plus compliqué. Si un outil de visualisation ne supporte pas un certain concept de l'environnement d'exécution tracé, il ne sera pas capable de le visualiser. Il faut dans ce cas soit ignorer ces données et visualiser seulement celles qui possèdent un équivalent sémantique, soit trouver une correspondance partielle entre le nouveau concept et un concept existant dans l'environnement de visualisation, soit développer des nouveaux composants, qui rendront l'environnement capable de traiter les nouvelles données. Une autre possibilité se présente si l'outil de visualisation supporte une description sémantique des données. Dans ce cas, le processus de conversion ajoute une description de la nouvelle sémantique à la trace convertie. Les chances de trouver une forme de conversion des nouvelles données vers quelque chose de compréhensible par l'outil sont alors bien plus élevées. Pajé possède une forme de description de la sémantique des données, qui est présentée dans la section 4.3.

4.2.2 Règles de développement de nouveaux composants

Pour bien intégrer un nouveau composant à un environnement de visualisation existant, il faut respecter les règles imposées implicitement ou explicitement par cet environnement. Plus l'environnement offre de support au développeur pour la connaissance et le respect de ces règles, plus il sera aisé d'utiliser cette possibilité pour étendre l'outil. On peut citer comme exemples de règles à être connues et respectées :

- Méthode d'accès aux données : comment le composant doit faire pour avoir accès aux données dont il a besoin.
- Disponibilité des données produites : comment le nouveau composant offre les données qu'il produit ou modifie aux autres composants de l'environnement.
- Présentation des données, dans le cas d'un composant de visualisation.
- Communication avec l'utilisateur (pour la configuration de l'environnement ou pour interagir avec l'utilisateur).
- Contrôle : comment le composant est contrôlé par le reste de l'environnement.
- Intégration : comment le reste de l'environnement reconnaît-il le nouveau composant (faut-il générer une nouvelle version de l'environnement, est-ce que le nouveau composant est chargé dynamiquement, etc ?).

4.2.3 Protocoles dans Pajé

Le contrôle des composants par Pajé, ainsi que leur accès aux données est défini par les messages de contrôle : requêtes et notifications (voir section 3.4.2.3). Pour simplifier la construction de composants et garantir que tous les composants implémentent toutes les requêtes et notifications définis, Pajé possède des classes qui implémentent le comportement de base d'un composant.

La classe utilisée pour implémenter les filtres, par exemple, a un comportement qui correspond à ne rien changer aux données : toutes les requêtes sont renvoyées au composant précédent dans le graphe et toutes les notifications sont renvoyées aux composants suivants. De cette façon, pour construire un nouveau filtre Pajé, il suffit de dériver cette classe (appelée `PajeFilter`) et d'implémenter seulement les méthodes correspondantes aux requêtes et notifications nécessaires au filtrage qu'on veut réaliser, les autres messages de contrôle passeront à travers le filtre intouchées.

Par exemple, pour construire un filtre simple qui change la couleur des messages qui ont une taille supérieure à un certain seuil, il suffit de créer une nouvele classe,

dérivée de `PajeFilter` et d'écrire la méthode `colorForEntity` pour cette nouvelle classe. Cette méthode reçoit en paramètre une entité et doit retourner sa couleur. Elle est employée par tout module désirant savoir la couleur d'une entité pour la présenter. L'implémentation de la méthode est assez simple : il suffit d'envoyer une requête au module précédent pour savoir le type de l'entité et dans le cas d'un message, demander sa taille et retourner, soit la nouvelle couleur, soit la couleur originale. Tout le reste (traitement des autres messages de contrôle, liaison au graphe de composants) est pris en compte par la classe `PajeFilter`.

4.3 Simulation générique

Les traitements et les visualisations de données offerts par un environnement de visualisation ne sont pas toujours suffisants pour mettre en évidence le comportement d'un programme (ou un problème de performance recherché). Un traitement ou une visualisation spécifiques aux données d'une application peut être la clé pour faciliter cette recherche. Malheureusement, un outil de visualisation n'est pas un programme simple, et l'idée de réaliser des changements à un tel programme rebute en général la plupart des programmeurs (intéressés par leurs applications et non par le développement des outils). En effet, le coût de développement semble trop grand pour le bénéfice qu'il apporte (même si parfois le coût perçu est plus important que le coût réel). Les outils de visualisation doivent donc s'attacher à rendre simple le développement de nouveaux composants et/ou offrir d'autres formes d'extensibilité.

Il y a, dans les environnements existants, une multitude de possibilités d'enregistrement de données « utilisateur », plus ou moins puissantes comme par exemple :

- Les événements de l'utilisateur, qui servent à repérer des instants ou intervalles de temps spécifiques de l'exécution [14]. Certains outils permettent d'ajouter des valeurs arbitraires, qui correspondent généralement à des indices de performance spécifiques calculés par l'application et qui peuvent ensuite être traités et visualisés par l'outil [60, 54].
- Le marquage de commencement et terminaison de régions internes au programme. ParaGraph [33], par exemple, comporte des visualisations pour montrer des indices de performances correspondants à une région définie par l'utilisateur dans un programme PICL.

Ces possibilités d'extension permettent la visualisation de quelques informations simples, spécifiques de l'application. Elles sont néanmoins limitées si on veut visualiser des données plus complexes, comme l'évolution des états d'un objet quelconque de l'application, des relations entre plusieurs données ou encore des

modèles de programmation plus évolués implantés sur l'environnement de programmation de base. Un environnement de visualisation classique ne permet en général pas de simuler un modèle de programmation différent de celui pour lequel il a été conçu sans l'écriture d'au moins un nouveau module de simulation.

Pour permettre ces types d'extension, Pajé possède un simulateur générique, contrôlé par des événements spéciaux contenus dans le fichier de traces. Des événements spécifiques permettent de définir les nouveaux types de données à visualiser et de créer des données de ces nouveaux types. Les paragraphes 4.3.2 et 4.3.3 décrivent les primitives permettant d'introduire ces événements dans la trace d'exécution.

Ces primitives sont de deux sortes :

- Définition de nouveaux types de conteneurs et d'entités (voir tableau 4.1).
- Création de conteneurs et d'entités des nouveaux types (voir tableau 4.2).

Ainsi, pour pouvoir visualiser un nouveau type de données, un programmeur doit insérer des appels à ces primitives directement dans l'application à tracer. Chaque appel génère un événement dans la trace, qui contient tous les paramètres de la primitive lesquels seront donc transmis de cette façon au simulateur.

Lors de la simulation, les descriptions des nouvelles données seront ajoutées par le simulateur à la hiérarchie des types de conteneurs et d'entités (voir section 3.4.2.2 page 67), de façon à être connues et affichées par les modules de filtrage et de visualisation.

La suite de cette section présente tout d'abord un programme qui sera utilisé comme exemple d'ajout de nouvelles données à visualiser, puis une description des primitives de définition et de génération des données, un exemple un peu plus complexe et, enfin, une évaluation de cette forme d'extension.

4.3.1 Présentation de l'application exemple

Soit un programme, constitué de fils d'exécution, qui réalise un calcul en phases, et supposons que l'on veuille visualiser l'évolution de ces phases de calcul dans chaque nœud qui exécute le programme. C'est le cas par exemple des applications scientifiques qui nécessitent la résolution de grands systèmes linéaires en parallèle : pour résoudre ce type de problème, certaines méthodes numériques se servent d'un schéma de calcul itératif alternant deux types de phases : les phases de « calculs locaux », durant lesquelles les calculs effectués sur chaque nœud n'utilisent que des données locales à ce nœud, et les phases de « calculs globaux », où les calculs nécessitent des données provenant d'autres nœuds. L'algorithme de l'application est montré, de façon schématique, dans la figure 4.1. Le programme réel est plus complexe, puisqu'il possède plusieurs fils d'exécution pour réaliser les différents

```
initialisation();
while (!converge) {
    iter++;
    calcul_local();
    send (local_data);
    receive (remote_data);
    calcul_global();
}
```

Figure 4.1 *Algorithme simplifié du programme exemple*

calculs. Nous voulons montrer, en plus des informations de communication et synchronisation normalement visualisées, la phase de calcul réalisée par chaque nœud à tout instant de l'exécution.

4.3.2 Définition de nouveaux types de données

4.3.2.1 Définition d'un nouveau type d'entité ou de conteneur

La hiérarchie de types est calculée par le simulateur, qui la fournit aux autres modules de Pajé. Pour permettre l'ajout de nouveaux types de données à la volée, nous avons doté le simulateur de Pajé de la possibilité de changer cette structure. Ces changements sont effectués lors de la simulation d'événements spéciaux contenus dans le fichier de traces. Pour générer ces événements, nous avons défini un ensemble de primitives qui doivent être insérés dans le programme parallèle de façon à ce que le programmeur puisse contrôler le simulateur.

Une primitive permet de créer des nouveaux types de conteneurs et quatre primitives sont destinées à créer des types d'entités, utilisées selon les entités que l'on veut créer : événements, états, liens ou variables. Un « événement » est une entité qui représente une action instantanée. Les « états » qui nous intéressent sont ceux des conteneurs. Un « lien » représente une liaison quelconque entre deux conteneurs, où un conteneur est identifié comme « source » et l'autre comme « destinataire ». Une « variable » contient l'évolution temporelle des valeurs d'une grandeur quelconque associée à un conteneur.

Les types de données définis par l'utilisateur s'ajouteront à la hiérarchie de types de conteneurs et d'entités présentée section 3.4.2.2 page 67. La hiérarchie d'ATHAPASCAN-0 est reproduite dans la figure 4.2. Les primitives du traceur permettant la définition de nouveaux types de conteneurs et d'entités sont listées dans le tableau 4.1. Les arguments de ces primitives définissent le nom du nouveau type et la position dans la hiérarchie de types où il doit être ajouté. Le résultat de l'appel sert à identifier le nouveau type créé, et sera utilisé pour toute référence ultérieure à ce type. La position dans la hiérarchie est spécifiée en passant le type du conteneur qui

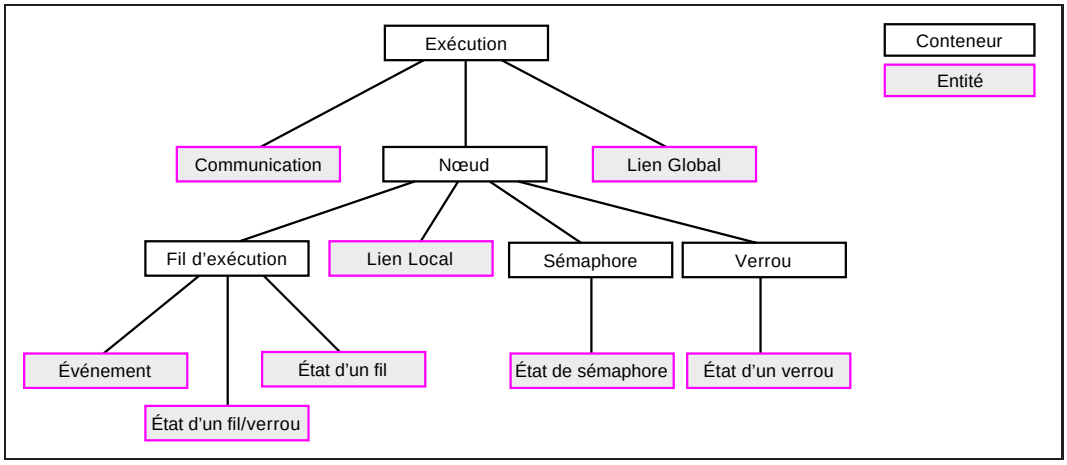

Figure 4.2 *Hiérarchie de types de conteneurs et d'entités d'*ATHAPASCAN-*0*
Reproduction de la figure 3.10

Résultat	Appel	Paramètres
CTYPE	pajeDefineUserContainerType	CTYPE NAME
ETYPE	pajeDefineUserEventType	CTYPE NAME
ETYPE	pajeDefineUserStateType	CTYPE NAME
ETYPE	pajeDefineUserLinkType	CTYPE NAME
ETYPE	pajeDefineUserVariableType	CTYPE NAME
EVALUE	pajeNewUserEntityValue	ETYPE NAME

Tableau 4.1 *Définition de types de conteneurs et entités par l'utilisateur*
Ces primitives permettent d'ajouter de nouveaux types de conteneurs et d'entités (événements, états, variables ou liens) à la structure de types du simulateur de Pajé. L'argument CTYPE *de ces primitives correspond au type de conteneur qui sera le père du nouveau type dans la hiérarchie. Il s'agit soit d'un type de conteneur créé par l'utilisateur à travers la primitive* pajeDefineUserContainerType*, soit d'une des constantes* A0_GLOBAL*,* A0_NODE *ou* A0_THREAD*, quand le type de conteneur père est un type prédéfini d'*ATHAPASCAN-*0. La dernière primitive permet de nommer les différents événements, états et liens de ces nouveaux types.*

sera le père du type créé, qui est soit un type de conteneur défini par l'utilisateur, soit dans le cas d'ATHAPASCAN-0 une des constantes `A0_GLOBAL`, `AO_NODE` ou `A0_THREAD`, qui représentent les types prédéfinis « Exécution », « Nœud » et « Fil d'exécution ».

Pour visualiser les phases de calcul de notre exemple, il faut insérer un nouveau type d'entité dans la hiérarchie de types de Pajé. Cela est illustré dans la figure 4.3, pour le contexte d'ATHAPASCAN-0.

Par exemple, l'exécution de la ligne suivante, inserée dans le programme utilisa-

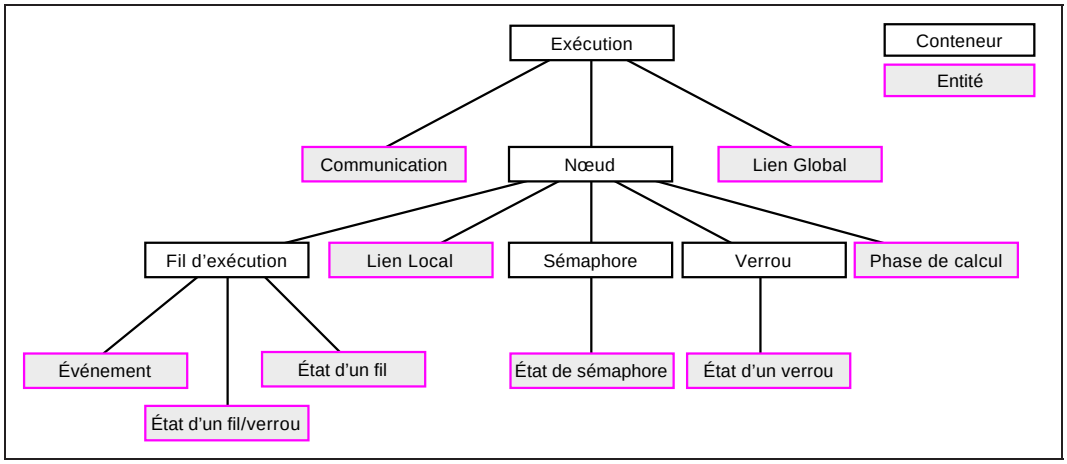

Figure 4.3 *Ajout d'un nouveau type de données à la hiérarchie d'*ATHAPASCAN*-0*
Le type « Phase de calcul » a été ajouté à la hiérarchie de types de conteneurs et entités de la figure 4.2.

teur, génère un événement dans le fichier de traces qui, interprété par le simulateur, ajoute le type d'entité « Phase de calcul » à la hiérarchie de types de conteneurs et entités de Pajé :

```
phase_state = pajeDefineUserStateType ( A0_NODE, "Phase de calcul" );
```

Le nouveau type d'entité est inséré en dessous de « Nœud » dans la hiérarchie, puisque l'objectif est de connaître la phase de calcul de chaque nœud. Cette information est fournie par le premier paramètre : la constante `A0_NODE`, pré-définie dans ATHAPASCAN-0, représente le type de conteneur « Nœud ». Les données du type « Phase de calcul » seront des états (« State »), représentés par des rectangles dans le diagramme espace-temps. La valeur de retour de cette fonction (stockée ici dans `phase_state`) sera utilisée par le programme pour faire référence au nouveau type.

4.3.2.2 Définition des valeurs des entités

La définition d'un nouveau type d'entité ne spécifie pas les valeurs que les entités de ce type pourront avoir. Dans l'exemple que nous considérons, il s'agit d'une phase d'initialisation et des phases de calculs locaux et globaux. Avant de pouvoir créer des entités du nouveau type, il faut définir ces valeurs. De la même façon que pour la définition du type, la définition des valeurs se fait à l'aide de fonctions générant des événements dans la trace. Le simulateur créera des objets pour représenter les nouvelles valeurs dans Pajé, lors de la simulation de ces événements spéciaux.

Les entités appartenant à ces nouveaux types, qui seront créées ultérieurement,

auront une valeur associée. À l'exception des entités « variables », dont la valeur représente une grandeur, les valeurs des entités doivent avoir des noms, pour faciliter leur identification lors de la visualisation. Pour réduire la taille des traces, ces noms sont identifiés par des entiers. La primitive `pajeNewUserEntityValue` permet d'associer un nom à un entier (l'entier est calculé puis retourné par la primitive). Lors de la visualisation, l'utilisateur pourra associer à chaque nom une couleur.

Supposons que dans notre exemple les phases de calcul soient « Initialisation », « Calcul local » et « Calcul global ». Ces valeurs sont définies par :

```
init_phase   = pajeNewUserEntityValue ( phase_state, "Initialisation" );
local_phase  = pajeNewUserEntityValue ( phase_state, "Calcul local" );
global_phase = pajeNewUserEntityValue ( phase_state, "Calcul global" );
```

Chaque appel définit donc une des valeurs possibles pour les entités du nouveau type (`phase_state`). Le premier argument de cette fonction est le type de l'entité dont les valeurs sont définies ; le deuxième argument est la valeur que l'on est en train de définir ; la fonction renvoie un entier qui sera utilisé pour référencer la valeur, quand des entités de ce nouveau type seront générées. La distinction entre la valeur d'une entité (par exemple, « Initialisation ») et l'entier qui lui est associé (stocké dans la variable `init_phase`) sert à diminuer la taille des traces, puisque les valeurs sont enregistrées dans la trace comme des entiers. L'utilisation d'une fonction pour générer les entiers a pour but garantir son unicité.

4.3.3 Génération des données

Pour créer les conteneurs et entités dont les types ont été définis, le programmeur doit utiliser les primitives du tableau 4.2 dans son programme. Ces primitives permettent la création et destruction de conteneurs, et la création d'entités. Il y a des primitives spécifiques pour la création d'événements, d'états (et d'états emboîtés, avec les primitives `Push` et `Pop`), de liens (chaque lien est créé par deux appels, le premier pour la source et le second pour la destination ; l'appariement est fait par le simulateur lorsque les paramètres `container`, `evalue` et `key` de la source coïncident avec ceux de la destination) et des changements de valeur d'une variable.

Dans l'exemple, un événement doit être généré chaque fois que la phase de calcul change. Cet événement sera interprété par le simulateur, qui générera l'état correspondant. Par exemple, l'appel suivant indique un passage de l'application à une phase de « Calcul local » :

```
pajeSetUserState ( phase_state, 0, local_phase, str_iter );
```

Le deuxième argument, « `0` », représente le conteneur de l'état créé (dans le cas, le « Nœud » dont la phase de calcul a changée). Le dernier argument est un commentaire, qui peut être visualisé dans Pajé. Dans l'exemple, il est utilisé pour afficher le numèro de l'itération en cours. L'algorithme du programme exemple,

Résultat	Appel	Paramètres
CONTAINER	pajeCreateUserContainer	CTYPE NAME CONTAINER
	pajeDestroyUserContainer	CONTAINER
	pajeUserEvent	ETYPE CONTAINER EVALUE COMMENT
	pajeSetUserState	ETYPE CONTAINER EVALUE COMMENT
	pajePushUserState	ETYPE CONTAINER EVALUE COMMENT
	pajePopUserState	ETYPE CONTAINER COMMENT
	pajeStartUserLink	ETYPE CONTAINER SRCCONTAINER EVALUE KEY COMMENT
	pajeEndUserLink	ETYPE CONTAINER DESTCONTAINER EVALUE KEY COMMENT
	pajeSetUserVariable	ETYPE CONTAINER VALUE COMMENT
	pajeAddUserVariable	ETYPE CONTAINER VALUE COMMENT

Tableau 4.2 *Primitives de création de conteneurs et entités par l'utilisateur*
Ces primitives sont insérées dans l'application tracée pour générer des « événements utilisateur » dont le traitement par le simulateur de Pajé ira créer des conteneurs et entités dont les types ont été définis par les primitives du tableau 4.1. L'argument « CONTAINER *» de ces primitives est soit un conteneur créé par l'utilisateur avec la primitive* pajeCreateUserContainer *soit « 0 », quand l'entité (ou conteneur) a créer appartient à un conteneur predéfini d'*ATHAPASCAN*-0 (*A0_GLOBAL, A0_NODE *ou* A0_THREAD*).*

alteré pour inclure les définitions et créations des entités « Phase de calcul » est montré dans la figure 4.4. La figure 4.5 contient deux diagrammes espace-temps du programme, avant et après l'ajout de ces entités.

```
unsigned phase_state, init_phase, local_phase, global_phase;

phase_state  = pajeDefineUserStateType ( A0_NODE, "Phase de calcul" );
init_phase   = pajeNewUserEntityValue ( phase_state, "Initialisation" );
local_phase  = pajeNewUserEntityValue ( phase_state, "Calcul local" );
global_phase = pajeNewUserEntityValue ( phase_state, "Calcul global" );

pajeSetUserState ( phase_state, 0, init_phase, "" );
initialisation();
while (!converge) {
    iter++;
    str_iter = itoa (iter);
    pajeSetUserState ( phase_state, 0, local_phase, str_iter );
    calcul_local();
    send (local_data);
    receive (remote_data);
    pajeSetUserState ( phase_state, 0, global_phase, str_iter );
    calcul_global();
}
```

Figure 4.4 *Algorithme simplifié du programme exemple*

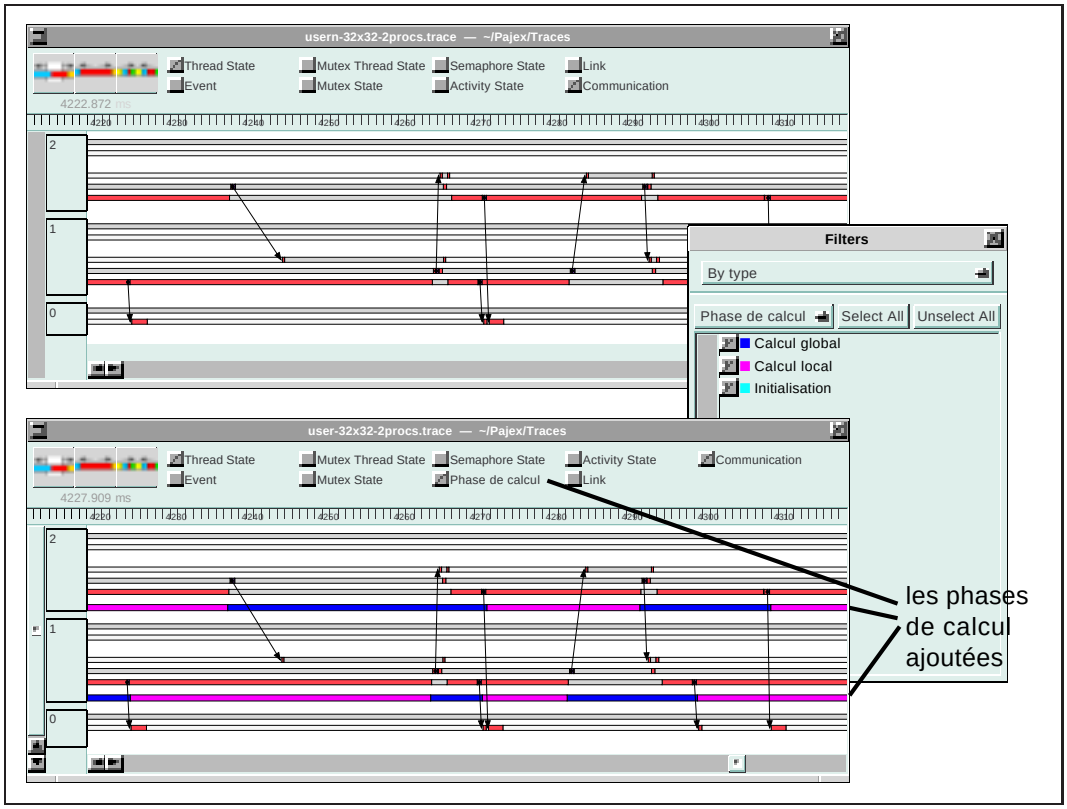

Figure 4.5 *Visualisation du programme exemple*
La deuxième figure montre les entités « Phase de calcul » définies par l'utilisateur

4.3.4 Exemple : visualisation d'ATHAPASCAN-1

La généricité offerte par Pajé a été utilisée pour instancier un outil de visualisation pour le modèle de programmation ATHAPASCAN-1, sans avoir à développer de nouveaux composants pour Pajé. ATHAPASCAN-1 [24, 11, 21, 10, 23] est une bibliothèque de programmation parallèle de haut niveau développée au sein du projet APACHE. Cette bibliothèque utilise ATHAPASCAN-0 comme support exécutif. ATHAPASCAN-1 permet une description explicite et dynamique du parallélisme par création asynchrone de tâches parallèles. Les synchronisations entre ces tâches sont implicites, chaque tâche déclarant les accès qui seront effectués sur une mémoire virtuelle partagée. Une tâche ne peut être exécutée que si toutes les données accédées en lecture sont prêtes (intuitivement, les lecteurs attendent les écrivains qui les précèdent). L'ensemble de ces tâches, des données partagées et des accès est structuré en un graphe construit de manière dynamique et distribuée.

ATHAPASCAN-1 est constitué de trois modules, répliqués dans chaque nœud participant à l'exécution : un module de gestion du graphe liant les tâches et les

données partagées, un module d'ordonnancement, chargé d'affecter un site d'exécution à chaque tâche et un module d'exécution, chargé d'exécuter les tâches qui lui sont confiées.

Cette structure modulaire séparant entièrement l'ordonnancement des autre parties d'ATHAPASCAN-1 permet l'écriture aisée de différentes stratégies d'ordonnancement et de placement des tâches. Le module d'ordonnancement a un libre accès au graphe et est averti de chacune de ses modifications (nouvelle tâche, tâche prête, etc.). Pour équilibrer la charge de calcul entre nœuds, il peut décider de migrer les tâches d'un nœud à un autre. Les données partagées seront migrées par le système, pour permettre aux tâches de les accéder localement lors de leurs exécutions.

L'exécution des tâches est assurée par un groupe de processeurs virtuels implanté sous forme de fils d'exécution ATHAPASCAN-0. Chaque processeur virtuel est soit en attente de travail, soit en train de contacter le module d'ordonnancement pour en trouver, soit en train d'exécuter le corps d'une tâche. Dans ce dernier cas, le module de gestion du graphe d'ATHAPASCAN-1 est activé quand la tâche en train d'exécuter crée une nouvelle tâche et quand elle accède des données partagées.

Nous avons tracé ATHAPASCAN-1 de façon à pouvoir visualiser :

- le comportement des processeurs virtuels ;
- le comportement d'une politique particulière d'ordonnancement ;
- les migrations des données.

La visualisation d'un processeur virtuel a pour but de savoir quelle partie du code il est en train d'exécuter à chaque instant. Cette partie peut être : aucune (il est bloqué en attendant qu'une tâche soit prête), le module d'ordonnancement, le module d'exécution, le module de gestion du graphe ou le corps d'une tâche. Nous avons donc créé un nouveau type d'entité de visualisation pour représenter la partie du code exécutée, appelé « état A1 », associé à chaque fil d'exécution. Ce nouveau type d'entité et les valeurs qu'il peut avoir sont définis à l'aide de méta-événements générés par le code montré dans la figure 4.6. La figure 4.7 montre un exemple de génération d'un événement pour changer l'état d'un processeur virtuel à l'état correspondant à l'exécution du corps de la tâche.

```
a1_state = pajeDefineUserStateType( A0_THREAD, "A1 State");
a1_state_blocked = pajeNewEntityValue( a1_state, "Blocked" );
a1_state_sched   = pajeNewEntityValue( a1_state, "Scheduler" );
a1_state_exec    = pajeNewEntityValue( a1_state, "Exec" );
a1_state_graph   = pajeNewEntityValue( a1_state, "Graph" );
a1_state_task    = pajeNewEntityValue( a1_state, "Task" );
```

Figure 4.6 *Définition des états des processeurs virtuels* ATHAPASCAN-*1*
Code qui crée un nouveau type d'état associé aux fils d'exécution et définit son espace de valeurs.

```
pajeSetUserState( a1_state, 0, a1_state_task, task_id );
```

Figure 4.7 *Changement de l'état d'un processeur virtuel* ATHAPASCAN-*1*
Cette ligne génère un événement qui représente un changement d'état d'un processeur virtuel. Son deuxième argument représente le conteneur de l'entité à générer. Dans ce cas, comme l'entité a été définie comme appartenant au conteneur de type prédéfini `A0_THREAD`*, qui correspond au fil d'exécution courant, il n'a pas besoin d'être indiqué. Cet argument n'a de sens que pour des conteneurs définis par l'utilisateur. Le quatrième argument est un commentaire et nous avons mis l'identification de la tâche courante (*`task_id`*), pour pouvoir la distinguer lors de la visualisation.*

La politique d'ordonnancement tracée est informée des changements d'état des tâches, et connaît donc les tâches qui sont prêtes à être exécutées et celles que ne le sont pas. Quand l'ordonnanceur reçoit une demande de tâche à exécuter, il en choisit une, parmi les tâches prêtes. S'il n'y a pas de tâche prête dans sur le nœud local, il demande des tâches à un autre nœud, choisi au hasard. Nous voulions suivre l'évolution des nombres de tâches prêtes et non-prêtes dans chaque nœud et les transitions des tâches (création, changement d'état, migration, exécution). Nous avons alors défini un type de conteneur « Compteur », appartenant à « Nœud », pour contenir les compteurs de tâches prêtes et non prêtes, ainsi que des types d'entités « Quantité », une variable pour représenter la quantité de tâches dans un compteur à chaque instant et « Transition de tâche », liens pour représenter les changement d'état des tâches. Pour visualiser les migrations des données, nous avons défini un autre type de lien, « Transition de donnée ».

Lors de la simulation, la hiérarchie de types de conteneurs et entités est enrichie de ces nouveaux types, et se présente comme dans la figure 4.8. Un exemple de hiérarchie de conteneurs et entités lors d'une exécution, sur un nœud et avec deux fils d'exécution, est montré dans la figure 4.9.

Le diagramme espace-temps correspondant à une telle exécution apparaît dans la figure 4.10. Cette figure montre seulement les nouvelles entités. Il est également possible de visualiser les entités ATHAPASCAN-1 en même temps que les entités ATHAPASCAN-0, en sélectionnant les types d'entités à visualiser, en haut de la fenêtre. Dans la figure 4.11, les états ATHAPASCAN-0 des fils d'exécution associés aux processeurs virtuels sont aussi montrés, à coté des états ATHAPASCAN-1. Les états ATHAPASCAN-0 montrent notamment les blocages des fils d'exécution dûs aux verrous d'exclusion mutuelle utilisés dans l'implémentation d'ATHAPASCAN-1 pour gérer les conflits aux structures communes (par exemple, les compteurs de tâches, la partie locale du graphe, etc.).

Ces traces on été générées par inclusion de quelques appels aux primitives de définition et génération de traces dans ATHAPASCAN-1. Aucun changement n'a été

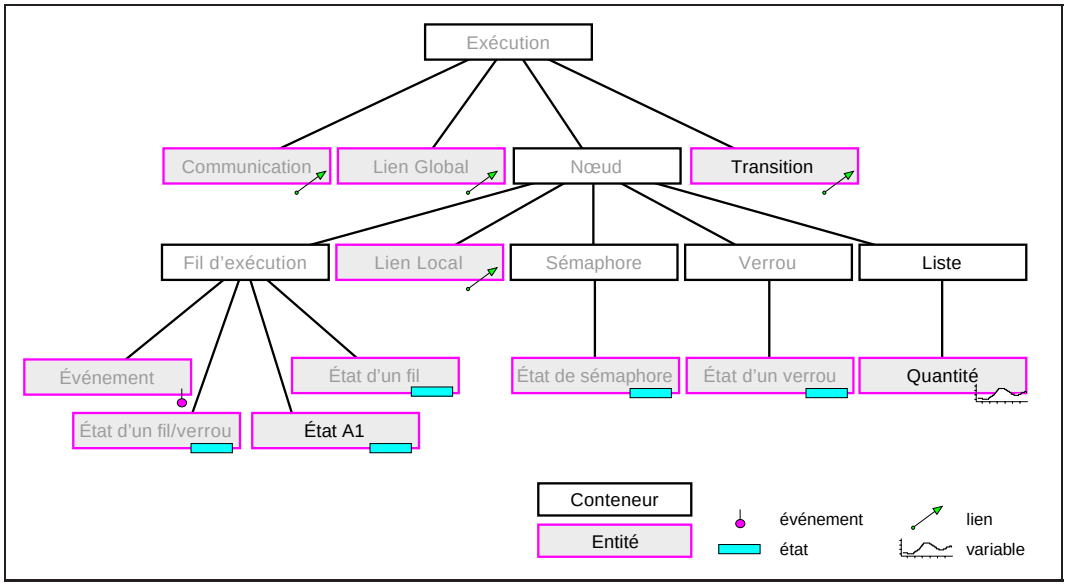

Figure 4.8 *Types de conteneurs et d'entités de l'ordonnanceur* ATHAPASCAN-*1*
*Les types ajoutés à la hiérarchie d'*ATHAPASCAN-*0 ont leurs noms écrits en noir.*

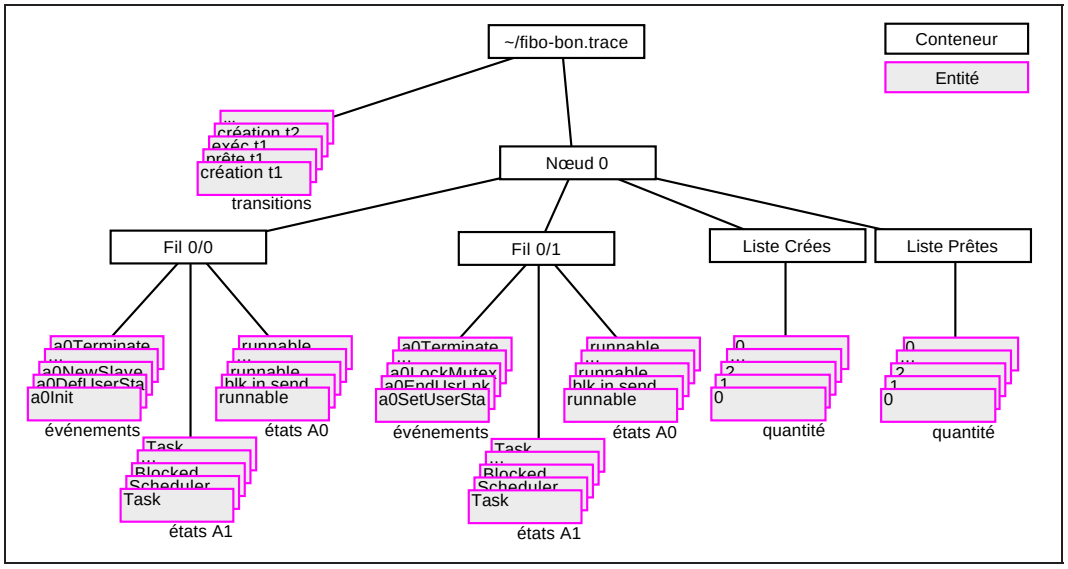

Figure 4.9 *Hiérarchie représentant une exécution* ATHAPASCAN-*1*

nécessaire dans Pajé.

Figure 4.10 *Diagramme espace-temps d'une exécution* ATHAPASCAN-*1*
Les trois traits d'en haut représentent les états des trois processeurs virtuels alloués par ATHAPASCAN-*1 pour l'exécution des tâches. Deux états sont représentés : le plus clair quand que le processeur virtuel exécute une tâche, le plus foncé quand il exécute l'ordonnanceur. Un état foncé inclus dans un état clair montre le déclenchement de l'ordonnanceur par une tâche (quand elle en crée une autre) ; l'état foncé est en dehors de l'état clair quand l'ordonnanceur est déclenché pour chercher une nouvelle tâche à exécuter. Dans les intervalles où aucun état n'est représenté, le processeur virtuel est en train d'exécuter d'autres parties d'*ATHAPASCAN-*1. Les deux traits d'en bas montrent les deux files de tâches, non-prêtes et prêtes ; la quantité de tâches dans une file est représentée par un dégradé de couleur. Les flèches montrent les déplacements subis par une tâche : création, changement d'état, exécution.*

4.3.5 Évaluation

L'existence d'un traceur/simulateur générique augmente beaucoup les capacités d'extensibilité d'un environnement de visualisation. L'outil peut être adapté, sans changement, à la visualisation d'une gamme très variée de données, spécifiques aux applications. Un traceur/simulateur générique peut aussi être utilisé pour la génération rapide de prototypes de traceurs et simulateurs spécifiques.

Il y a, hélas, un prix à payer, par rapport à un simulateur spécifique : la quantité de données générées dans la trace d'exécution est en général plus importante avec un simulateur générique. Par exemple, dans le traçage de l'ordonnanceur d'ATHAPASCAN-1 (section 4.3.4), lors de la migration d'une tâche d'un nœud à l'autre, quatre événements sont générés par le traceur générique : un pour décrémenter le nombre de tâches du nœud émetteur, un pour incrémenter le nombre de tâches du nœud destinataire et deux pour créer le lien qui représente la migration de la tâche.

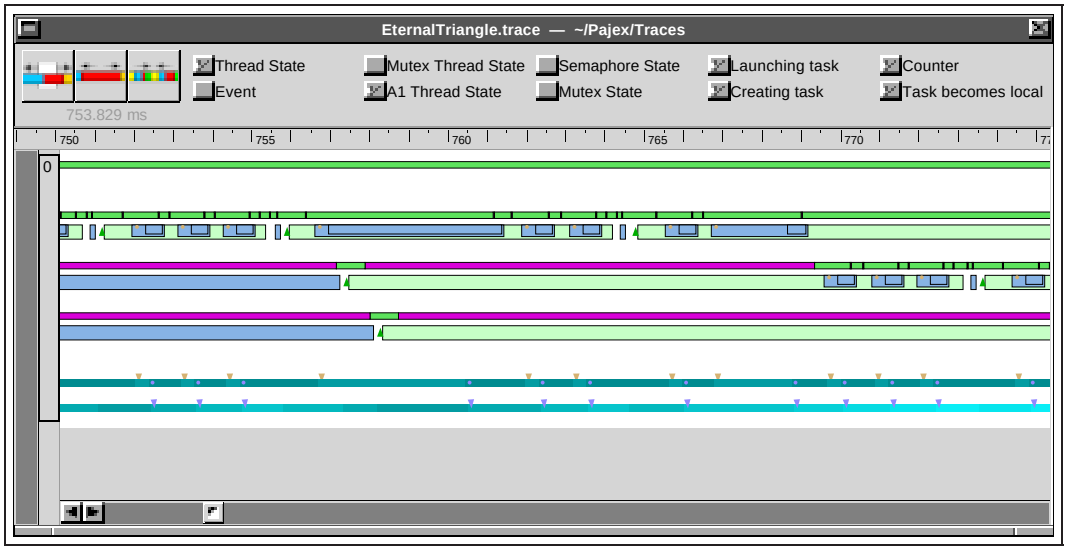

Figure 4.11 *Diagramme espace-temps d'une exécution* ATHAPASCAN-*1 avec les états* ATHAPASCAN-*0*
La même exécution que celle montrée dans la figure 4.10, avec les états des fils d'exécution ATHAPASCAN-*0 en plus (les traits plus fins au-dessus de chaque processeur virtuel* ATHAPASCAN-*1). Les états plus clairs montrent les instants où les fils d'exécution sont actifs, tandis que les états plus foncés correspondent aux instants où ces fils d'exécution attendent la libération d'un verrou d'exclusion mutuelle.*

Avec un traceur et un simulateur spécifiques, deux événements suffiraient, un pour l'émission de la tâche et un autre pour sa réception. Le simulateur connaîtrait la sémantique de ces événements et produirait les entités correspondantes.

Ce surcoût pourrait être réduit en augmentant l'expressivité de la description des événements, de façon à pouvoir décrire des événements qui produiraient plus d'une entité. Cela aurait néanmoins l'inconvénient de compliquer le simulateur, et bien pire, compliquer la tâche du programmeur lors de la génération de la description des événements.

Le simulateur générique de Pajé a d'autres limitations :

- Les nouveaux types de données ajoutés doivent forcément être organisés hiérarchiquement. Pajé est basé sur la structure hiérarchique des données, mais pas forcément la hiérarchie utilisée par ATHAPASCAN-0 : il n'y a pas de dépendences dans l'existence de nœuds, fils d'exécution, etc. On peut cependant remarquer qu'il est possible de définir une hiérarchie entièrement nouvelle, indépendante de la hiérarchie connue par le simulateur (ici celle d'ATHAPASCAN-0), en utilisant les primitives existantes.
- Les entités ajoutées sont des événements, des états, des liens ou des variables.

L'addition d'une nouvelle forme d'entité nécessite altérations dans Pajé (comparables aux altérations nécessaires dans d'autres environnements).

4.4 Conclusion

La contrainte d'extensibilité a des conséquences importantes sur l'architecture d'un environnement de visualisation. Ces contrainte a été prise en compte à plusieurs niveaux dans l'environnement Pajé. Tout d'abord, l'environnement est structuré de façon modulaire, comme un graphe acyclique de composants interconnectés. Il est ainsi possible de constituer un outil particulier, en constituant un graphe particulier, dont les nœuds sont les composants fournis par l'environnement. Une autre forme de possibilité d'adaptation réside dans la possibilité de configurer les composants, conformément aux préférences exprimées par l'utilisateur et enregistrées par l'outil. Afin de permettre d'ajouter simplement de nouvelles fonctionnalités à l'environnement, des protocoles de communication entre composants ont été définis systématiquement. Enfin, la possibilité d'extension la plus puissante offerte aux utilisateurs leur permet d'adapter l'environnement au traitement de données nouvelles, telles que celles résultant du traçage de programmes parallèles utilisant un autre modèle de programmation que ceux traités jusque là, sans avoir à modifier l'outil de visualisation et donc sans avoir à développer de composant spécifique. Cette possibilité d'extension est basée sur la généricité du simulateur, l'instanciation d'un type de données nouvelles se faisant en utilisant des commandes transmises avec les traces. Cette possibilité d'extension a été testée pour visualiser des traces d'exécution de programmes ATHAPASCAN-1, dont le modèle de programmation diffère du modèle de programmation offert par ATHAPASCAN-0 et qui constituait le modèle de référence de traité par Pajé. À notre connaissance, aucun environnement de visualisation existant n'offre autant de possibilités d'extensibilité. En particulier, la généricité du simulateur ne semble pas avoir été proposée auparavant.

5

Interactivité

Il serait très intéressant de disposer d'un outil susceptible d'analyser automatiquement l'exécution d'un programme et d'informer son programmeur d'une éventuelle sous-utilisation des ressources de la machine et en plus de lui en donner les raisons ainsi que des suggestions pour résoudre ce problème. Malheureusement, on ne connaît pas d'outil de ce genre. Notre outil a un objectif plus modeste : il s'agit de montrer au programmeur l'exécution de son programme et de lui offrir des informations susceptibles de l'aider à comprendre plus facilement le comportement de son programme pour qu'il trouve de lui-même les moyens d'améliorer son efficacité.

Les principaux problèmes que posent la conception d'un outil de visualisation proviennent de l'énorme quantité d'information produite pendant l'exécution d'un programme et des difficultés à la gérer et à la montrer au programmeur, de façon à ce qu'il puisse en extraire quelque chose d'utile à la mise au point ou l'amélioration de son programme. L'outil de visualisation peut être comparé à un « navire utilisé par le programmeur pour naviguer sur cet océan d'informations », et doit comme tel être le plus confortable possible pour qu'il arrive indemne à son but : un programme libre d'erreurs et qui s'exécute aussi rapidement que possible. La capacité de l'outil à interagir avec son utilisateur (plutôt réagir convenablement à ses actions) est ce qui va donner à ce dernier les moyens d'entreprendre ce voyage.

Comme le volume d'informations est trop important, il n'est pas possible de les montrer toutes au même moment, et même si cela était possible, il serait humainement impossible de les comprendre. En donnant à l'utilisateur une vision simplifiée (abstraite, synthétique) des données et en lui permettant, d'une façon simple et intuitive, d'obtenir plus de détails là où il lui semble qu'il y ait des problèmes, l'outil

augmente le pouvoir de l'utilisateur. « Obtenir plus de détails » peut être l'explosion (« zooming ») d'une vision synthétique vers une vision plus détaillée ou simplement l'obtention des données brutes qui ont servi à la construction de cette vision synthétique ou encore des données qui existent mais qui n'ont pas été utilisées ou n'ont pas de rapport avec la visualisation.

Quand le niveau de détail requis par l'utilisateur est important, l'outil doit facilement mettre en évidence les liaisons entre les différentes données qu'il présente à l'utilisateur, pour aider à la compréhension du tout. Cette mise en évidence de données liées est aussi très utile quand la même information apparaît dans des visualisations multiples, pour l'identification d'une même donnée sur des points de vue différents. Une autre utilisation de la liaison entre visualisations distinctes est le partage des actions pertinentes de l'utilisateur, par exemple, la sélection d'un intervalle de temps dans une visualisation est utilisée pour cibler un module qui calcule des statistiques.

Parfois il n'est pas très évident de faire la correspondance entre une visualisation du comportement d'un programme et le code source du même programme ; le niveau d'abstraction de la programmation et de la visualisation n'est souvent pas le même. L'outil doit faciliter la tâche de trouver cette correspondance, de préférence dans les deux sens.

Ces interactions entre l'utilisateur et l'outil peuvent être explicites ou implicites. Une interaction explicite correspond à la réaction de l'outil à une commande directe de l'utilisateur, (généralement à une sélection par la souris). Dans une interaction implicite, l'outil affiche des information en réaction à des actions plus simples de l'utilisateur, comme le déplacement du pointeur sur la réprésentation graphique d'un objet, qui cause l'affichage d'informations succinctes à propos de l'objet. Un bon équilibre entre ces deux types d'interactions est important pour le confort d'utilisation de l'outil. Une action explicite a généralement besoin d'une nouvelle fenêtre pour afficher les informations demandées, ce qui peut être encombrant en termes d'utilisation de l'espace écran ; de plus, on exige des efforts supplémentaires de l'utilisateur. Trop d'informations implicites risquent d'être visuellement fatigantes, ou peuvent rendre l'outil inutilisable en raison de la lenteur causée par la recherche/affichage d'informations inutiles. Par ailleurs, certaines informations sont plus facilement montrées de façon implicite, comme la relation entre deux objets, qui peut être montrée par surbrillance de la représentation d'un des objets quand le pointeur est sur l'autre. D'autres informations, telles que la description textuelle d'une donnée, sont plus appropriées à une interaction explicite, qui cause l'affichage des information dans une fenêtre graphique à part.

Une information peut être représentée graphiquement de plusieurs façons possibles (forme, couleur, texture, taille, position, texte, etc). La correspondance entre la forme graphique d'une information et ce qu'elle représente doit être naturelle,

changeable et facilement rappelable.

Dans un environnement interactif, l'utilisateur doit pouvoir se déplacer librement dans les visualisations générées à partir de la trace (cela correspond très souvent à se déplacer dans le « temps de l'exécution »). L'ordre suivant lequel un utilisateur analyse une trace d'exécution ne correspond pas forcement à l'ordre de l'exécution. Il est normal d'identifier les problèmes de performances à un niveau d'abstraction élevé puis de rechercher leurs causes à un niveau d'abstraction plus détaillé — ce qui est possible grâces aux techniques de « scalabilité » décrites dans le chapitre 6 —, en analysant les événements qui se sont produits avant l'apparition de chaque problème. Or, pour que cette forme d'analyse soit possible, l'outil doit permettre le déplacement en arrière dans le temps. Le déplacement libre dans les données est incompatible avec la circulation de données dans un diagramme flot de données pur. La solution retenue pour combiner l'interactivité avec l'extensibilité — qui motive la structuration de Pajé en composants d'un graphe flot de données — est de casser le côté flot de données pur du graphe de composants. Ce problème et la solution qui lui a été apportée — par la définition du composant « Encapsuleur » qui regroupe les données dans la structure « Fenêtre d'observation » et centralise les accès interactifs à ces données — ont déjá été présentés au paragraphe 3.4.2 et ne seront pas abordés dans ce chapitre.

Nous nous intéressons dans ce chapitre au problème du temps de réponse de Pajé aux sollicitations de l'utilisateur. Pour permettre un niveau d'interactivité satisfaisant, il faut que l'outil accède rapidement aux données concernées. Une session de visualisation est constituée d'un grand nombre d'opérations interactives. La quantité de données nécessaire à une opération interactive ne représente qu'une infime partie de la quantité totale de données existante stockée dans la fenêtre d'observation. Il faut donc organiser ces données de façon à permettre une sélection rapide des données pertinentes.

Le cœur de ce chapitre est donc constitué par la description de l'organisation des données dans la fenêtre d'observation : plusieurs organisations possibles sont décrites et pour chacune le coût mémoire et la complexité des algorithmes de recherche d'informations sont évalués. Cette description utilise une typologie des objets et des types d'accès aux objets qui est décrite au préalable. Le chapitre se conclut montrant les fonctionnalités de Pajé liées à l'interactivité qui ont été rendues possibles par les choix présentés au préalable.

5.1 Terminologie

Les objets stockés dans la fenêtre d'observation sont les entités et conteneurs qui ont été définis dans la section 3.4.2.2. Pour la bonne intelligence de ce qui suit, une

classification plus fine des entités et des types d'accès à ces entités est nécessaire.

5.1.1 Classification des entités

Nous avons classifié les entités en fonction de leurs dates de début et de fin. Cette classification nous permettra de mieux analyser et optimiser l'accès aux entités et leur organisation dans la mémoire (on reparlera de cette classification dans la section 5.2). Dans Pajé, les entités sont organisés hiérarchiquement, dans des conteneurs (cf. section 3.4.2.2).

Les événements sont des entités instantanées, c'est à dire qu'elles ne sont associées qu'à une seule valeur de temps et représentent des actions atomiques qui ont eu lieu dans le programme tracé. D'autres entités ne sont pas instantanées, et représentent des actions non atomiques du programme, comme les communications ou les états des fils d'exécution. Nous avons classifié les entités qui ne sont pas instantanées en :

- Entités consécutives : une entité ne commence pas avant la fin de l'entité précédente ; par exemple, les états d'un fil d'exécution, où le fil d'exécution est dans un et un seul des états possibles à chaque instant.
- Entités « emboîtées », contenues les unes dans les autres, telles que les entités qui représentent une fonction exécutée par le programme : lorsqu'une fonction se termine, l'exécution continue dans la fonction appelante. La figure 5.1 montre des entités emboîtées qui représentent un arbre d'appel de fonctions.
- Entités n'ayant pas de relation temporelle les unes avec les autres, dans un même conteneur. Le meilleur exemple d'entité de ce type sont les communications entre deux paires indépendants de correspondants, dans ATHAPASCAN-0. Ces deux communications n'ont pas forcément des relations temporelles les unes avec les autres, mais elles possèdent le même conteneur, puisque dans la hiérarchie de types d'ATHAPASCAN-0 (voir figure 3.10) les communications sont globales.

5.1.2 Types d'accès aux entités

Pendant une session de visualisation, plusieurs modules de l'environnement de visualisation doivent accéder aux entités pour en obtenir les informations dont ils ont besoin. Selon la fonctionnalité du module ou son mode de fonctionnement, cet accès peut se réaliser de différentes façons et à différentes fréquences. L'organisation de ces données et les algorithmes utilisés pour y accéder jouent un rôle fondamental dans la capacité de l'outil à être réactif aux actions de l'utilisateur avec un temps de réponse compatible avec l'interactivité. On analysera les types d'accès

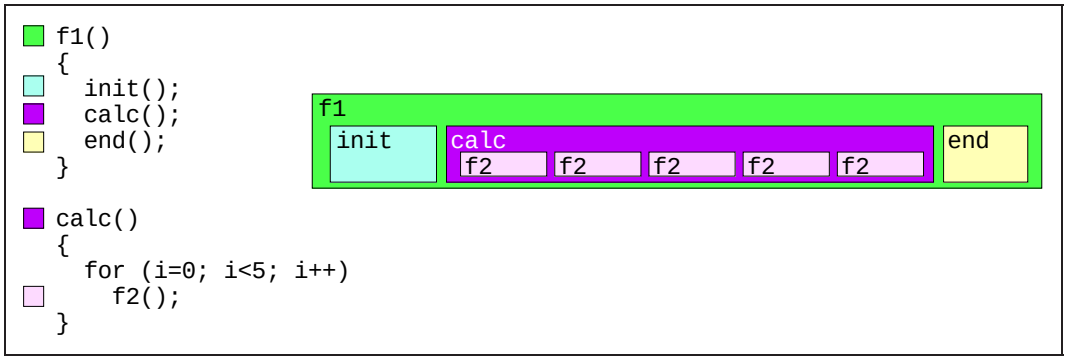

Figure 5.1 *Entités emboîtées*
La hauteur d'un rectangle représente le niveau d'emboîtement ; la couleur représente la fonction exécutée.

liés à l'interactivité pour en déduire une organisation des données pour optimiser ces accès. Les types d'accès possibles sont :

- Addition d'entités, quand les données de la trace sont depuis un fichier pour être stockées en mémoire.
- Suppression d'entités, quand il faut libérer une partie de la mémoire pour pouvoir lire davantage de données.
- Recherche des entités à afficher, quand l'utilisateur déplace la région d'affichage sur l'écran.
- Filtrage, quand l'outil doit produire des données synthétiques à partir de données plus détaillés.
- Accès unique par des modules qui n'ont besoin d'accéder qu'une seule fois à chaque entité et ne reviennent jamais en arrière.
- Interactivité : réaction de l'outil aux actions de l'utilisateur, comme la surbrillance de l'objet sous pointeur.

De ces types d'accès, ceux qui ont la relation la plus directe avec l'utilisateur sont les plus importants du point de vue de l'interactivité et sont ceux qui donnent l'impression de réactivité—utilisation plus agréable de l'outil. Ces sont les accès nécessaires à l'interactivité et à l'affichage.

Accès liés à l'interactivité. Dans ce cas, les entités auxquelles on veut accéder sont celles que l'utilisateur peut visualiser et sur lesquelles il peut interagir. L'interaction est ponctuelle physiquement sur l'écran : à chaque instant, l'utilisateur peut interagir avec une entité, celle qui est sous le pointeur. Il faut une façon d'associer rapidement une coordonnée sur l'écran avec l'entité représentée à cette position.

La sélection d'une entité par l'utilisateur peut être active, quand cette entité est sélectionnée explicitement afin d'obtenir de l'outil plus d'informations, ou passive, quand l'outil donne des informations supplémentaires sur l'entité qui est au-dessous du pointeur. Dans le deuxième cas, chaque fois que l'utilisateur bouge la souris il faut trouver l'entité associée. Comme l'action de bouger la souris est loin d'être rare, les accès de ce type sont très fréquents.

Sur une visualisation temporelle comme un diagramme espace-temps, une des dimensions est le temps ; l'autre dimension se traduit plus au moins facilement par un ou plusieurs types d'entités. Pour accéder à l'entité avec laquelle on veut interagir il faut donc trouver l'entité d'un certain type qui se trouve à l'intérieur d'un intervalle de temps qui correspond à la largeur d'un pixel.

Accès liés à l'affichage Dans une visualisation temporelle, seule une partie de la trace est habituellement affichée, limitée dans le temps (on visualise un intervalle de temps limité) et dans l'« espace » (on visualise certains types d'entités, appartenant à certains « conteneurs »). Pour réaliser un affichage, il faut donc accéder à toutes les entités de tous les types concernés dans l'intervalle de temps considéré.

Ces deux types d'accès sont très semblables et correspondent à une demande de toutes les entités d'un certain type appartenant à un même conteneur dans un intervalle temporel. Par exemple, tous les événements du fil d'exécution 0 du nœud 1 entre 0,2 et 0,5 secondes d'exécution. Des accès de ce type sont réalisés fréquemment durant une session de visualisation, à chaque fois que le pointeur de la souris est bougé et à chaque fois que l'utilisateur change les données à afficher (en changeant l'intervalle de temps voulu ou en changeant les paramètres de filtrage des données).

5.2 Organisation des données dans la fenêtre d'observation

La trace d'exécution d'un programme utilise de grands fichiers. Pour pouvoir être analysée, cette trace doit être lue en mémoire. La simulation de cette trace produit un grand nombre d'entités (états, communications, synchronisations, etc.) ainsi que les divers liens entre ces entités nécessaires à la gestion de l'interaction avec l'utilisateur et à la construction de visualisations synthétiques. L'ensemble des données produites par la simulation d'une trace prend donc beaucoup plus de place en mémoire que la taille du fichier de trace. Cette différence s'accroît suivant le niveau de détail du simulateur et suivant le niveau d'interaction permis par l'outil,

car plus la simulation est détaillée, plus elle va créer d'entités ; de la même façon, plus on dispose de possibilités d'interactions, plus il faut de liens entre les entités.

La gestion mémoire doit résoudre deux problèmes : d'une part la gestion du très grand volume de données produit par la simulation — afin d'éviter un effondrement du mécanisme de pagination de la machine qui exécute l'outil de visualisation — et d'autre part l'accès efficace aux entités — requis par l'interactivité de l'outil. Le premier point est résolu en limitant les données stockées en mémoire à celles qui correspondent à une fenêtre d'observation bornée par deux dates (voir section 3.4.2). La fenêtre d'observation « glisse » dans le temps, contrôlée par l'utilisateur, quand il déplace une visualisation temporelle en dehors des limites courantes de la fenêtre. La largeur de la fenêtre est un paramètre de l'environnement. À intervalle régulier, l'état du simulateur est sauvegardé afin de permettre un retour en arrière en deçà des limites de la fenêtre courante. Le mécanisme de déplacement de la fenêtre d'observation est détaillé dans la section 3.4.2.5.

Le second problème, permettre un accès rapide aux données est résolu par une organisation sophistiquée des données contenues dans la fenêtre de visualisation. Dans les sections qui suivent, nous présentons l'organisation générale de la fenêtre d'observation, suivie de la discussion de plusieurs structurations des entités, en distinguant les entités instantanées — qui ne nécessitent pas de structuration complexe pour être accessibles efficacement — et les entités non instantanées — qui nécessitent une organisation sophistiquée pour que leur temps d'accès reste compatible avec les contraintes d'interactivité.

5.2.1 Structure générale

Dans la section 5.1.2, nous avons vu que l'accès le plus courant aux données imposé par l'interactivité est la recherche de toutes les entités d'un certain type appartenant à un même conteneur dans un intervalle temporel donné. Nous avons donc séparé le espace d'entités à rechercher par type et par conteneur. La structure utilisé est représentée dans la figure 5.2. À partir d'un type d'entité et d'un conteneur, nous arrivons à une structure de données qui contient les entités de ce type appartenant à ce conteneur. Cette dernière structure doit être optimisée pour permettre l'accès à ses données, contenues dans un intervalle de temps.

On ne connaît pas d'avance les types d'entités (les types varient selon le simulateur, et on peut avoir des types d'entités définis par le programmeur) ni les conteneurs (ils varient d'une exécution à une autre, et peuvent être créés dynamiquement par le programme). En plus, on ne dispose pas toujours d'une façon directe de traduire une identification d'un type ou d'un conteneur vers une position dans un tableau simple. Pour ces raisons, une table de hachage de taille variable nous paraît

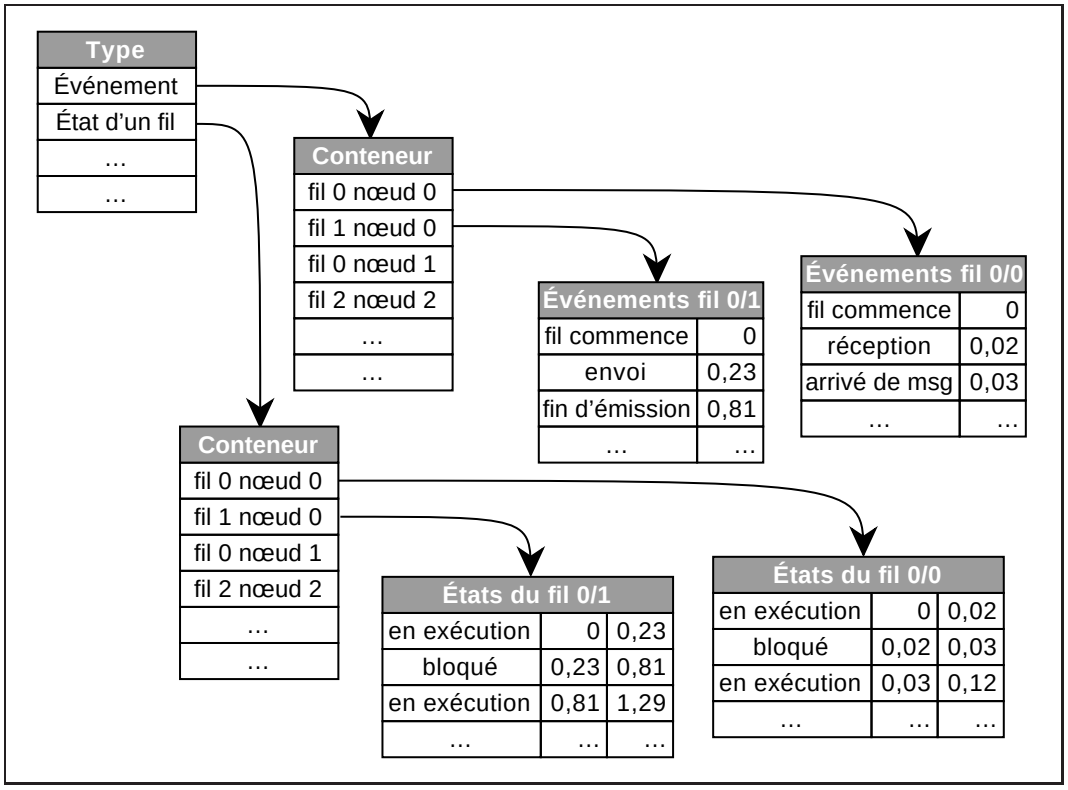

Figure 5.2 *Structure interne de la fenêtre d'observation*
Les types et les conteneurs sont stockés dans des tables de hachage ; les entités sont stockées dans des structures de données différentes, selon leur classification (instantanées ou non instantanées)

la structure de données la plus adaptée pour stocker les types d'entités et les conteneurs et les accéder efficacement. En plus, le temps d'accès à un élément d'une table de hachage est indépendant du nombre d'éléments dans la table.

Pour tout accès aux données on doit donc d'abord accéder deux tables de hachage, une pour le type d'entité, une autre pour le conteneur. Cela permet de diminuer l'espace de recherche des structures qui contiennent les entités.

Complexité en mémoire de la structure générale On a besoin d'une table de hachage pour les types, et d'une table de hachage pour chaque type, pour les conteneurs : $k_h N_\mathrm{t} + N_\mathrm{t} \times k_h N_\mathrm{c}$ où N_t est le nombre de types, N_c est le nombre de conteneurs et k_h est le surcoût mémoire pour une table de hachage. La complexité mémoire de la structure générale de la fenêtre d'observation est donc $\mathcal{O}(N_\mathrm{t} \times N_\mathrm{c})$

Complexité en temps de la structure générale Le temps nécessaire pour accéder à la structure des entités correspond à la durée d'un accès à la table de hachage des types et un accès à la table de hachage des conteneurs. Comme le temps d'accès à une table de hachage est constant, la complexité en temps est : $\mathcal{O}(1)$.

À l'espace et au temps nécessaires pour la structure générale de la fenêtre d'observation, on doit encore ajouter l'espace et le temps nécessaires pour stocker et accéder aux entités, qui sera discuté dans les sections qui suivent. Nous utilisons deux structures différentes pour l'accès aux entités, une pour les entités instantanées, plus simple, et la deuxième pour les entités non instantanées, qui nécessitent une structure plus complexe.

5.2.2 Recherche des entités instantanées

Les entités instantanées sont celles qui sont associées à une seule date. Donc, pour identifier les entités instantanées qui se situent dans un intervalle temporel, il suffit de savoir si la date associée se trouve dans cet intervalle. On recherche donc $e_i | t_1 < t^i < t_2$, où t_1 et t_2 sont les limites inférieure et supérieure de l'intervalle et t^i est la date de l'entité e_i. Considérant les données de la figure 5.2, la requête *« tous les événements du fil d'exécution 0 du nœud 1 entre 0,2 et 0,5 secondes d'exécution »* retournerait l'événement envoi | 0,23.

Organisant les entités dans un tableau et en les triant par date, on arrive efficacement à trouver les entités qui appartiennent à l'intervalle, il suffit de trouver la première par recherche dichotomique puis de parcourir le tableau jusqu'à ce qu'on trouve une entité qui n'est pas dans l'intervalle.

Complexité en mémoire L'espace mémoire nécessaire pour stocker les N entités dans un tableau correspond à l'espace occupé par ces entités auquel s'ajoute le surcoût dû au tableau (nombre d'éléments, pointeur vers la zone mémoire où sont les éléments). La consommation mémoire est donc : $k_0 + Nk_1 \rightarrow \mathcal{O}(N)$, où k_0 est le surcoût dû au tableau, k_1 est l'espace moyen nécessaire pour chaque élément et N est le nombre d'éléments à stocker.

Complexité en temps La durée nécessaire pour accéder les N_i entités contenues dans l'intervalle de temps considéré comporte donc :

- Le nombre de comparaisons nécessaires pour trouver la première entité dans le tableau, par recherche dichotomique : $\log_2 N$

- Le parcours des entités de l'intervalle voulu. Il s'agit d'un parcours simple, il suffit de tester chaque entité, dans l'ordre du tableau. Si $t^i < t_2$, l'entité est dans l'intervalle ; le cas contraire, le parcours est terminé. $\mathcal{O}(N_i)$

La complexité en temps est donc : $\mathcal{O}(\log N + N_i)$. Si on considère le cas où $N_i \ll N$ (c'est le cas pour la recherche, très fréquente, des entités sous le pointeur, où l'intervalle de temps recherché équivaut la largeur d'un pixel sur l'écran) : $\mathcal{O}(\log N)$. Dans le pire cas, quand N_i s'approche de N, elle est de $\mathcal{O}(N)$.

5.2.3 Recherche d'entités non-instantanées dans un conteneur

Les entités non-instantanées sont calculées par le simulateur à partir des événements contenus dans le fichier de traces. Elles sont en général plus importantes que les entités instantanées, du point de vue des analyses des performances, et sont consultées plus fréquemment. Cependant, le tri simple utilisé pour organiser les entités instantanées ne produit pas des performances adéquates de recherche. Dans cette section, nous présentons les différentes organisations qui ont été envisagées pour améliorer ces performances.

Pour les entités consécutives dans le temps (comme par exemples les états successifs de fils d'exécution, cf. section 5.1.1), le problème posé par la recherche des données est très similaire à celui de la recherche des entités instantanées, et peut être facilement résolu par la même structure de données, avec le même temps de recherche.

Dans le cas des entités emboîtées (appels de fonctions) et des entités indépendantes (communications, liens), un simple tri ne suffit plus pour grouper les entités voulues, et il faut une structure plus complexe pour que la recherche des donnés soit faite en un temps raisonnable.

La figure 5.3 montre quelques entités, triées soit par date initiale soit par date finale. L'ordre du tri est représenté verticalement, du haut vers le bas : a,b,c,..,l dans le premier cas et c,b,e,a,d,..,h dans le deuxième. On veut identifier les entités qui ont une intersection avec l'intervalle de temps montré (dans ce cas, les entités a,d,f,g,h,i,j). Si on parcourt les entités dans l'ordre du tri, on trouve des entités qui n'ont pas d'intersection avec l'intervalle étudié qui se trouvent après des entités qui l'ont. Les entités voulues ne sont pas groupées pas un tri simple, on arrive pas à les identifier sans les tester individuellement. Nous devons donc trouver une organisation qui nous permette de minimiser le nombre de tests inutiles.

La figure 5.4 montre les configurations possibles des dates de début et de fin d'une entité par rapport aux dates de début et de fin de l'intervalle temporel de-

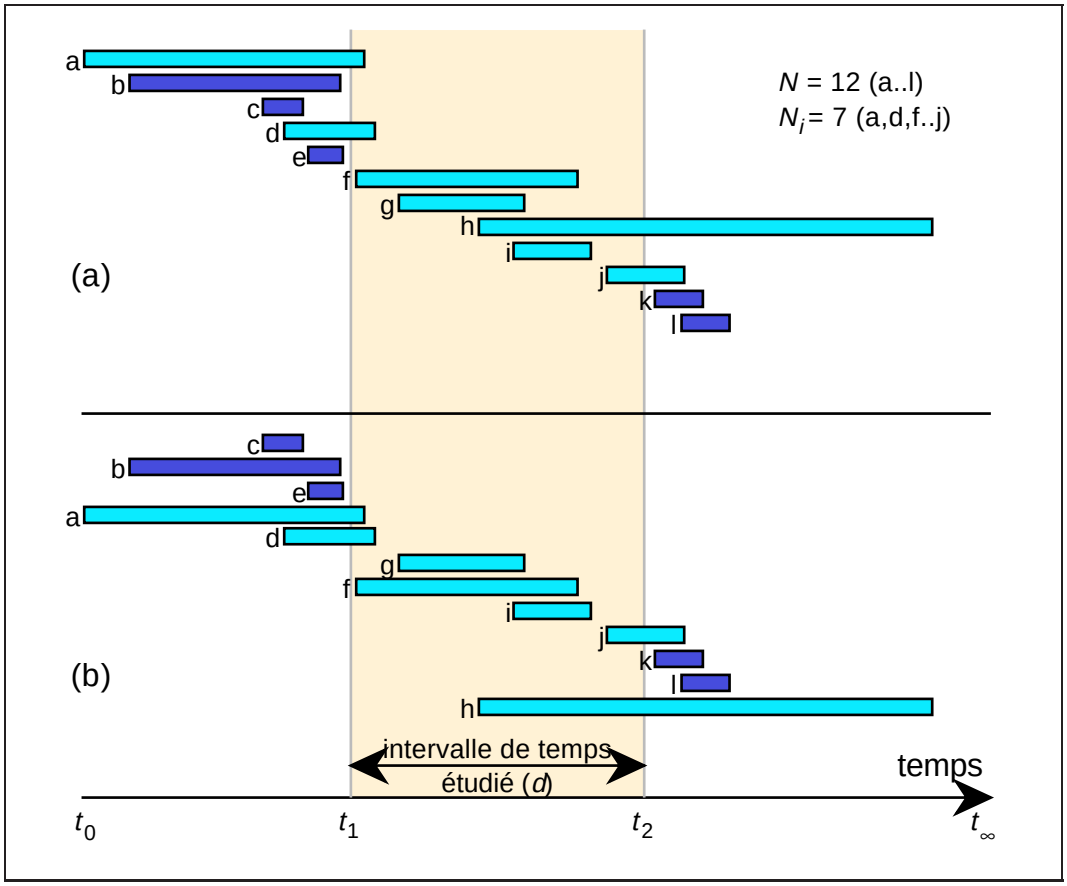

Figure 5.3 *Quelques entités, triées par (a) date de début et (b) date de fin*

mandé. On voit sur la figure que seules les entités des types « c » et « e » n'ont pas d'intersection avec l'intervalle voulu. Les entités qui ne sont pas recherchées sont celles qui ont terminé avant le début de l'intervalle (type « c ») et celles qui ont commencé après la fin de l'intervalle (type « e »). Autrement dit, nous cherchons toutes les entités dont la date de début est antérieure à la fin de l'intervalle *et* la date de fin est postérieure à la date initiale de cet intervalle : $e^i | t_1^i < t_2$ et $t_2^i > t_1$ où t_1^i et t_2^i sont la date initiale et finale de l'entité e^i, respectivement.

La figure 5.5 montre une représentation alternative des entités de la figure 5.3, un graphe avec la date initiale de chaque entité en abscisse et la date finale en ordonnée. Les points représentent les entités, et leur distance verticale (ou horizontale) à la ligne $x = y$ représente leur durée. Comme les entités terminent toujours après avoir commencé, toutes les entités sont en dessus[1] de cette ligne $x = y$. Les entités voulues sont celles qui sont placées dans le rectangle — en grisé sur la figure 5.5—

[1]On pourrait avoir des entités qui terminent avant d'avoir commencé, dans le cas de communications entre machines qui n'ont pas d'horloge globale.

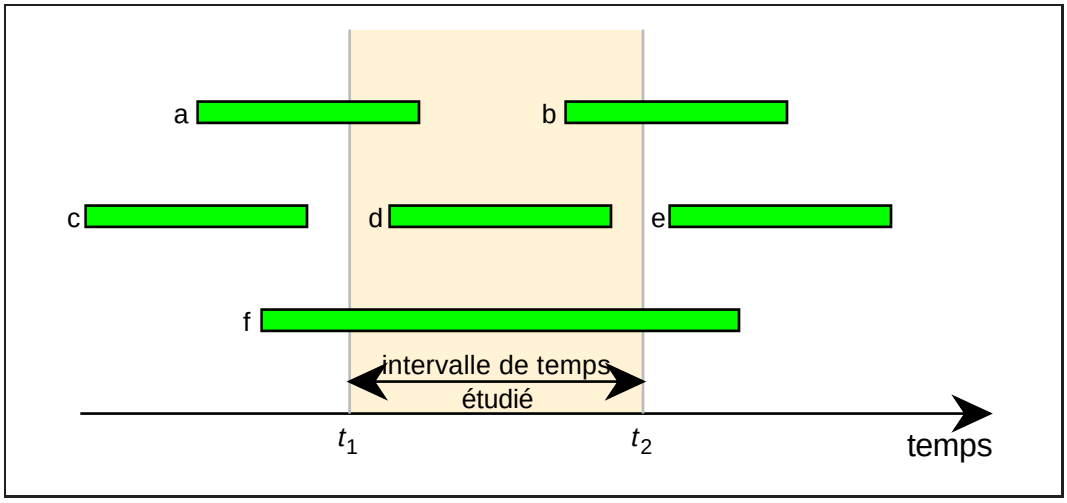

Figure 5.4 *Rapport entre les dates des entités et un intervalle de temps*
Seulement les configurations de type « c » et « e » ne sont pas dans l'intervalle.

limité par t_1 en ordonnée (bas) et par t_2 en abscisse (à droite).

Nous avons envisagé quatre organisations différentes des données dans la mémoire, avec pour objectif de les accéder le plus rapidement possible : tri simple, tri simple avec durée maximale, tri simple par $t_1^i + t_2^i$ et séparation en tranches selon la date initiale. Ces façons sont présentées ensuite, suivies d'un résumé comparatif.

5.2.3.1 Les symboles

Dans ce qui suit, nous utiliserons les symboles suivants :

t_0, t_∞	dates de début et de fin de la fenêtre d'observation
T	durée totale de la fenêtre d'observation ($T = t_\infty - t_0$)
N	nombre total d'entités dans la fenêtre d'observation (d'un certain type, dans un certain conteneur)
τ	nombre moyen d'entités par unité de temps d'exécution (supposé constant : $\tau = N/T$)
t_1, t_2	date de début et de fin de l'intervalle temporel exploré
d	largeur de l'intervalle temporel exploré ($d = t_2 - t_1$)
N_i	nombre d'entités dans l'intervalle temporel cherché

5.2.3.2 Tri simple

Le tri simple par une des deux dates n'est pas très efficace pour trouver les entités recherchées, quand elles ne sont pas instantanées ou successives. Ces entités

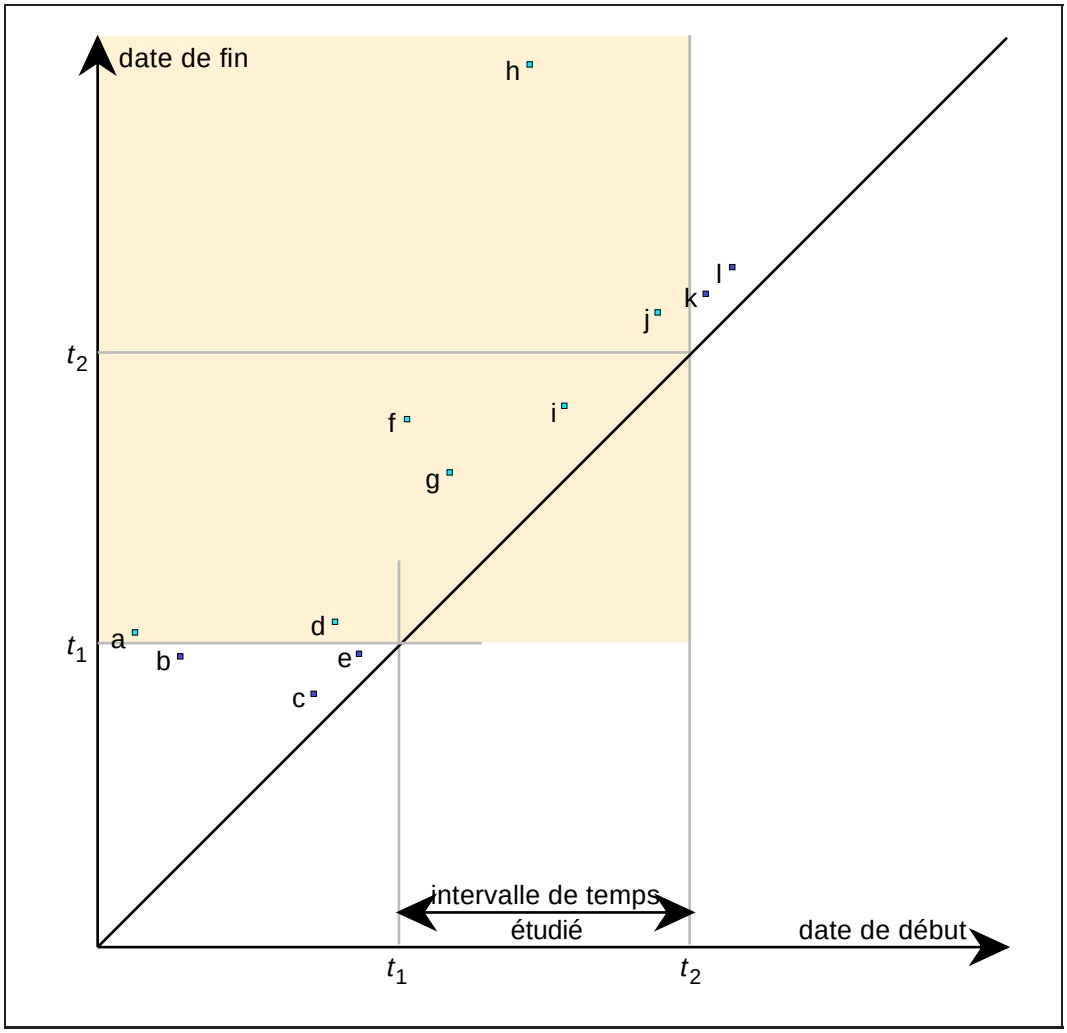

Figure 5.5 *Les entités de la figure 5.3, dans un graphe date initiale × date finale*

ont deux dates, et la recherche qui est faite est dépendante de ces deux dates, et pas d'une seule. Si on trie les entités simplement par date de début, on peut éliminer facilement de la recherche toutes les entités qui ont commencé après la fin de l'intervalle (type « e » de la figure 5.4), mais on doit rechercher toutes les autres entités, pour vérifier si leur date de fin est située après la date de début de l'intervalle. Les entités qui sont vérifiées correspondent aux points dont l'abscisse est inférieure à t_2 dans la figure 5.6. Toutes les entités dont l'ordonnée est inférieure à t_1 sont vérifiées en trop. Si l'intervalle recherché est vers la fin de la trace, cela peut signifier vérifier toutes les N entités. Le même problème se pose si le critère de tri est la date de fin et que l'on recherche le début des données. Dans ce cas, on vérifierait tous les points dont l'ordonnée dépasse t_1 dans la figure 5.6.

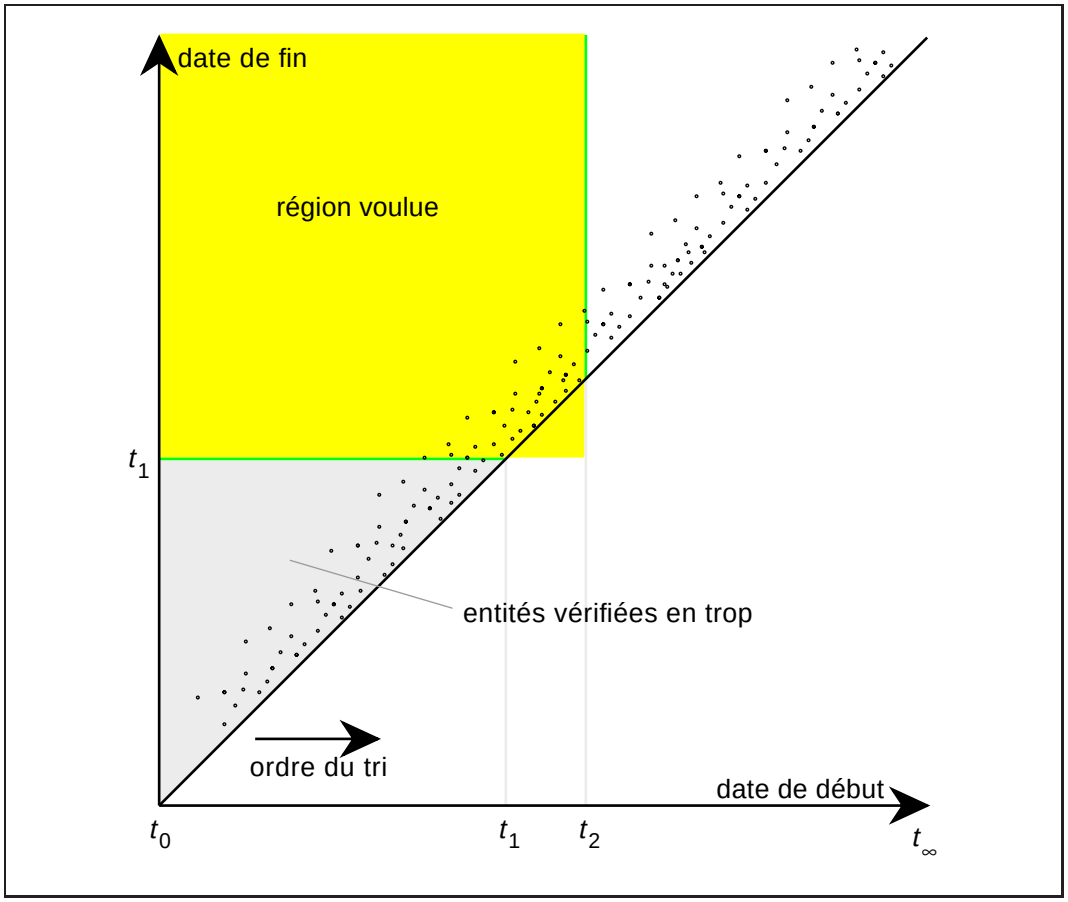

Figure 5.6 *Entités triées par date initiale*

Complexité en mémoire On a besoin d'un tableau trié pour stocker les N entités. L'espace mémoire requis est donc $\mathcal{O}(N)$

Complexité en temps Toutes les entités qui commencent avant t_2 sont testées. On a donc : $\tau(t_2 - t_0)$ entités à tester. Dans le pire des cas, $t_2 = t_\infty$, et le nombre d'entités à tester devient : $\tau(t_\infty - t_0) = N$

5.2.3.3 Tri simple avec durée maximale

Le problème posé par le tri simple est qu'il faut commencer la recherche au début (ou à la fin, si le critère de tri est la date de fin), parce qu'il peut exister des entités de grande durée, situées au début (resp. à la fin) de la trace et qui ont une intersection avec l'intervalle voulu. Or, si la durée de l'entité la plus longue est connue, une partie de la région de recherche peut être coupé. Aucune entité ne peut

avoir au même moment une intersection avec l'intervalle recherché et commencer à une distance du début de l'intervalle plus grande que la durée maximale. Cette durée maximale est representée par d_M sur la figure 5.7. Dans ce cas, au lieu de

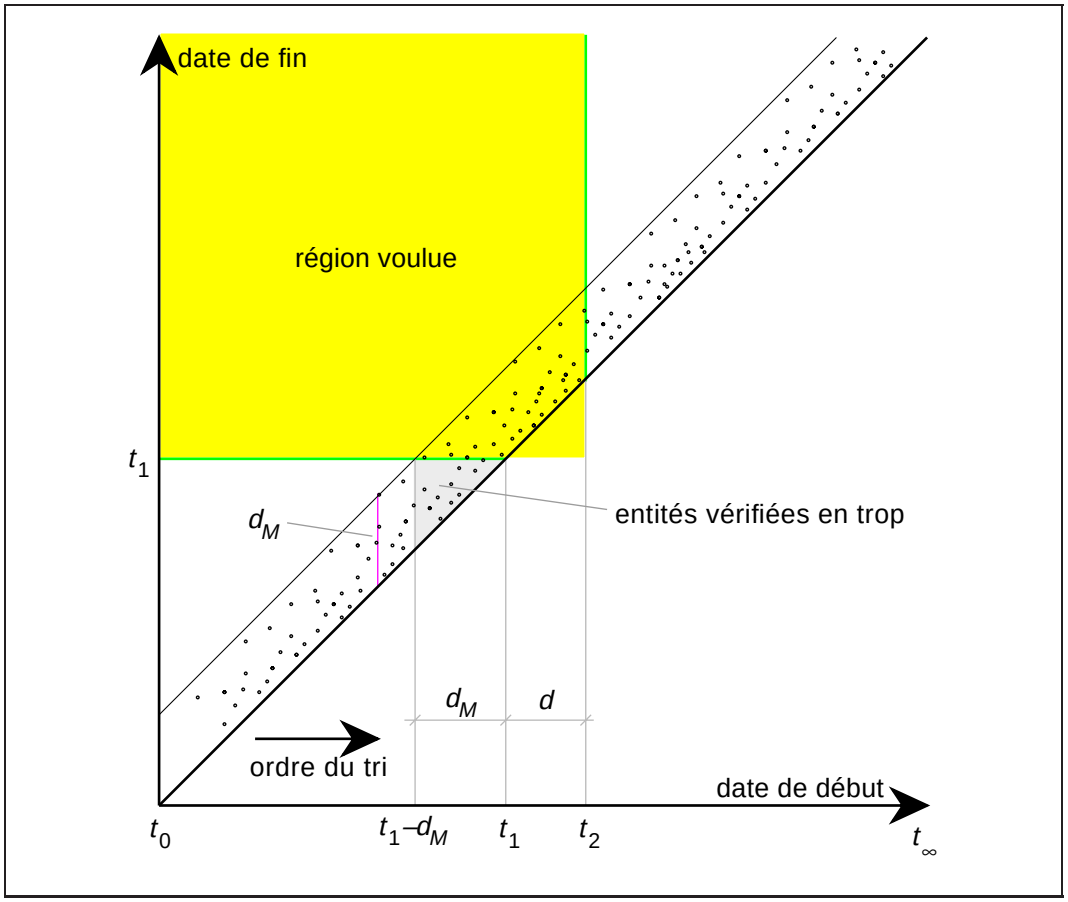

Figure 5.7 *Tri par t_1^i avec durée maximale*

chercher toutes les entités qui ont commencé avant la fin de l'intervalle recherché, on commence la recherche à l'instant qui correspond au début de l'intervalle moins la durée maximale (on cherche entre $t_1 - d_M$ et t_2). On limite ainsi les entités qui sont vérifiées inutilement. Le gain par rapport au tri simple de la section précédente c'est le test des entités entre t_0 et $t_1 - d_M$. Le problème est qu'il suffit d'avoir *une* entité qui dure très longtemps (ce qui signifie un d_M grand) pour que le gain dû à cette optimisation tombe. Plus la valeur de d_M est grande, plus $t_1 - d_M$ sera proche de t_0 et moins le gain sera grand. Dans le cas des entités emboîtées, la possibilité d'avoir une entité qui dure longtemps n'est pas négligeable, bien au contraire, les entités qui ont un bas niveau d'emboîtement tendant à être plus longues. Dans le cas des communications, il est possible d'en avoir quelques unes de grande durée, principalement si on considère la possibilité de communications asynchrones.

Complexité en mémoire Par rapport au cas de tri simple de la section précédente, on ajout simplement la durée maximale. La complexité en mémoire reste encore $\mathcal{O}(N)$

Complexité en temps On fait une recherche dichotomique dans le tableau pour trouver la première entité à tester (au temps $t_1 - d_M$). Pour cette recherche, le nombre d'entités testées est : $\log_2 N$. Puis, on teste les entités dont les dates de début sont situées entre $t_1 - d_M$ et t_2 pour vérifier leur intersection avec l'intervalle. Cela correspond à : $\tau(d + d_M)$ entités. Dans le pire des cas, $d_M \to d$ et $t_2 \to t_\infty$, et le nombre d'entités à tester devient N.

5.2.3.4 Tri simple par $t_1^i + t_2^i$

On peut aussi imaginer l'utilisation d'un tri qui utilise les deux temps, les entités étant par exemple triées par valeur croissante de $t_1^i + t_2^i$. Avec un tel tri, le parcours des entités est fait dans le sens perpendiculaire à la ligne $x = y$, ce qui est intuitivement plus proche de la forme de la région voulue que le sens vertical ou horizontal des autres tris. La figure 5.8 montre les entités affectées par une recherche de ce type, et met en évidence les recherches inutiles. L'avantage de ce tri relativement aux méthodes citées précédemment est qu'il n'est plus nécessaire de tester toutes les entités dans le pire des cas, la quantité d'entités à tester restant plus « stable ». Mais cette quantité représente quand-même à peu près la moitié des entités.

Complexité en mémoire Un tableau trié, comme dans les cas précédents, nécessite : $\mathcal{O}(N)$

Complexité en temps Le nombre de comparaisons necéssaires pour trouver la première entité à tester, avec une recherche dichotomique est $\log_2 N$. Nombre d'entités dont on doit tester l'appartenance à l'intervalle (entités entre $\frac{t_1+t_0}{2}$ et $\frac{t_\infty+t_2}{2}$) :

$$\tau[\frac{t_\infty + t_2}{2} - \frac{t_1 + t_0}{2}] = \tau\frac{(t_\infty - t_0) + (t_2 - t_1)}{2} = \tau\frac{T + d}{2} = \frac{N + \tau d}{2}.$$

Dans le pire des cas, $d \to T$, et le nombre d'entités à tester devient $\frac{N+\tau T}{2} = N$.

5.2.3.5 Séparation en tranches selon la date de début

La figure 5.9 montre une autre possibilité d'organisation des données. Dans cette organisation, l'espace des entités est divisé en tranches de temps, selon la date de début des entités. Dans un tableau correspondant à chaque tranche sont stockées

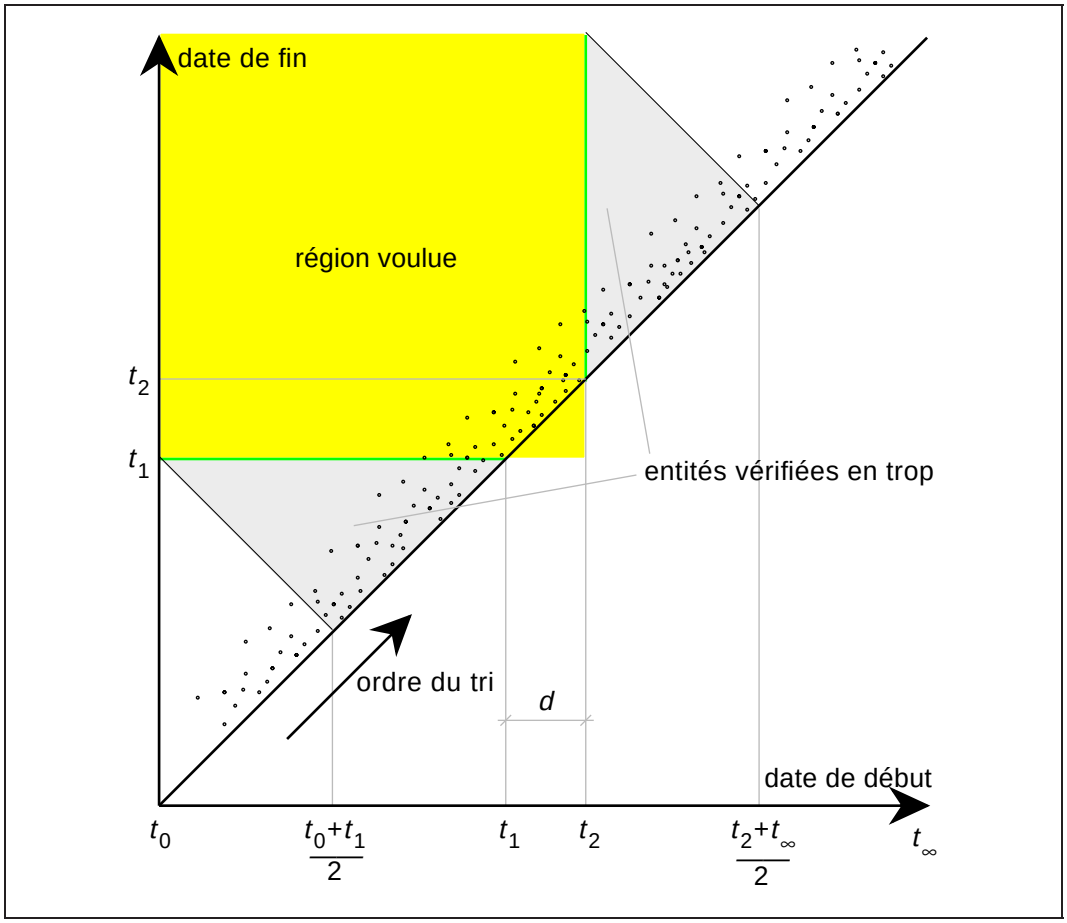

Figure 5.8 *Organisation des entités triées par* $t_1^i + t_2^i$

les entités dont la date de début se trouve dans cette tranche. Dans chaque tableau, les entités sont triées selon leurs dates de terminaison (t_2^i). De cette façon, on peut éliminer la recherche dans les tranches qui sont après la fin de l'intervalle recherché (entités telles que $t_1^i > t_2$). Sur les tranches initiales ($t_1^i < t_2$), il faut trouver les entités qui terminent après le début de l'intervalle recherché (celles qui ont $t_2^i > t_1$). Comme les entités sont triées par t_2^i dans chaque tranche, une simple recherche dichotomique suffira dans chaque tranche pour trouver la première entité qui appartient à l'intervalle. Toutes les entités de cette tranche qui se situent après cette entité initiale appartiennent à l'intervalle. Seule la tranche qui contient la fin de l'intervalle recherché (celle qui est traversée par t_2), pose quelques problèmes, parce qu'elle contient des entités qui commencent après t_2 et qui n'appartiennent pas à l'intervalle. Toutes ces entités doivent être testées pour voir celles qui ont $t_1^i < t_2$.

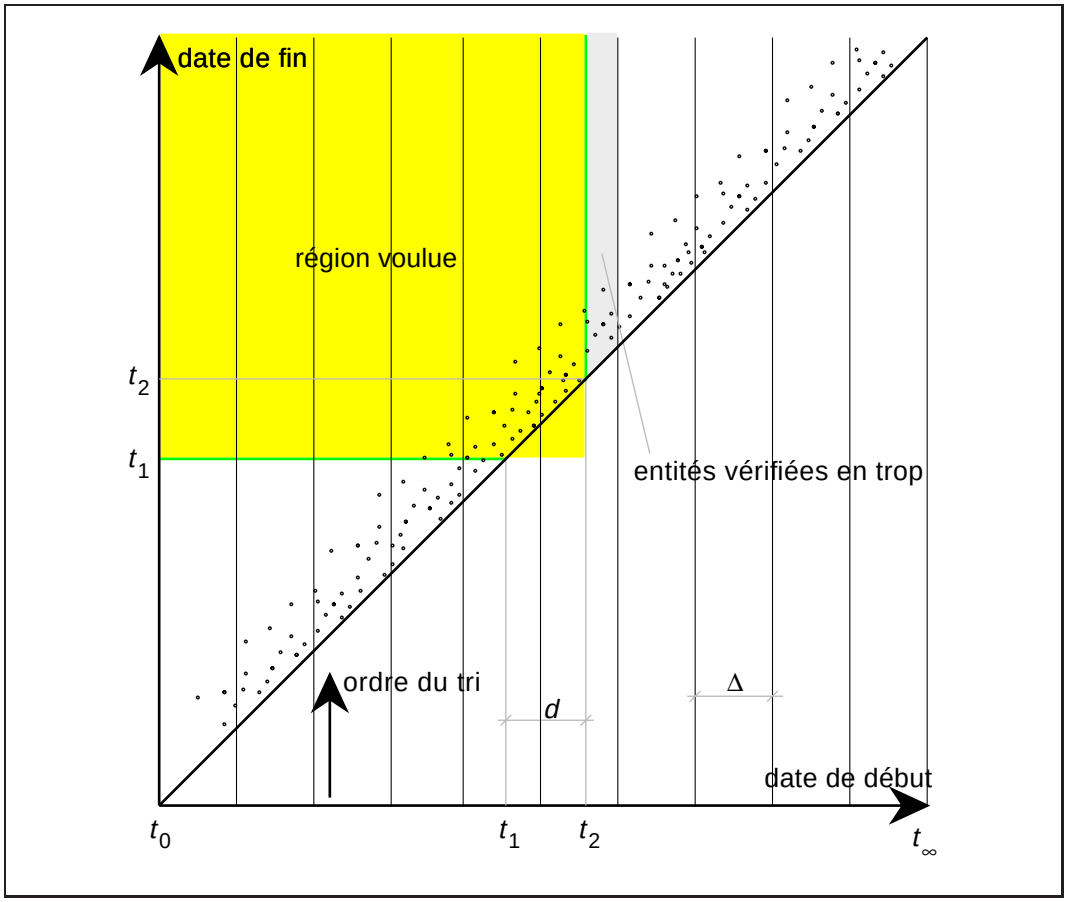

Figure 5.9 *Organisation des entités triées par t_2, séparées en tranches selon t_1*

Complexité en mémoire Soient :

Δ	la largeur de chaque tranche selon t_1
m	le nombre de tranches de largeur Δ ($m = \frac{T}{\Delta}$)
n_i	le nombre d'entités dans la tranche i
a	la première tranche de l'intervalle ($a = 1 + \lfloor \frac{t_1 - t_0}{\Delta} \rfloor$)
b	la dernière tranche de l'intervalle ($b = 1 + \lfloor \frac{t_2 - t_0}{\Delta} \rfloor$)

On a besoin d'un tableau pour stocker les m tranches : $k_0 + mk_1$. Chaque tranche contient un tableau trié de n_i éléments : $\sum_{i=1}^{m}(k_0 + n_i k_2)$. Total ($\sum_{i=1}^{m} n_i = N$) : $k_0 + m(k_0 + k_1) + Nk_2$. Complexité : $\mathcal{O}(m + N)$

Complexité en temps La figure 5.10 montre comment les données sont organisées dans la mémoire. Pour rechercher les entités qui ont une intersection avec l'intervalle d'intérêt, on a besoin de trouver la première entité qui a une intersec-

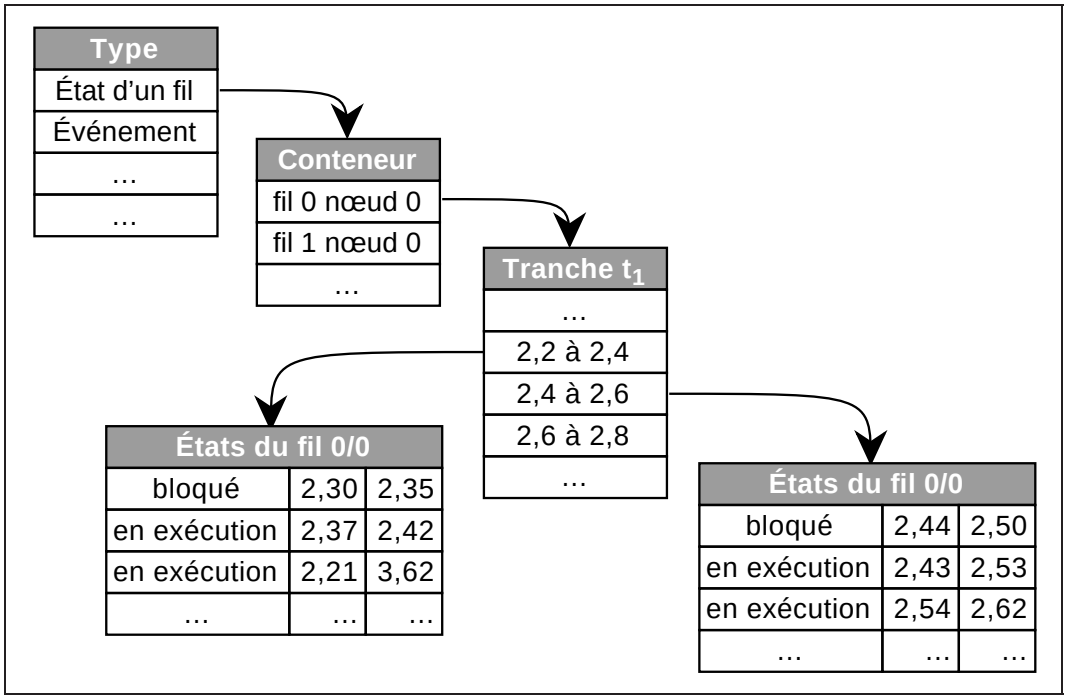

Figure 5.10 *Organisation des entités triées par t_2, séparées en tranches selon t_1*

tion avec l'intervalle dans chaque tranche entre t_0 et t_1. Cela correspond à réaliser : $\sum_{i=1}^{a-1} \log_2 n_i$ comparaisons. Seules les entités de la tranche b sont testées individuellement pour vérifier si elles ont une intersection avec l'intervalle. Donc, on fait n_b tests.

Si on suppose n_i constant et égal à $\frac{N}{m}$, le nombre de comparaisons devient : $a \log_2 \frac{N}{m} + \frac{N}{m}$. Dans le pire cas, ($a = m$), et le nombre de comparaisons devient donc : $m \log_2 \frac{N}{m} + \frac{N}{m}$

5.2.3.6 Résumé

Les tableaux 5.1 et 5.2 montrent un résumé des consommations en termes d'espace et de temps des méthodes qui ont été discutées dans les sections précédentes.

Le problème avec les deux premières méthodes est que le pire cas peut se produire facilement. Pour le tri simple, il suffit que la tranche de temps recherchée concerne la mauvaise extrémité de la fenêtre d'observation. Pour le tri simple avec durée maximale, une entité trop longue suffit pour qu'il devienne comme le tri simple. Dans le cas du tri par deux dates, son cas normal est déjà de comparer la moitié des entités, ce qui est inadmissible.

Méthode	Espace
tri simple	$k_0 + Nk_2$
tri simple + d_M	$k_0 + Nk_2 + 1$
tri par $t_1^i + t_2^i$	$k_0 + Nk_2$
tranches par t_1^i	$k_0 + Nk_2 + m(k_0 + k_1)$

Tableau 5.1 *Complexité en mémoire*

Méthode	comparaisons	comparaisons (pire cas)
tri simple	$\tau(t_2 - t_0)$	N
tri simple + d_M	$\log_2 N + \tau(d + d_M)$	N
tri par $t_1^i + t_2^i$	$\log_2 N + \frac{N+\tau d}{2}$	N
tranches par t_1^i	$a \log_2 \frac{N}{m} + \frac{N}{m}$	$m \log_2 \frac{N}{m} + \frac{N}{m}$

Tableau 5.2 *Complexité en temps (recherche)*

Dans le cas du tri par tranches, qui est la méthode adoptée dans Pajé, le problème est de choisir le nombre de tranches m qui donne les meilleures performances. Plus m est petit, plus on aura de données dans chaque tranche. La méthode de recherche peut réaliser des comparaisons avec toutes les données dans la dernière tranche de l'intervalle. Dans le cas où il y a beaucoup de données par tranche, les performances peuvent ne pas être bonnes. À noter le cas où $m = 1$, quand la méthode s'équivaut à un tri simple par date de fin. Si, au contraire, m est trop grand, le temps pour faire les recherches dichotomiques dans chacune des tranches avant le début de l'intervalle domine le temps de recherche.

Nous avons mesuré le temps de recherche avec différentes quantités de données, différentes valeurs de m et différentes tailles d'intervalles de recherche. Nous avons constaté que la gamme de valeurs de m ayant une performance acceptable est assez large. La figure 5.11 présente quelques courbes comparatives. Chaque courbe comporte en abscisse la position de la recherche dans la fenêtre d'observation (0 correspond au début de la fenêtre d'observation et 1 correspond à sa fin) et en ordonnée le temps pour réaliser chaque recherche. Chaque ligne représente une valeur de m, et les différents graphiques montrent différents nombres d'entités dans la fenêtre d'observation ($N = 1000$, 10000 et 100000) et différentes tailles de recherche (intervalles d d'un millième et d'un cinquième de la taille de la fenêtre d'observation). On peut observer que la forme générale des courbes est assez proche pour les différentes valeurs de nombre d'entités et de taille de la recherche. On voit que la performance des recherches avec $m = 1$, qui correspond au tri simple est mauvaise surtout au début de la fenêtre ; la performance s'améliore nettement pour $m = 3$ et reste acceptable jusqu'à ce que m s'approche de $\pm\frac{N}{30}$.

D'autres organisations des données sont envisageables, comme par exemple

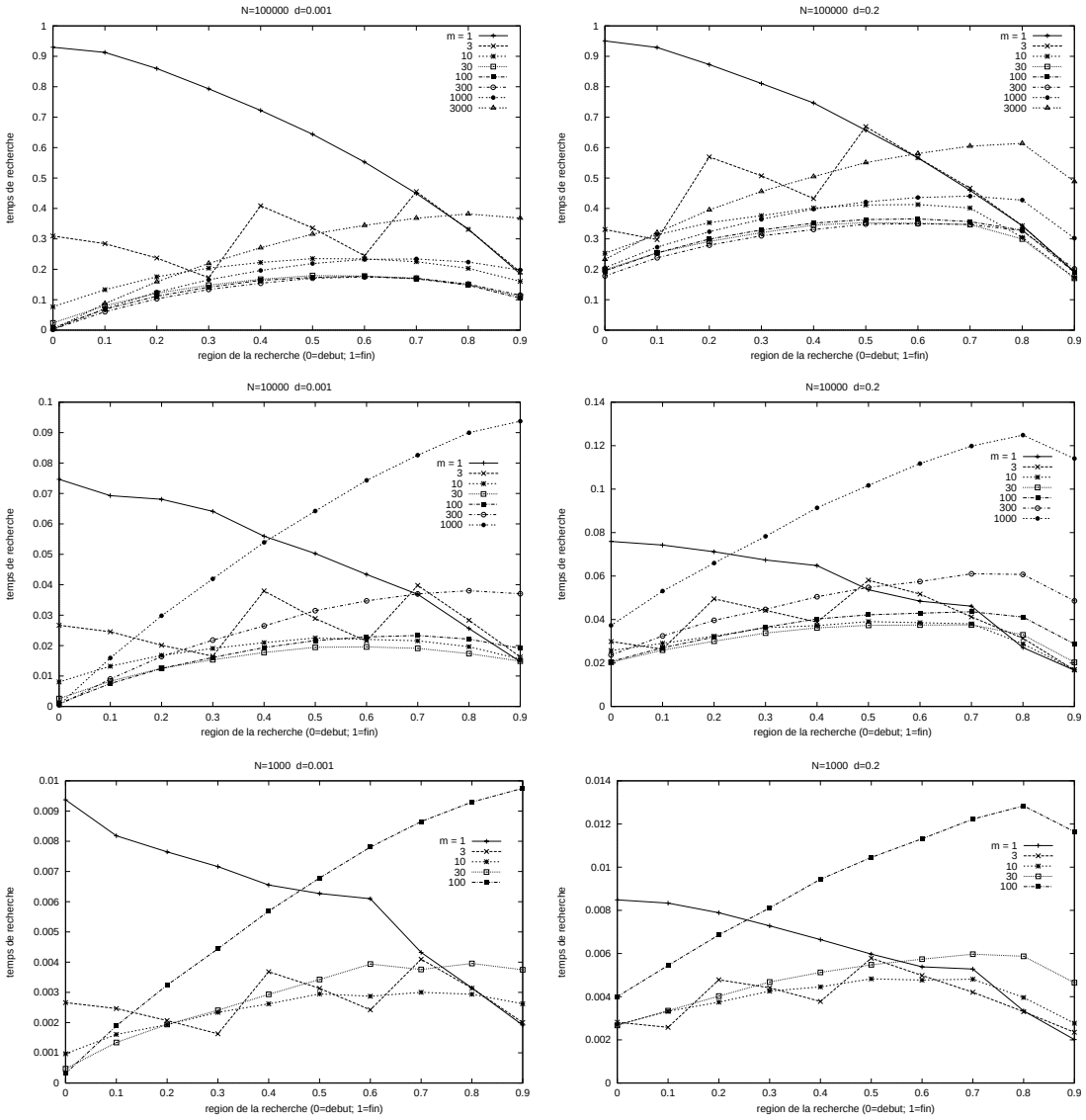

Figure 5.11 *Comparaison du temps de recherche des entités*
En abscisse, la position de l'intervalle recherché dans la fenêtre d'observation (0 correspond au début de la fenêtre d'observation et 1 correspond à sa fin) et en ordonnée le temps pour réaliser chaque recherche. N *est le nombre d'entités dans la fenêtre d'observation et* d *est la fraction de ces entités qui est recherché.*

d'utiliser la durée des entités pour les classer. Comme la dernière organisation présentée nous a donnée les performances attendues, nous avons arrêté nos recherches.

5.3 Interactivité dans Pajé

L'interactivité a pour but d'aider l'utilisateur à naviguer parmi les données de l'exécution de son programme en utilisant les visualisations proposées par l'outil. Dans Pajé, le support à l'interactivité emprunte deux voies : la clarification des présentations des données et l'exploration de ces données. Pour clarifier les présentations des données, Pajé permet d'identifier interactivement la correspondance entre les représentations graphiques et les données, sans encombrement de l'espace écran. En plus, les visualisations sont amplement configurables : l'utilisateur peut choisir les couleurs, formes et tailles qui lui sont plus significatives pour représenter les données. L'exploration des données se traduit par une navigation facile parmi les données — dans le temps, dans l'espace (parmi les diverses entités), dans les différents niveaux d'abstraction des données — et l'identification des liens entre les données représentées. Les caractéristiques principales d'interactivité de Pajé sont abordées dans les sections qui suivent : tout d'abord l'interactivité dans le diagramme espace-temps, suivi de la communication inter-modules destinée à l'interactivité.

5.3.1 Interactivité dans le diagramme espace-temps

Le principal module de visualisation de Pajé est un diagramme espace-temps (voir figure 5.12). Ce diagramme représente l'évolution dans le temps des diverses entités du programme parallèle (états des fils d'exécution, communications, états des verrous d'exclusion mutuelle, etc.). Dans cette section, nous abordons les caractéristiques d'interactivité existantes dans ce module. Son fonctionnement a été décrit dans la section 3.4.2.4.

Quand l'utilisateur désigne à l'aide de la souris l'une des représentations d'une entité, une description succinte de l'entité est montrée en haut de la fenêtre, sa représentation est mise en évidence, et les *entités liées* avec l'entité sélectionnée sont aussi mises en évidence. Dans le cas d'ATHAPASCAN-0, les liaisons qui définissent une entité liée peuvent être :

- Pour un état d'un fil d'exécution, les événements qui ont participé à sa construction.
- Pour un état bloqué en synchronisation, les états correspondants de l'objet de synchronisation (verrou d'exclusion mutuelle, sémaphore).
- Pour l'état bloqué d'un objet de synchronisation, tous les états des fils d'exécution qui sont bloqués par cet objet à cette date.
- Pour une communication, les événements qui en marquent les limites.

Un exemple de mise en évidence des liaisons entre entités est présenté dans la figure 5.12, où on peut voir qu'un état, correspondant à la durée durant laquelle un

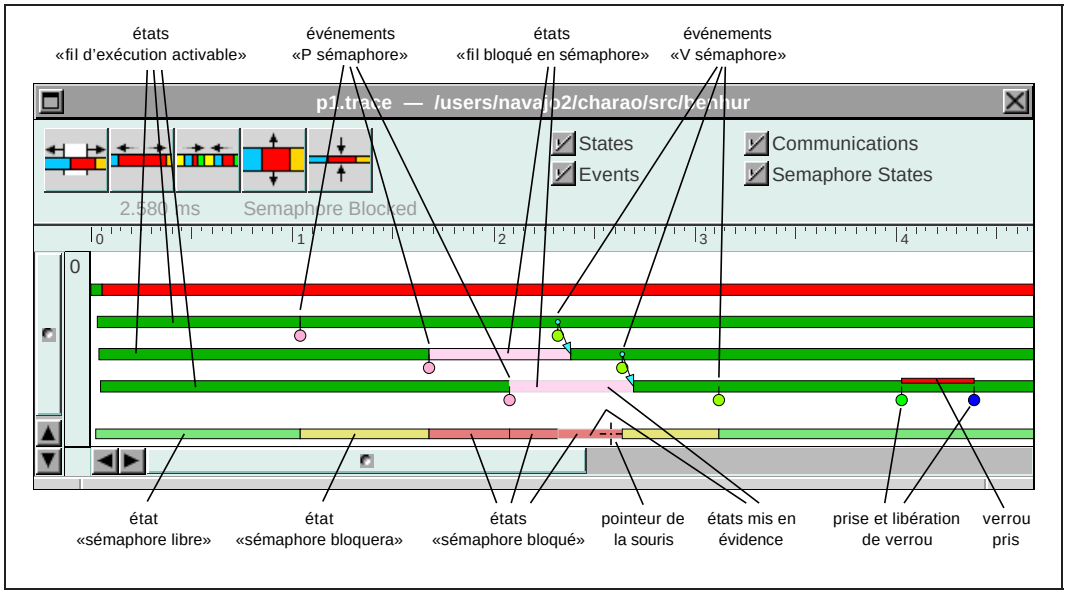

Figure 5.12 *Mise en évidence d'entités liées*
Quand le pointeur de la souris est mis sur l'état bloqué du sémaphore, les états des fils d'exécution bloqués dans ce sémaphore à cet instant (dans le cas, un seul fil d'exécution est bloqué) sont mis en évidence, en même temps que l'entité sous le pointeur. On peut voir aussi qu'une description succincte de l'entité sous le curseur est affichée en haut de la fenêtre (« Semaphore Blocked »).

fil d'exécution est resté bloqué en attente d'un jeton du sémaphore, est mis en évidence par surbrillance, lorsque le pointeur de la souris est sur l'état correspondant du sémaphore.

Si l'utilisateur sélectionne (en pressant le bouton de la souris) l'une des représentations d'une entité, une « fenêtre d'inspection » est ouverte, comportant toutes les informations connues à propos de cette entité. Si la trace générée comporte des informations de liaison avec le code source, cette fenêtre en permet l'affichage, mettant en évidence la ligne qui a généré l'événement inspecté. La figure 5.13 montre une fenêtre d'inspection d'une entité ATHAPASCAN-0.

5.3.2 Communication inter-modules

Une information peut être représentée sous différentes formes, éventuellement par différents modules de visualisation. Pour faciliter l'utilisation de l'outil de visualisation, il est important que l'utilisateur puisse retrouver facilement les différentes représentations d'une information. Pour cela, l'interaction de l'utilisateur avec un module de visualisation doit être diffusée aux autres modules de visualisation, afin qu'ils puissent mettre en évidence les autres représentations éventuelles de l'objet

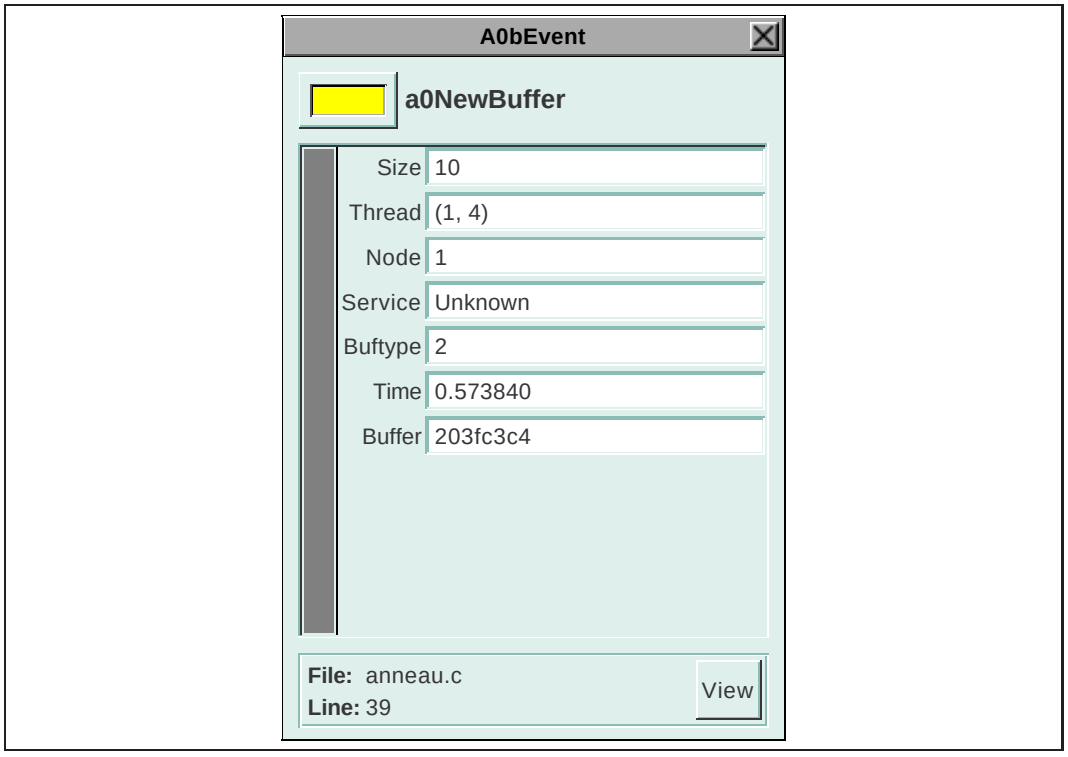

Figure 5.13 *Une fenêtre d'inspection*

manipulé par l'utilisateur. Pour éviter de surcharger l'outil, seules les interactions explicites de sélection sont diffusées.

Par exemple, la sélection d'une ligne du code source dans la fenêtre prévue à cet effet provoque la mise en évidence des représentations des événements qui ont été générés par l'exécution de cette ligne par le(s) module(s) qui les affichent.

Les modules de visualisation temporelle, comme le diagramme espace-temps, permettent la sélection d'un intervalle de temps. Cette sélection est diffusée aux autres modules. Elle sert à choisir une région pour le « zoom » ou, pour des modules qui affichent des informations synthétiques de la trace (un module de statistiques, par exemple), à générer ces informations sur l'intervalle sélectionnée de la trace. Cette interaction est montrée dans la figure 5.14.

5.3.3 « Zoom-out » et fenêtre d'observation

Quand on fait un « zoom-out » dans un module de visualisation, en réduisant l'espace alloué à chaque unité de temps, la quantité d'entités visualisées augmente. Il est possible que, contrairement à la situation habituelle où l'écran ne visualise

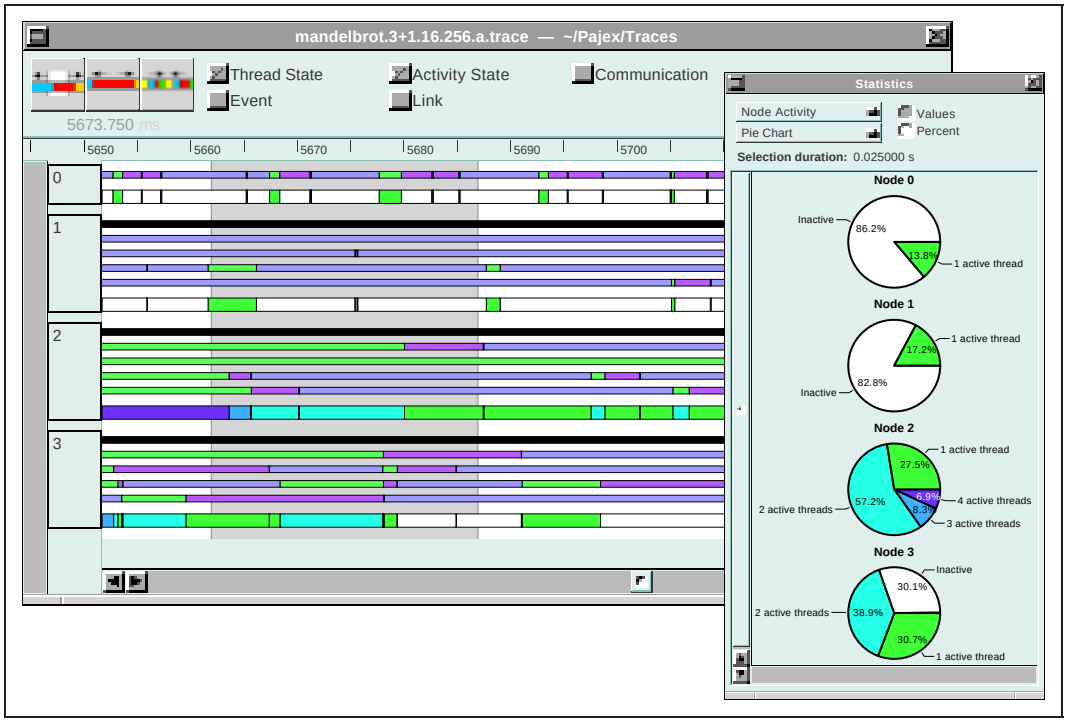

Figure 5.14 *Communication inter-modules*
La sélection d'un intervalle de temps sur un module de visualisation (la partie plus foncée du diagramme espace-temps) est communiquée aux autres modules de visualisation. Le module de statistiques présente l'accumulation des états qui représentent le nombre de fils d'exécution activables à chaque instant sur chaque nœud (les traits plus larges sur le diagramme espace-temps).

qu'une partie de la fenêtre d'observation, l'ensemble de la fenêtre d'observation se trouve incluse sur l'écran. Se pose alors le problème de la visualisation hors de la fenêtre d'observation. Comme il n'est possible de visualiser que les entités stockées dans la fenêtre d'observation, une première solution consisterait à limiter la possibilité de « zoom-out » aux limites de cette fenêtre. Dans Pajé, on utilise le fait que des données relatives aux conteneurs sont maintenues pour l'ensemble de l'exécution pour ne pas limiter la possibilité de « zoom-out ». Néanmoins la quantité d'information affichée hors des limites de la fenêtre d'observation est considérablement réduite puisque seules les dates de début et fin des conteneurs sont maintenues. On visualise donc seulement les contours des fils d'exécution pour tout ce qui est en dehors de la fenêtre courante. L'utilisateur peut quand-même changer la position de cette fenêtre dans ce mode d'opération.

5.4 Conclusion

Le premier problème posé par la conception d'un environnement de visualisation interactif a été de résoudre la contradiction entre la contrainte que pose cette propriété sur la gestion des données de visualisation et le choix de la structuration de l'environnement en graphe flot de données de composants, pour des raisons d'extensibilité. La solution à ce problème, qui a été exposée au chapitre 3, a consisté à casser l'aspect flot de donnée pur du graphe de composants en introduisant un composant particulier appelé encapsuleur et stockant l'ensemble des données de visualisation dans une structure appelée fenêtre d'observation.

Ce chapitre s'est intéressé essentiellement à un autre problème crucial pour l'interactivité à savoir le temps de réponse aux sollicitations des utilisateurs. Sachant que le moindre mouvement de souris provoque le réaffichage de partie de l'écran et que cette opération implique la recherche d'un petit nombre d'objets concernés parmi une grande quantité d'objets stockés, il est crucial d'optimiser ce type d'opération. Plusieurs structurations des données de visualisation ont donc été considérées et pour chacune d'entre eux ont été étudiés les coûts de stockage en mémoire ainsi que les coûts des opérations de recherche. Ce chapitre a également permis de rappeler un certain nombre de fonctionnalités de Pajé, résultant de son interactivité.

6

Aptitude au passage à l'échelle (« Scalabilité »)

L'aptitude à passer à l'échelle est une caractéristique importante d'un environnement de visualisation de programmes parallèles. Il s'agit d'être utilisable pour représenter l'exécution d'un programme sur un système parallèle de grande taille — plusieurs centaines de nœuds par exemple — ou de représenter l'exécution d'un programme de longue durée, les deux situations pouvant éventuellement être combinées.

Si la propriété de scalabilité implique que l'environnement puisse traiter de gros volumes de données, la réciproque serait réductrice. En effet, rien ne prouve qu'un environnement de visualisation sachant traiter de gros volumes de données puisse aider un utilisateur à résoudre un problème de performances se posant dans une des deux situations évoquées plus haut.

Une approche possible est de limiter le volume des données à visualiser en limitant le volume des données collectées. Les techniques développées dans ce but sont présentées dans la section 2.2.1 page 25. Même en utilisant ces techniques, la quantité de données à visualiser reste potentiellement importante.

Une autre approche envisageable est rechercher des visualisations indépendantes du nombre d'éléments à visualiser : « A *scalable view* is one whose format, clarity, meaning, and size are independent of the number of processing elements involved in computation » [15]. L'auteur de cette définition propose de grouper les éléments de calcul selon une caractéristique de comportement et de montrer les données de chaque catégorie, ce qui est scalable. On peut cependant remarquer que,

si une visualisation globale est utile pour commencer à l'existence et les causes de problèmes de performances, elle ne saurait être suffisante dans la mesure où elle omet forcément des données. Il est donc nécessaire que l'utilisateur puisse opérer des analyses fines sur des représentations détaillées, en plus de l'examen global de l'exécution fourni par une visualisation « scalable » dans le sens précédent.

L'approche qui a été suivie dans Pajé est de donner la possibilité au programmeur d'observer une exécution tracée à plusieurs niveaux d'abstraction : à un niveau d'abstraction élevé, il est possible d'observer l'ensemble du système parallèle ; si un problème concernant un groupe de nœuds ou un groupe de processus est identifié, il est alors possible d'observer en détail l'exécution du groupe concerné. On ne s'intéresse pas à définir des visualisations « scalables » différentes des visualisations détaillées mais plutôt à rendre « scalables » des visualisations qui ne le sont pas a priori. Ainsi, les visualisations proposées aux différents niveaux d'abstraction ne sont pas indépendentes les unes des autres mais plutôt dérivent les unes des autres : une visualisation abstraite dérive d'une visualisation plus concrète par regroupement d'objets élémentaires en objets de niveau d'abstraction plus élevés. Par exemple, on peut passer d'une représentation de l'ensemble des fils d'exécution à une représentation plus abstraite où les fils d'exécution de chacun des nœuds sont regroupés. Ce mécanisme d'abstraction — ainsi que l'éclatement, passage inverse à un niveau de détails plus fin — permet de maintenir des liens entre les différents niveaux d'abstraction, de telle sorte qu'il est possible à un utilisateur de passer d'un niveau d'abstraction à un autre sans perte de contexte. Combinée à l'interactivité offerte par Pajé, la possibilité d'observer une exécution à plusieurs niveaux d'abstraction possibles permet à l'utilisateur de naviguer entre ces niveaux suivant l'état d'avancement de son étude et ses nécessités du moment.

L'aptitude au passage à l'échelle a été intégrée au reste de l'environnement Pajé par la définition de techniques de filtrage de données mises en œuvre par des composants spécialisés de l'environnement appelés **filtres**. La figure 6.1 présente un schéma d'analyse simple, avec un filtre entre l'encapsuleur et le module de visualisation. L'insertion ou la suppression dans le schéma d'analyse de ces filtres, faite dynamiquement durant la visualisation, ainsi que des changements de leur configuration, permettent de passer d'un niveau d'abstraction à un autre. L'utilisation de filtres permet de conserver dans la fenêtre d'observation l'intégralité des données nécessaires à une présentation détaillée, tout en ne présentant au module de visualisation que les données correspondant au niveau d'abstraction sélectionné par l'utilisateur. Les filtres évitent donc la duplication des données de la fenêtre d'observation — nécessaire sans cela pour chacun des niveaux d'abstraction — ou alors l'obligation pour les modules de visualisation de sélectionner les données à représenter et par conséquent de connaître la sémantique des programmes visualisés.

La « scalabilité » de la visualisation n'a pas de sens si elle est isolée. Ainsi il est

nécessaire que la collecte de données présente également cette propriété, pour qu'on puisse avoir des traces d'exécution de programmes de grande taille. De même, si les données sont préparées avant leur visualisation, il faut que les programmes qui les préparent soient aussi « scalables ».

Ce chapitre aborde tout d'abord les problèmes susceptibles d'affecter l'aptitude de passage à l'échelle de l'environnement, sans laquelle la réalisation d'un environnement de visualisation « scalable » n'a pas de sens. Il reprend ensuite la définition de visualisation « scalable » en examinant les différentes techniques existantes pour y parvenir. Le principe du passage à l'échelle dans Pajé, basé sur l'utilisation de techniques de filtrage mises en œuvres par des modules spécifiques, est alors décrit avant de présenter en détail les filtres les plus communément utilisés.

6.1 « Scalabilité » de l'environnement

Pour qu'un environnement de visualisation soit « scalable », la capacité à traiter un grand volume de données doit se manifester de la collecte à la visualisation. Un seul maillon faible dans cette chaîne peut être suffisant pour la briser. Les limitations de cette chaîne peuvent avoir origines diverses, comme par exemple :

- Limitations à cause de structures de données de taille fixe dans l'implementation de l'outil de visualisation. ParaGraph [33], par exemple, comporte une structure pré-allouée pour les processeurs, limitant la quantité maximale de processeurs que l'outil peut traiter. En conséquence, il n'est pas possible d'utiliser ParaGraph pour visualiser un programme dont les processus sont créés dynamiquement, en utilisant pour représenter chaque processus la représentation utilisée par ParaGraph pour les processeurs.
- Limitations de taille mémoire : c'est le cas de certains outils qui, comme Upshot [35], lisent toute la trace en mémoire où ils conservent les données relatives à l'ensemble de l'exécution. Il n'est ainsi pas possible de lire une trace trop grande et les capacités d'interactivité de l'outil s'en retrouvent en conséquence remises en cause car limitées aux exécutions sur faible nombre de processeurs ou aux programmes de taille modeste.
- Limitations des traceurs : certains traceurs, tels que UTE [70] par exemple, n'enregistrent les traces qu'à la fin de l'exécution du programme tracé. Durant toute une exécution tracée, les traces sont donc conservées en mémoire, ce qui limite sérieusement la taille des programmes observables.
- D'autres facteurs sont susceptible de limiter l'aptitude d'un environnement à passer à l'échelle : il peut par exemple s'agir de l'utilisation de programmes

intermédiaires non « scalables » pour le traitement des données avant la visualisation.

6.2 « Scalabilité » de la visualisation

Une visualisation « scalable » est une visualisation qui reste utilisable et utile même en présence d'un grand nombre de données.

Une classification [30] des techniques utilisées pour créer des visualisations qui supportent un grand nombre de données les sépare en quatre catégories :

- *Adaptation graphique* : l'outil choisit parmi différentes techniques graphiques existantes, plus ou moins adaptées à la quantité des données.
- *Réduction et filtrage* : visualisation de données qui représentent une abstraction des données originales, ou suppression de données choisies de la visualisation.
- *Disposition spatiale* : les éléments graphiques sont disposés de façon à ce que la taille de la visualisation s'agrandisse à une vitesse inférieure à la croissance de la quantité des données ; un exemple dans ce domaine est le choix d'une forme géométrique (hypercube) pour représenter spatialement les processus d'un système parallèle.
- *Défilement* : à un instant donné, la visualisation présente une vision localisée d'une masse de données beaucoup plus grande.

Pajé supporte les trois derniers points : réduction, filtrage et disposition spatiale avec l'utilisation de filtres, et défilement directement dans les modules de visualisation.

Une visualisation détaillée offre une aide importante, parfois cruciale, à la compréhension des causes d'une exécution inefficace. Cependant, on ne peut pas espérer visualiser (ou comprendre) une exécution complexe à un niveau de détail trop élevé. Même si la visualisation est possible, très peu de personnes sont capables de déceler un problème parmi une telle quantité de données.

La visualisation de données de performance à des niveaux d'abstraction plus élevés, comme un graphe comparant les taux d'utilisation moyens des processeurs, se prête bien à la recherche de l'existence de problèmes. À un niveau d'abstraction intermédiaire, comme un diagramme qui montre l'évolution des taux d'utilisation des divers processeurs (où de ceux qui présentent des problèmes), on peut chercher la localisation de ces problèmes, dans l'espace — module du programme, processeur — et dans le temps, mais pas toujours leurs causes. Leurs causes pourraient être trouvées en analysant un diagramme encore plus détaillé, avec les échanges et synchronisations entre les composants du programme, aux endroits où se posent potentiellement des problèmes.

Il est donc très important de pouvoir approfondir le niveaux de détail d'une visualisation, au fur et à mesure qu'on s'approche du problème, dans l'objectif d'en connaître les causes et de pouvoir les corriger. Il ne suffit pas d'avoir des visualisations à différents niveaux d'abstraction, il faut une liaison entre ces visualisations que permette de retrouver et analyser les mêmes problèmes aux différents niveaux. Ce principe a guidé la conception des techniques de passage à l'échelle dans Pajé dont la présentation fait l'objet du reste du chapitre. Parmi les techniques possibles présentées ci-dessus, Pajé implante les trois dernières qui concernent le défilement, la réduction et le filtrage ainsi que la disposition spatiale. La première technique envisageable n'a pas été retenue car elle risquerait de briser l'homogéneité des représentations pour l'utilisateur.

6.3 Principe du passage à l'échelle dans Pajé

Comme il a déjà été indiqué, la « scalabilité » a été l'une des motivations essentielles pour le développement de Pajé. Il n'était en effet pas possible de représenter l'exécution de programmes parallèles constitués de processus légers communicants, — créés et terminés dynamiquement et tels que l'exécution d'un programme, même de petite taille, est susceptible de générer un nombre important de ces processus — en utilisant les outils de visualisation existants.

La « scalabilité » est prise en compte dans l'ensemble de l'environnement Pajé. Il s'agit tout d'abord des structures de données employées dont aucune n'est de taille fixe et n'introduit de limitation supplémentaire à celles imposées par la machine et le système utilisés pour la visualisation. La contrainte de « scalabilité » a également été au centre des choix qui ont conduit à la définition de la fenêtre d'observation. Cette structure de données a déjà été présentée dans le chapitre 3, avec l'architecture de Pajé et détaillée dans le chapitre 5 consacré à l'interactivité. Rappelons que les contraintes d'interactivité imposent de conserver en mémoire les données générées par le simulateur. Le choix de ne conserver en mémoire, dans la fenêtre d'observation, que les données relatives à une tranche de temps de l'exécution et de faire glisser cette fenêtre dans le temps, selon les souhaits de l'utilisateur, permet de s'affranchir des sévères limitations auxquelles doivent faire face d'autres outils interactifs tels que Upshot [35], qui conserve en mémoire l'ensemble des données relatives à l'exécution observée. Afin de permettre ce glissement de la fenêtre d'observation dans le temps d'exécution du programme observé, il a été nécessaire de définir les mécanismes d'aggrégation de données nouvelles à la fenêtre ainsi que libération des données (*garbage collection*) rendues « périmées » par le glissement dans le temps.

La contribution essentielle présentée dans le reste de ce chapitre concerne la

mise en œuvre de visualisations « scalables ». Parmi les techniques envisageables (voir paragraphe précédent), le défilement a déjà été présenté à propos de l'interactivité et de la limitation de la consommation mémoire. Les deux autres techniques sont mises en œuvre dans Pajé par le **filtrage** de données qui est présenté dans ce qui suit. L'intérêt du filtrage découle du choix du diagramme espace-temps étendu comme principale interface avec l'utilisateur. Il s'agit d'un diagramme étendu, puisque de nombreuses informations sont agrégées aux lignes marquant l'écoulement du temps des entités représentées — fils d'exécution, nœuds ou groupes de nœuds suivant le degré d'abstraction de la visualisation —; il s'agit principalement de l'état d'activation de l'entité, habituellement (dans ParaGraph par exemple) représenté dans une autre visualisation appelée diagramme de Gantt ; il s'agit également des événements élémentaires autres que l'émission ou la réception de messages tels que les événements de synchronisation (prise et relachement de verrou, etc.) ; ce diagramme jouant le rôle d'interface pour l'utilisateur, il s'agit enfin de toutes les informations, susceptibles d'aider à l'identification des problèmes, que l'utilisateur peut faire afficher en inspectant l'un des objets visualisés. Le diagramme espace-temps a été choisi, en accord avec les premiers utilisateurs, comme principale visualisation et interface de Pajé car il permet de visualiser le comportement détaillé d'une application, avec l'évolution des états de ses divers composants et la relation temporelle et causale entre eux. Malheureusement ce diagramme n'est pas très « scalable », mis à part sa capacité à changer la taille des représentations graphiques et à déplacer la zone d'observation à un endroit quelconque de l'univers de données visualisées. Il ne possède aucune capacité de réduction des données. Plus il a de données à visualiser, plus il a besoin de place sur l'écran pour les afficher. Ce diagramme a néanmoins une forte indépendance relativement à sémantique des données qu'il présente (il ne connaît que le temps, la relation hiérarchique entre les données et la façon de les présenter), et peut présenter des données concises et synthétiques ainsi que des données très détaillées. Pour passer à un niveau d'abstraction plus élevé permettant de représenter un grand volume de données, il a donc été décidé de faire appel à des modules de filtrage[1], placés en amont dans le graphe de composants.

6.3.1 Le filtrage de données dans Pajé

Les filtres de Pajé sont des composants qui fabriquent, pour les composants placés à leur suite dans le diagramme des composants, une abstraction des données qu'ils reçoivent en entrée. Ces données peuvent provenir de la fenêtre d'observation ou d'un autre filtre. Les filtres ne modifient pas les données filtrées, ce qui permet

[1] Dans Pajé, nous ne faisons pas la distinction entre modules d'analyse et de filtrage de données discuté dans la section 2.3

de les accéder à nouveau si, par suite d'une requête utilisateur le plus souvent, le filtre est enlevé du diagramme de composants. Le processus de filtrage intervient dans le mécanisme d'accès aux données, les messages de contrôle (section 3.4.2.3), et dans la façon dont les données sont organisées, une hiérarchie changeable (section 3.4.2.2).

Les filtres fonctionnent en changeant les réponses aux requêtes sans modifier les données. Comme les modules de visualisation redemandent les données chaque fois qu'ils en ont besoin, un filtrage peut changer dynamiquement sa configuration et répondre différemment à une prochaine requête, ou simplement être désactivé, en court-circuitant ses entrés et sorties (à la prochaine requête, les données non filtrées seront visualisées). Pour garantir que les visualisations redemandent les données à chaque fois que se produit un changement de configuration d'un filtre, ce changement est accompagné d'une notification. Un exemple d'un schéma d'analyse comportant un filtre est présenté figure 6.1.

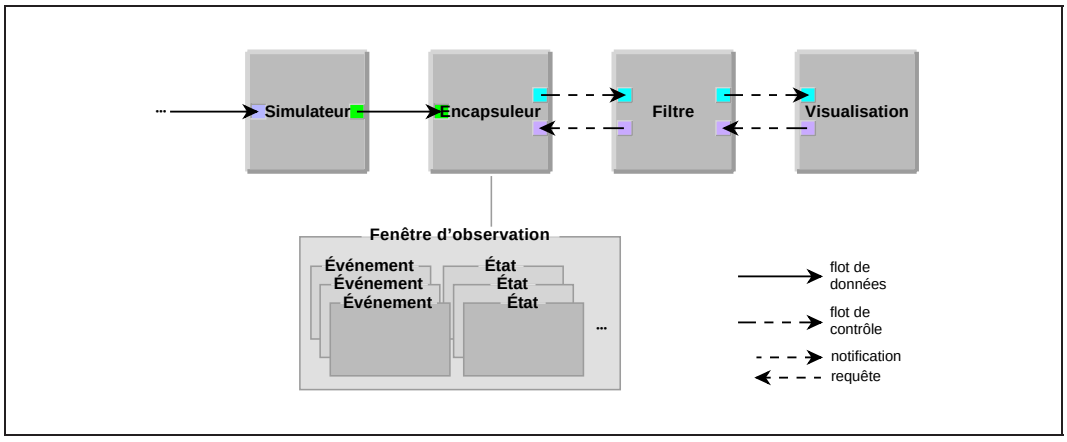

Figure 6.1 *Schéma d'analyse avec un filtre*
Quand la visualisation a besoin de données, elle envoi une requête en demandant au filtre. Le filtre peut répondre à la requête directement, s'il dispose des données nécessaires, où envoyer une(des) requête(s) à l'encapsuleur pour lui demander des données. La encore, le filtre peut changer la réponse de l'encapsuleur avant de répondre au module de visualisation.

Comme les modules de filtrage agissent sur les requêtes, ce sont les requêtes existantes (section 3.4.2.3) qui définissent les possibilités de filtrage. Le filtrage peut avoir pour effet d'éliminer des données de la visualisation ou d'en regrouper dans des objets visuels plus abstraits. Pour simplifier l'exposé nous nous intéressons tout d'abord à l'élimination de données. Les requêtes concernées sont les requêtes émises par les modules de visualisation pour connaître :

- La **hiérarchie des types** : cette requête constitue le seul moyen dont dispose un module de visualisation pour connaître l'existence d'un type de donnée à

visualiser. Il ne fera pas de référence, dans d'autres requêtes (cf. ci-dessous), à un type qu'il ne connaît pas. Cette requête est émise au début de la simulation ou après un changement de configuration d'un filtre. En intervenant sur cette requête, un filtre peut éliminer la visualisation de tous les conteneurs ou toutes les entités d'un certain type (tous les sémaphores, ou tous les événements, par exemple).Le module de filtrage n'a donc pas besoin de filtrer les autres types de requêtes. En filtrant la réponse à un seul type de requête, un filtre très simple peut donc complètement éliminer l'affichage de n'importe quel type de conteneur ou d'entité.

- La **hiérarchie des conteneurs** : à partir de la connaissance de la hiérarchie des types (cf. requête précédente), cette requête permet aux modules de visualisation de connaître les conteneurs de chaque type (les nœuds, les fils d'exécution, etc.). En filtrant cette requête, on peut éliminer sélectivement quelques conteneurs (un des nœuds, quelques fils d'exécution, etc.).
- Les **entités** d'un certain type appartenant à un conteneur donné, à partir de la connaissance de la hiérarchie des conteneurs fournie en réponse à la requête précédente. En filtrant la réponse à cette requête, on peut empêcher l'affichage de certaines entités, en général sur la base d'un de leurs attributs. Par exemple, on peut ne visualiser que les communications dont la taille est supérieure à une quantité donnée, ou les états des fils d'exécution qui attendent la libération d'un verrou d'exclusion mutuelle.
- Les **attributs** d'une entité (sa couleur, ses dates, son conteneur, etc.) : le filtrage de cette requête permet, par exemple, de choisir la couleur des flèches qui représentent les communications en fonction de leur taille. Le filtre qui réutilise l'espace écran, discuté dans la section 6.4.3 est un autre exemple de filtrage des attributs des entités.

En plus des techniques de filtrage présentées ci-dessus, qui éliminent des données, d'autres formes de filtrage sont utilisées pour en ajouter ou substituer. La substitution est utilisée par exemple pour regrouper des entités d'un certain niveau d'abstraction (des fils d'exécution par exemple) en entités d'un plus grand niveau d'abstraction (des nœuds par exemple). Des exemples plus détaillés sont présentés dans le paragraphe 6.4.

La correspondance entre données à deux niveaux d'abstraction (les données filtrées et les données non filtrées) est maintenue par chaque filtre. À la demande de l'utilisateur, un filtre peut « éclater » les données plus abstraites, permettant la visualisation des données plus concrètes correspondantes.

Les filtres peuvent être enchaînés : un filtre accède aux données produites par le module qui le précède dans le graphe pour produire des données destinées aux modules qui le suivent. De cette façon, les résultats des filtres sont cumulés ; il est possible de construire des filtrages puissants avec un ensemble réduit de filtres

simples. À propos de simplicité, si la plupart des filtres utilisés dans Pajé sont compatibles avec l'interactivité et la fenêtre d'observation, ce n'est pas le cas des filtres devant accumuler une information relative à l'ensemble d'une exécution tracée ; ces deux situations sont examinées dans les paragraphes suivants.

6.3.2 Filtres compatibles avec la fenêtre d'observation

Certains filtres s'intègrent très bien avec la fenêtre d'observation. Ce sont des filtres qui produisent des données suivant la même échelle de temps que les données qu'ils utilisent. Prenons comme exemple un filtre qui doit produire l'évolution des taux instantanés d'utilisation des processeurs à partir des états des fils d'exécution. Pour générer ces taux, correspondants à un certain intervalle du temps d'exécution, le filtre n'a besoin que des états des fils d'exécution correspondants durant le même intervalle. Chaque fois que le filtre reçoit une requête de production de taux d'utilisation, il peut demander les états des fils d'exécution au module précédant et effectuer les calculs nécessaires. De cette façon, les données produites par un filtre possèdent les mêmes caractéristiques de disponibilité que les données de la fenêtre d'observation (on dispose d'un accès direct aux données de la fenêtre ; la fenêtre glisse automatiquement dans le temps si l'utilisateur cherche à accèder à des données qui n'y sont pas). Les filtres qui ont ce type de fonctionnement n'ont donc pas besoin de traiter l'existence de la fenêtre d'observation d'une façon spéciale.

6.3.3 Filtres qui accèdent à toute la trace

Pour d'autres filtres au contraire, la fenêtre d'observation peut poser problème. Il s'agit des filtres qui ont besoin d'accéder aux données de toute la trace pour produire leurs résultats, et qui en plus ne doivent pas accéder plus d'une fois à chaque donnée, sous peine de produire des résultats faux. Les filtres calculant le taux global d'utilisation des processeurs, ou comptant le nombre de messages échangés pendant toute l'exécution du programme, en sont des exemples.

Si un tel filtre est implanté naïvement, il réalisera une requête pour les données du type voulu dans un intervalle qui correspond à toute l'exécution du programme, et fera ses comptes sur le résultat. Hélas, une requête de ce type risque d'effondrer les performances de l'outil, puisqu'elle implique une lecture complète de la trace, avec un glissement correspondant de la fenêtre d'observation. Ce type de filtre serait en revanche très facile à réaliser dans un environnement non interactif, implanté comme un graphe flot de données pur, et où il suffirait d'attendre l'arrivée de toutes les données, une par une jusqu'à la fin de la trace.

Une solution naïve consisterait à placer ces filtres dans le graphe flot de données

de Pajé (voir section 3.4.1), de façon à éviter l'encapsuleur et la fenêtre d'observation, comme dans la figure 6.2. Malheureusement, ce placement poserait aussi des

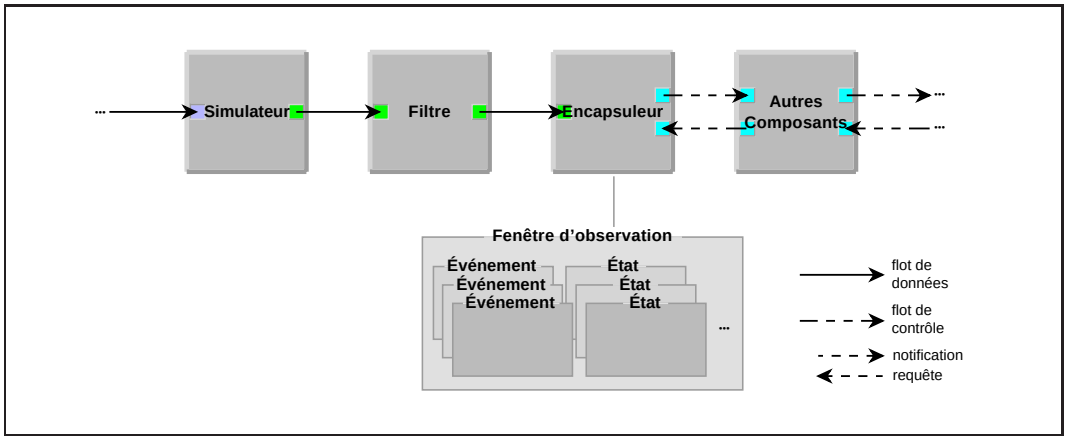

Figure 6.2 *Inclusion d'un filtre dans le flot de données*

problèmes. Le premier problème serait dû aux relectures de données générées par le mécanisme qui permet à la fenêtre d'observation de revenir en arrière dans la trace (cf. section 3.4.2.5). Ce problème pourrait être résolu en enregistrant et récupérant l'état interne de ces filtres en même temps que ceux du simulateur et du lecteur de traces. Un deuxième problème serait le manque d'accès, par le nouveau filtre, aux données produites par les filtres qui sont dans le flot de contrôle. La visualisation des données produites par le nouveau filtre serait aussi problématique, puisqu'il faudrait soit insérer les données dans la fenêtre d'observation, soit disposer des modules de visualisation dans le flot de données.

L'autre possibilité, qui semble préférable, consiste à placer le filtre après l'encapsuleur. Dans ce cas, le filtre peut avoir accès aux données produites par les autres filtres, mais il est nécessaire qu'il n'accède qu'une seule fois à chaque donnée. Une façon de résoudre le problème consiste à attendre les notifications de glissement de la fenêtre d'observation et de demander des nouvelles données quand la fenêtre glisse en avant. Pour ne pas traiter deux fois la même donnée, il faut que le filtre connaisse la date jusqu'à laquelle les données ont déjà été traitées. Il faut considérer aussi le fait que les données non instantanées peuvent être reçues plus d'une fois, même si les intervalles des requêtes n'ont pas d'intersection. Par exemple, une communication envoyée à la date 9 et reçue à la date 11 sera citée dans les réponses aux requêtes demandant « toutes les communications entre les dates 5 et 10 » ainsi que dans celles qui demandent « toutes les communications entre les dates 10 et 15 ». Dans ce cas, le filtre peut compter uniquement les dates de réception, et ignorer les communications qui ne sont pas reçues dans la tranche de temps de la requête.

6.4 Exemples de filtres de Pajé

Nous avons développé des filtres de trois types dans Pajé : des filtres de **réduction**, qui produisent des données abstraites à partir de données plus élémentaires ; des filtres de **sélection**, qui permettent de ne pas afficher certaines entités, et des filtres de **repositionnement**, qui améliorent le placement des entités sur l'écran. Ces trois types de filtres sont discutés dans les sections qui suivent. Il convient de remarquer que tous ont un bon degré de compatibilité avec la fenêtre d'observation (voir les paragraphes précédents).

6.4.1 Filtres de réduction

Un filtre de réduction produit une entité nouvelle à partir de l'application d'une fonction de réduction à un nombre quelconque d'autres entités. L'objectif d'une telle réduction est de générer des données plus abstraites et moins nombreuses, pour en produire des visualisations plus synthétiques. Un exemple de filtre de réduction dans Pajé est un module qui, à partir des états des fils d'exécution d'un nœud, produit des états d'activité de ce nœud. Les états d'activité représentent le nombre de fils d'exécution simultanément activables à chaque instant (voir figure 6.3).

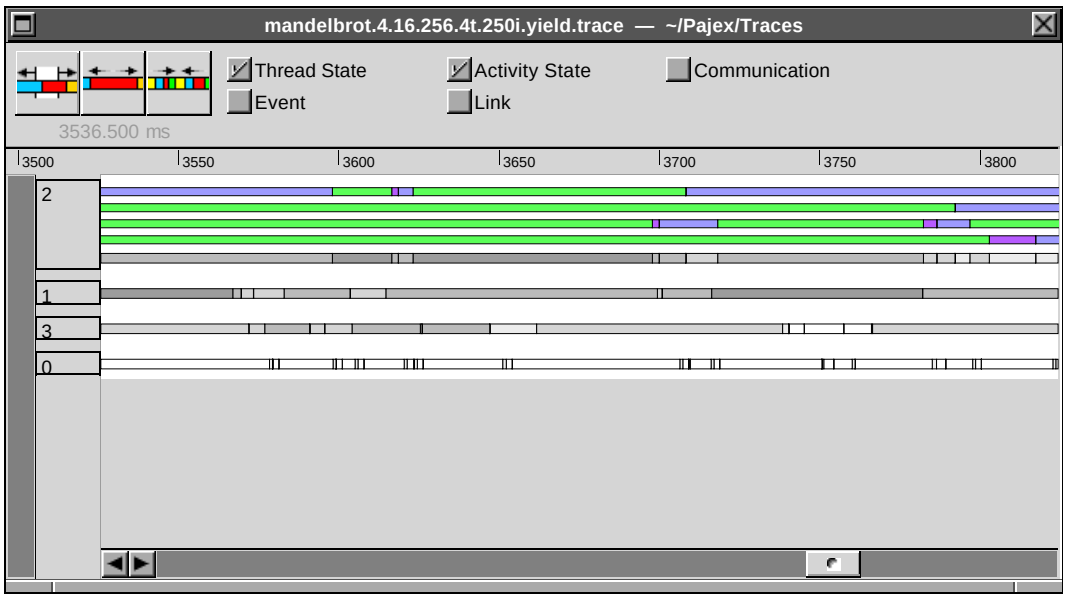

Figure 6.3 *Filtre de réduction synthétisant l'utilisation des nœuds*
La dernière ligne du nœud 2 et la seule ligne des autres nœuds représente, en dégradé, le niveau d'utilisation du nœud. Plus il comporte de fils d'exécution activables, plus sa couleur est foncée ; les intervalles où le nœud est oisif sont représentés en blanc.

Un deuxième exemple de filtre de réduction est un filtre de regroupement, qui permet de diminuer le nombre de conteneurs en réunissant des conteneurs (en général à comportement similaire) dans des groupes de conteneurs. Tous les membres d'un groupe sont représentés comme s'ils n'en faisaient qu'un seul ; le groupe devient un nouveau conteneur qui remplace ses membres. Les entités qui appartiennent à n'importe lequel des membres du groupe sont représentées comme si elles appartenaient au groupe. Dans la figure 6.4 (b), on voit que les communications de tous les fils d'exécution des nœuds groupés sont envoyées au (ou reçues du) seul groupe. Un seul filtre dans Pajé permet de grouper n'importe quel type de conteneur (des nœuds, des fils d'exécution, des types de conteneurs définis par l'utilisateur, etc.).

Le principe de fonctionnement d'un filtre de réduction consiste à donner une version altérée de la hiérarchie de types ou de la hiérarchie de conteneurs aux composants en aval dans le graphe de composants et à gérer les requêtes, réponses et notifications concernées par ce changement.

Le filtre de regroupement, par exemple, donne une hiérarchie altérée des conteneurs, où des groupes remplacent les conteneurs groupés. Les modules en aval ne verront que cette structure altérée, et réaliseront des requêtes pour des données appartenant aux nouveaux conteneurs (groupes). Le travail du filtre est de traduire chacune de ces requêtes en plusieurs, une pour obtenir chaque membre du groupe, au module en amont, et de regrouper les réponses.

6.4.2 Filtres de sélection

Un filtre de sélection permet de supprimer des informations qui ne sont pas intéressantes pour une analyse, afin de mettre en évidence d'autres informations. Cette suppression peut se faire selon plusieurs critères :

- Par type d'entité : les entités d'un certain type ne sont pas montrées. Comme ce filtrage a une utilisation très courante, il est directement accessible dans la fenêtre principale du diagramme espace-temps de Pajé. On peut voir dans la figure 6.6 que seuls les événements de type « Thread State » sont montrés ; les autres (« Event », « Activity State », « Link », « Communication ») ont été supprimés de la visualisation.
- Par nom d'entité : les entités sont filtrées selon leurs noms. On peut, par exemple, ne visualiser que les états qui correspondent aux périodes où les fils d'exécution sont bloqués dans des sémaphores. La figure 6.6 montre un exemple de filtrage par nom : seuls les états « Thread runnable », les états « activables » des fils d'exécution, restent après le filtrage.

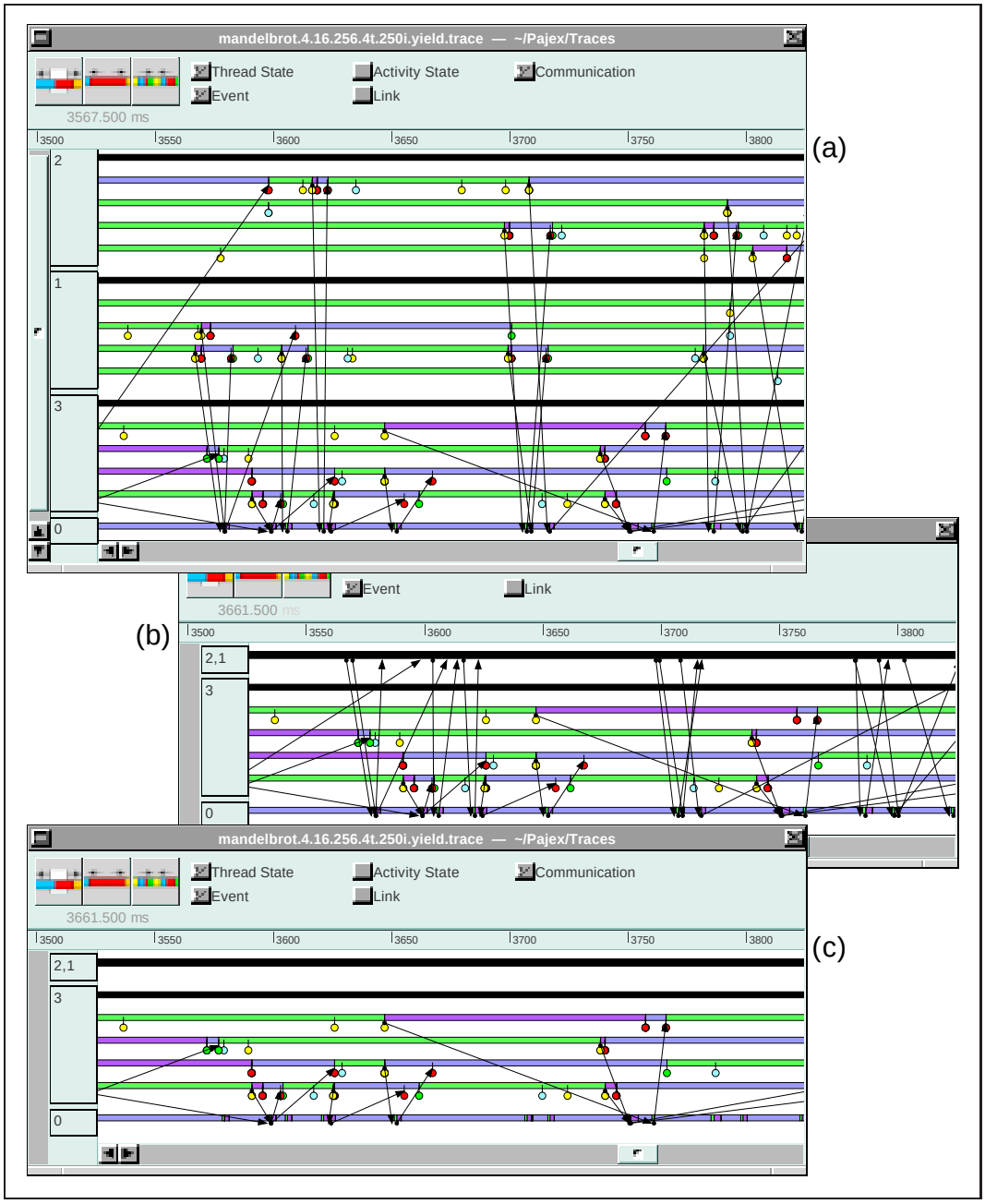

Figure 6.4 *Filtres de regroupement et de sélection d'entités*
(a) les événements du nœud 0 sont filtrés ; (b) les nœuds 1 et 2 sont groupés et les événements du groupe sont filtrés ; (c) les communications du groupe juste créé sont filtrées.

- Par conteneur : l'utilisateur choisit, pour des conteneurs individuels (nœud, fil d'exécution, etc.), quels types d'entités et quels « sous-conteneurs » seront

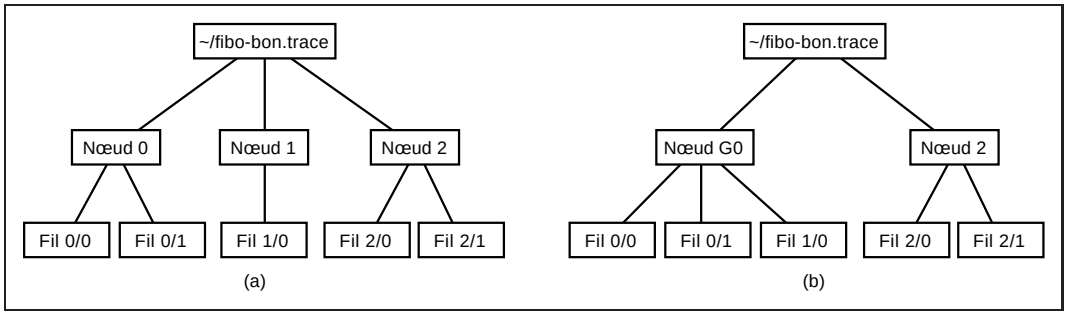

Figure 6.5 *Hiérarchie des conteneurs avant et après groupement*
Les nœuds 0 et 1 ont été groupés pour former le groupe G0. Les fils d'exécution des deux nœuds appartiennent au groupe, pour les modules après le filtre. Ce regroupement est virtuel, la nouvelle hiérarchie n'existe que pour les modules en aval du filtre ; la hiérarchie originale continue à exister, et est accédée par les autres filtres.

visualisés, ou de ne pas les visualiser du tout. Dans la figure 6.4, des entités sont progressivement enlevées : d'abord, les événements du nœud 0 (a) ; puis, les événements du groupe réunissant les nœuds 1 et 2 (b) et finalement, les communications qui impliquent ce groupe (c).

- Par valeur d'un attribut : généralisation du filtre par nom d'entité, ce filtre permet de choisir les entités à filtrer à partir de la valeur de leurs attributs. Par exemple, filtrer toutes les communications dont le « Nœud Source » vaut 0 et le « Nœud Destinataire » vaut 3 ; ou tous les états de durée inférieure à 100 μs.

6.4.3 Filtres de repositionnement

L'objectif d'un filtre de repositionnement est l'amélioration du placement des entités sur l'écran. Contrairement aux filtres présentés dans les sections précédentes, un filtre de repositionnement ne diminue pas la quantité de données à afficher. L'optimisation de l'emplacement des objets sur l'écran est intéressante pour rapprocher physiquement les représentations d'entités qui sont liées, pour faciliter l'étude de leur relation (afficher côte à côte deux nœuds qui communiquent beaucoup, par exemple). Une amélioration de ce type peut aussi viser à la réduction de l'espace écran occupé : par exemple il existe un filtre permettant de réutiliser l'espace écran correspondant aux conteneurs de courte durée (comme des fils d'exécution, des sémaphores, des verrous d'exclusion mutuelle) qui sont terminés. Un exemple de l'action de ce filtre est montré dans la figure 6.7, qui montre l'exécution d'un programme avec création et destruction de fils d'exécution très dynamique. Le filtre a permis de réutiliser l'espace des fils d'exécution terminés par des fils d'exécu-

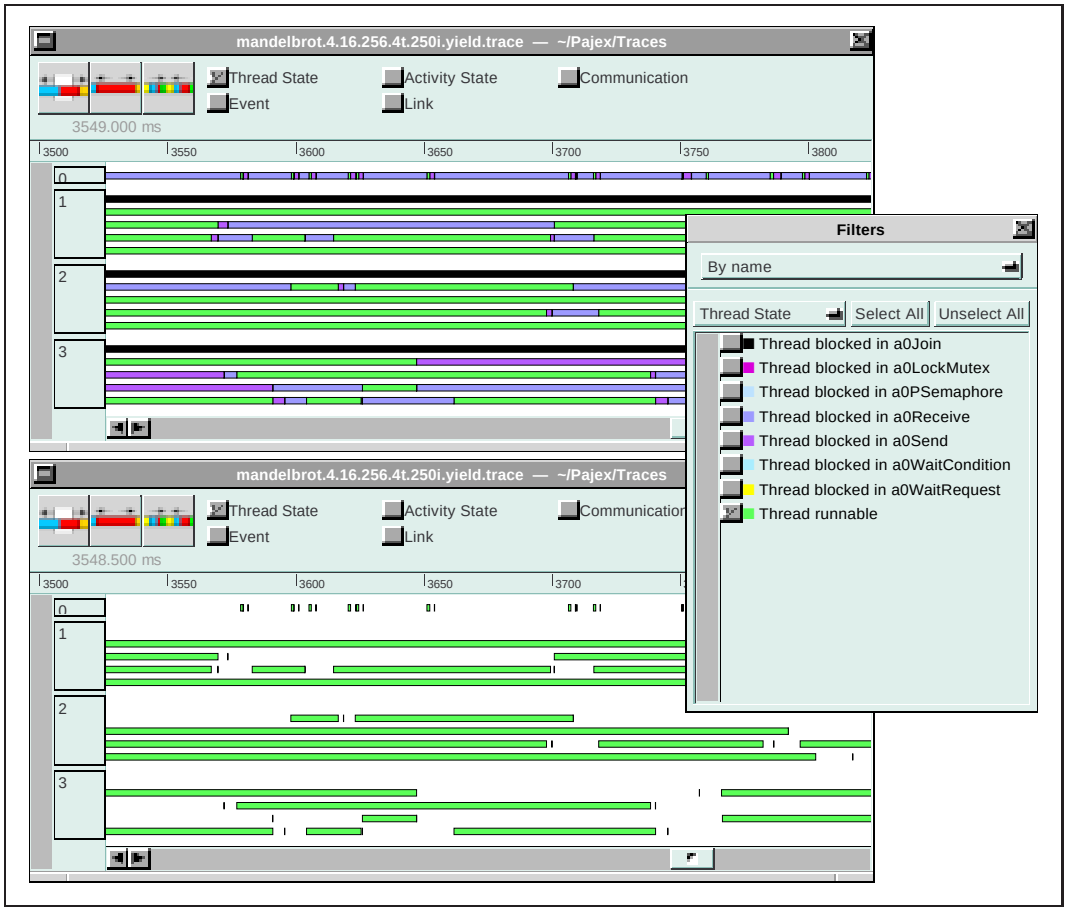

Figure 6.6 *Filtre de sélection par nom*
Avant le filtrage, tous les états de fils d'exécution sont affichés ; après, seuls les états activables sont conservés.

tion qui ont été créés après la terminaison des premiers. On peut voir que l'espace nécessaire à l'affichage du trace d'exécution a été grandement réduit.

6.5 Conclusion

En raison en particulier du comportement très dynamique des modèles de programmation visualisés, un soin tout particulier a été apporté pour permettre la visualisation d'applications susceptible de mettre en œuvre un grand nombre d'entités et de générer des fichiers de traces de très grande taille. Toutes les structures de données de l'environnement ont donc été conçues pour s'adapter, dans la limite de la taille de la machine utilisée pour la visualisation, à la taille du fichier de traces utilisé en entrée. Afin d'aider à l'identification des problèmes de performances se posant

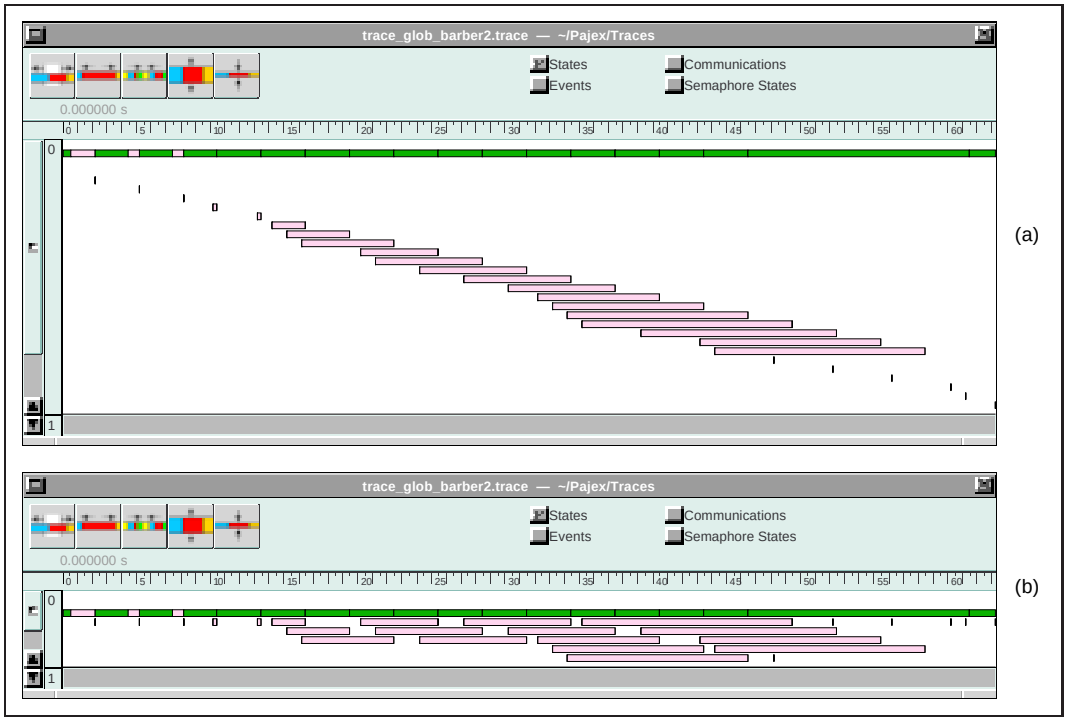

Figure 6.7 *Réutilisation de l'espace de fils d'exécution de courte durée*
(a) avant l'application du filtre ; (b) même programme, avec réutilisation de l'espace des fils d'exécution terminés.

dans les programmes de grande taille, Pajé offre la possibilité de naviguer interactivement entre plusieurs niveaux d'abstraction : un niveau d'abstraction élevé permet une vue d'ensemble d'une exécution et éventuellement la mise en évidence d'un problème concernant un sous-ensemble des entités mises en œuvre. Il est alors possible de concentrer la visualisation sur les entités posant apparemement problème et de les observer avec un grand niveau de détail. La technique employée repose sur le filtrage des données manipulées par l'environnement et en particulier de celles qui proviennent de la fenêtre d'observation et qui sont transmises aux modules de visualisation en réponse à leurs requêtes.

7

Conclusion

Cette thèse a permis de tracer les contours de ce qu'un environnement de visualisation de programmes parallèles à base de fils d'exécution doit être, d'identifier les problèmes que pose la mise en œuvre d'un tel environnement, de proposer des solutions et de les implémenter dans l'outil Pajé, dans le cadre de l'environnement de programmation parallèle ATHAPASCAN.

La principale difficulté du travail présenté a consisté à combiner des caractéristiques, toutes souhaitables mais a priori peu compatibles entre elles, au sein d'un même environnement. Ce résultat a pu être obtenu en faisant évoluer une architecture « classique » d'environnement de visualisation comme un graphe flot de données pour résoudre la contradiction apparente entre les contraintes d'extensibilité et d'interactivité que l'on s'était posé. Ce travail a abouti à la définition d'une structuration complexe des données calculées pour la visualisation en fenêtre d'observation, la structure définie étant d'une part facilement extensible et d'autre part permettant des recherches de données très efficaces. Une grande importance a également été accordée à la conception de composants indépendants de la sémantique du modèle de programmation tracé, afin de permettre la programmation générique de l'outil. Un petit langage de commande a été défini, qui permet d'instancier l'outil pour un modèle de programmation donné au moyen d'un petit programme inséré avant une trace à visualiser.

L'environnement Pajé est couramment utilisé dans le projet APACHE. Ainsi, il a été utilisé pour améliorer les performances d'une application de dynamique moléculaire, en permettant d'améliorer le recouvrement entre calcul et communications [19]. Il a également été utilisé pour étudier le comportement de programmes écrits

en ATHAPASCAN-1, ce qui a également permis de tester la généricité de l'outil.

De nouvelles utilisations de Pajé, sortant du cadre du débogage pour les performances, sont aussi envisagées. Dans la mesure où Pajé simplifie la compréhension d'exécutions parallèles, il semble qu'il puisse être également utilisé pour la recherche d'erreurs « classique ». Plusieurs travaux sont en cours pour faire de Pajé un composant important d'un environnement de débogage. L'objectif est de « connecter » Pajé avec un débogueur distribué [16] de telle sorte que le programmeur puisse disposer de la visualisation offerte par Pajé durant le débogage de son programme. Pour permettre cette « connection », il est nécessaire que la visualisation des exécutions paralléles puisse être faite en ligne, concurremment à l'utilisation du débogueur parallèle. En raison de l'intrusion importante introduite par le traçage et la visualisation, il est nécessaire d'effectuer cette mise au point durant une réexécution déterministe du programme mis au point [66]. En effet la réexécution déterministe préserve l'ordre causal des événements entre une exécution initiale faiblement perturbée par la prise d'une trace primaire et les réexécutions déterministes guidées par cette trace primaire. Le traceur de ATHAPASCAN ayant été modifié pour permettre la transmission des traces en ligne vers Pajé [64], il reste à implanter un lecteur de traces en ligne dans Pajé pour que la visualisation soit possible durant la phase de débogage.

D'autres perspectives concernent l'extension de Pajé pour augmenter la finesse du traçage dans les parties de l'exécution nécessitant une analyse plus fine : la visualisation sera utilisée pour séléctionner une partie de la trace et en déduire les informations permettant de guider le traceur à l'endroit à tracer avec plus de détails, dans une exécution ultérieure du programme parallèle.

La simulation générique peut également faire l'objet d'améliorations sensibles : la technique utilisée actuellement impose en effet le traçage d'un grand nombre d'événements, qui semble pouvoir être réduit. Par exemple le traçage du lancement d'une tâche ATHAPASCAN-1 implique l'enregistrement de quatre événements : décrémentation du nombre de tâches prêtes, transmission de la tâche au processeur virtuel ATHAPASCAN-1 qui va l'exécuter, réception de la tâche par ce processeur virtuel, passage du processeur virtuel à l'état actif. Il est envisageable de réduire ce nombre d'événements en augmentant l'expressivité du langage de définition d'événements utilisateur. Le prix à payer sera l'augmentation de la complexité du programme d'instanciation du simulateur. Seule l'expérience permettra d'établir le bon compromis entre expressivité, simplicité d'utilisation et taille des traces.

Bibliographie

[1] Arrouye (Y.). – *Environnements de visualisation pour l'évaluation des performances des systèmes parallèles : étude, conception et réalisation.* – Thèse de doctorat, INPG, Nov. 1995.

[2] Arrouye (Y.). – The EVE extensible visual environment, Mars 1995. LMC-IMAG, 100, rue des mathématiques, Campus, 38041 Grenoble, France.

[3] Aydt (R. A.). – *The Pablo Self-Defining Data Format*. – Rapport technique, University of Illinois at Urbana Champaign, 1992. `http://www-pablo.cs.uiuc.edu/Publications/Documents/documents.htm`.

[4] Bernard (P.-E.), Plateau (B.) et Trystram (D.). – Using Threads for developing Parallel Applications : Molecular Dynamics as a case study. *In : Parallel Numerics*, éd. par Trobec, pp. 3–16. – Gozd Martuljek, Slovenia, Sept. 1996.

[5] Blumofe (R. D.), Joerg (C. F.), Kuszmaul (B. C.), Leiserson (C. E.), Randall (K. H.), Zhou (Y. C. E.), Randall (K. H.) et Zhou (Y.). – Cilk : an efficient multithreaded runtime system. *ACM SIGPLAN Notices*, vol. 30, nN° 8, Août. 1995, pp. 207–216.

[6] Briat (J.), Ginzburg (I.) et Pasin (M.). – ATHAPASCAN-0B : un noyau exécutif parallèle. *Lettre du Calculateur Parallèle*, vol. 10, nN° 3, 1998, pp. 273–293.

[7] Briat (J.), Ginzburg (I.), Pasin (M.) et Plateau (B.). – Athapascan runtime : efficiency for irregular problems. *In : EURO-PAR'97 Parallel Processing*, éd. par Lengauer (C.) et al. pp. 591–600. – Springer. `http://www-apache.imag.fr`.

[8] Browne (S.), Dongarra (J.) et London (K.). – Review of performance analysis tools for mpi parallel programs. `http://www.cs.utk.edu/~browne/perftools-review`.

[9] Bull (J.). – A hierarchical classification of overheads in parallel programs. *In : Proceedings of First IFIP TC10 International Workshop on Software Engineering for Parallel and Distributed Systems*. pp. 208–219. – Chapman Hall.

[10] Cavalheiro (G. G. H.). – ATHAPASCAN-*1 : un mécanisme d'ordonnancement pour un environnement d'exécution parallèle*. – Thèse de doctorat, Laboratoire de Modélisation et de Calcul, Institut National Polytechnique de Grenoble, France, Octobre 1999.

[11] Cavalheiro (G. G. H.), Denneulin (Y.) et Roch (J.-L.). – A general modular specification for distributed schedulers. *In : Proceedings of Europar'98*, éd. par Springer Verlag (L. .). – Southampton, England, Sept. 1998.

[12] Cavalheiro (G. G. H.), Galilée (F.) et Roch (J.-L.). – Athapascan-1 : Parallel programming with asynchronous tasks. *In : Proceedings of the Yale Multi-threaded Programming Workshop*. – Yale, USA, Juin 1998.

[13] Clémençon (C.), Endo (A.), Fritscher (J.) et Mueller (A.). – Annai scalable run-time support for interactive debugging and performance analysis of large-scale parallel programs. *Lecture Notes in Computer Science*, vol. 1123, 1996, pp. 64–69. – `http://www.cscs.ch`.

[14] Clémençon (C.), Endo (A.), Fritscher (J.), Müller (A.) et Wylie (B. J. N.). – *Annai scalable run-time support for interactive debugging and performance analysis of large-scale parallel programs*. – Technical Report nN° CSCS-TR-96-04, CH-6928 Manno, Switzerland, Centro Svizzero di Calcolo Scientifico, Avr. 1996. `ftp://ftp.cscs.ch/pub/CSCS/techreports/1996/TR-96-04.ps.gz`.

[15] Couch (A. L.). – Categories and context in scalable execution visualization. *Journal of Parallel and Distributed Computing*, vol. 18, nN° 2, 1993, pp. 195–204.

[16] Cunha (J. C.), Lourenço (J.), Vieira (J.) et ao (B. M.). – A framework to support parallel and distributed debugging. *Lecture Notes in Computer Science*, vol. 1401, 1998, p. 708.

[17] de Kergommeaux (J. C.), Stein (B.) et Waille (P.). – Mise au point d'applications parallèles irrégulières. *In : ICaRE'97 Conception et mise en œuvre d'applications parallèles irrégulières de grande taille*, éd. par Barth (D.), de Kergommeaux (J. C.), Roch (J.-L.) et Roman (J.), chap. 13, pp. 267–282. – CNRS, Déc. 1997.

[18] de Oliveira Stein (B.) et Chassin de Kergommeaux (J.). – Environnement de visualisation de programmes parallèles basés sur les fils d'exécution. *In : Actes des* $9^{ièmes}$ *Rencontres Francophones du Parallélisme, RenPar9*, éd. par Trystram (D.). – Lausanne, Mai 1997.

[19] de Oliveira Stein (B.), de Kergommeaux (J. C.) et Bernard (P. É.). – Pajé, an interactive and visual tool for tuning multi-threaded parallel applications. *soumis à publication*, 1999.

[20] De Rose (L.), Zhang (Y.) et Reed (D. A.). – SvPablo : A multi-language performance analysis system. *Lecture Notes in Computer Science*, vol. 1469, Sept. 1998, pp. 352–355.

[21] Doreille (M.). – ATHAPASCAN-*1 : vers un modèle de programmation parallèle pour le calcul scientifique*. – Thèse de doctorat, Laboratoire de Modélisation et de Calcul, Institut National Polytechnique de Grenoble, France, Octobre 1999.

[22] Francioni (J. M.) et Jackson (J. A.). – Breaking the silence : Auralization of parallel program behavior. *Journal of Parallel and Distributed Computing*, vol. 18, nN° 2, Juin 1993, pp. 181–194.

[23] Galilée (F.). – ATHAPASCAN-*1 : interprétation distribuée du flot de données d'un un mécanisme de programme parallèle*. – Thèse de doctorat, Laboratoire de Modélisation et de Calcul, Institut National Polytechnique de Grenoble, France, Octobre 1999.

[24] Galilée (F.), Roch (J.-L.), Cavalheiro (G. G. H.) et Doreille (M.). – Athapascan-1 : On-line building data flow graph in a parallel language. *In : PACT'98*. – Paris, France, Oct. 1998. `http://www-apache.imag.fr`.

[25] Gautier (T.). – *Calcul formel et parallélisme : Conception du Système Givaro et Applications au Calcul dans les Extensions Algébriques*. – Thèse de PhD, Institut National Polytechnique de Grenoble, France, Juin 1996.

[26] Geist (G. A.), Heath (M. T.), Peyton (B. W.) et Worley (P. H.). – *PICL : A portable instrumented communications library*. – Rapport technique nN° TM-11130, Oak Ridge, Tennessee, Oak Ridge National Laboratory, 1990.

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

[28] Ginzburg (I.). – *Athapascan-0b : Intégration efficace et portable de multiprogrammation légère et de communications*. – Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France, Sept. 1997.

[29] Graham (S.), Kessler (P.) et McKusik (M.). – gprof : A call graph execution profiler. *In : Proceedings of the SIGPLAN'82 Symposium on Compiler Construction*. pp. 120–126. – ACM.

[30] Hackstadt (S.), Malony (A.) et Mohr (B.). – Scalable performance visualization for data-parallel programs. *In : Proceedings of the Scalable High Performance Computing Conference (SHPCC)*. – Knoxville, Tennessee, Mai 1994.

[31] Hayes (A. H.), Simmons (M. L.), Brown (J. S.) et Reed (D. A.). – *Debugging and Performance Tuning for Parallel Computing Systems*. – IEEE Computer Society, Juil. 1996.

[32] Heath (M.). – Recent Developments and Case Studies in Performance Visualization using ParaGraph. *In : Workshop on Performance Measurement and Visualization of Parallel Systems*. – Moravany, Czecho-Slovakia, Oct. 1992.

[33] Heath (M. T.) et Etheridge (J. A.). – Visualizing the Performances of Parallel Programs. *IEEE Trans. Softw. Eng.*, vol. 8, nN° 5, Mai 1991, pp. 29–39.

[34] Heath (M. T.) et Finger (J. E.). – *ParaGraph : A Tool for Vizualizing Performance of Parallel Programs*. – Rapport technique nN° ORNL/TM-11813, Oak Ridge National Laboratory, Oak Ridge, TN, 1991.

[35] Herrarte (V.) et Lusk (E.). – *Studying parallel program behavior with Upshot*. – Rapport technique nN° ANL-91/15, Mathematics and Computer Science Division, Argonne National Laboratory, Août. 1991. `http://www-fp.mcs.anl.gov/~lusk/upshot`.

[36] Hoare (C. A. R.). – Communicating sequential processes. *Communications of The ACM*, vol. XXI, nN° 8, 1978, pp. 666–677.

[37] Hofmann (R.). – Monitoring and evaluation of parallel and distributed systems. *In : Proc. European School on Parallel Programming Environments, ESPPE'96*, éd. par Chassin de Kergommeaux (J.) et Trystram (D.). pp. 135–153. – Institut d'Études Scientifiques Avancées de Grenoble. `ftp://ftp.imag.fr/pub/APACHE/ESPPE96`.

[38] Jacobson. (R.), Zhang (X.-J.), DuBose (R.) et B.W.Matthews. – Three-dimensional Structure of β-galactosidase from *E.coli*. *Nature*, vol. 369, 1986, pp. 761–766.

[39] Jézéquel (J. M.). – *Building a Global Time on Parallel Machines*. – Rapport technique nN° PI-513, Irisa, 1990.

[40] Kraemer (E.) et Stasko (J. T.). – The visualization of parallel systems : An overview. *Journal of Parallel and Distributed Computing*, vol. 18, nN° 2, Juin 1993, pp. 105–117.

[41] Kranzlmueller (D.), Grabner (S.) et Volkert (J.). – Debugging massively parallel programs with ATEMPT. *In : High-Performance Computing and Networking*, éd. par Liddell (H.), Colbrook (A.), Hertzberger (B.) et Sloot (P.). pp. 806–811. – Springer Verlag.

[42] Lam (M. S.). – Jade : a coarse-grain parallel programming language. *In : Proceedings of the Heterogeneous Network-Based Concurrent Computing Workshop*. Supercomputing Computations Research Institute, Florida State University. – Tallahassee, FL, Oct. 1991. Proceedings available via anonymous ftp from ftp.scri.fsu.edu in directory pub/parallel-workshop.91.

[43] Larking (D.) et Wilson (G.). – *Object-Oriented Programming and the Objective C Language*. – NeXT Software Inc., 1993. `http://www.gnustep.org/Documentation/ObjectivCBook.pdf`.

[44] Lopez (L.). – *The NAS Trace Visualizer (NTV) Rel. 1.2 User's Guide*, Sept. 1995. `http://science.nas.nasa.gov/Pubs/TechReports/NASreports/NAS-95-018/NAS-%95-018.ps`.

[45] Maillet (E.). – Issues in Performance Tracing with Tape/Pvm. *In : Proceedings of EuroPVM'95*. pp. 143–148. – HERMES (ISBN 2-86601-497-9).

[46] Maillet (E.). – *Traçage de logiciel d'applications parallèles : conception et ajustement de qualité*. – Thèse de PhD, Institut National Polytechnique de Grenoble, Sept. 1996.

[47] Maillet (E.) et Tron (C.). – On Efficiently Implementing Global Time for Performance Evaluation on Multiprocessor Systems. *Journal of Parallel and Distributed Computing*, vol. 28, Juil. 1995, pp. 84–93.

[48] Malony (A. D.), Reed (A.) et Wijshoff (H.). – Performance Measurement Intrusion and Perturbation Analysis. *IEEE Transactions on parallel and distriuted systems*, vol. 3, nN° 4, Juil. 1992.

[49] Miller (B. P.). – What to draw ? when to draw ? an essay on parallel program visualization. *Journal of Parallel and Distributed Computing*, vol. 18, nN° 2, Juin 1993, pp. 265–269.

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

[51] National HPCC Software Exchange (NHSE). – Parallel tools library software catalog. `http://www.nhse.org/rib/repositories/ptlib/catalog/`.

[52] NeXT Computer. – *OpenStep specification*, Oct. 1994. `http://www.gnustep.org/Resources/OpenStepSpec/OpenStepSpec.ps.gz`.

[53] Nickolayev (O. Y.), Roth (P. C.) et Reed (D. A.). – Real-time statistical clustering for event trace reduction. *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, nN° 2, Summer 1997, pp. 144–159.

[54] Pillet (V.), Labarta (J.), Cortes (T.) et Girona (S.). – PARAVER : A tool to visualise and analyze parallel code. *In : Proceedings of WoTUG-18 : Transputer and occam Developments*. pp. 17–31. – Amsterdam, Avr. 1995. `ftp://ftp.ac.upc.es/pub/reports/CEPBA/1995/UPC-CEPBA-95-03.ps.Z`.

[55] Plateau (B.) et al. – *Présentation d'*APACHE. – Rapport APACHE nN° 1, Grenoble, IMAG, Oct. 1993. `http://www-apache.imag.fr`.

[56] Reed (D.), Shields (K.), Scullin (W.), Tavera (L. F.) et Elford (C.). – Virtual reality and parallel systems performance analysis. *IEEE Computer*, Nov. 1995.

[57] Reed (D. A.). – Experimental Performance Analysis of Parallel Systems : Techniques and Open Problems. *In : Proceedings of the 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, éd. par Haring (G.) et Kotsis (G.). pp. 25–51. – Springer-Verlag.

[58] Reed (D. A.), Aydt (R. A.), Madhyastha (T. M.), Noe (R. J.), Shields (K. A.) et Schwartz (B. W.). – *An Overview of the Pablo Performance Analysis Environment*. – Rapport technique, Department of Computer Science, University of Illinois, Urbana, Illinois 61801, 1992.

[59] Reed (D. A.) et al. – *An Overview of the Pablo Performance Analysis Environment*. – Rapport technique, Department of Computer Science, University of Illinois, 1992.

[60] Reed (D. A.) et al. – Scalable Performance Analysis : The Pablo Performance Analysis Environment. *In : Proceedings of the Scalable Parallel Libraries Conference*, éd. par Skjellum (A.). pp. 104–113, 1993. – IEEE Computer Society.

[61] Reed (D. A.), Gardner (M. J.) et Smirni (E.). – Performance visualization : 2-D, 3-D, and beyond. *In : Proceedings of the International Computer Performance and Dependability Symposium (IPDS'96)*. – Urbana, Illinois, Sept. 1997. `http://www-pablo.cs.uiuc.edu/Publications/Papers/IPDS96.ps.gz`.

[62] Reed (D. A.) et Ribler (R. L.). – Performance analysis and visualization. *In : The Grid : Blueprint for a New Computing Infrastructure*, éd. par Foster (I.) et Kesselman (C.), chap. 15. – Morgan-Kaufman Publishers, Juil. 1998. `http://vibes.cs.uiuc.edu/Publications/Papers/GridChapter.ps.gz`.

[63] Reed (D. A.), Shields (K. A.), Scullin (W. H.), Tavera (L. F.) et Elford (C. L.). – Virtual reality and parallel systems performance analysis. *Computer*, vol. 28, nN° 11, Nov. 1995, pp. 57–67.

[64] Ribette (F.). – Traçage en ligne d'applications parallèles à base de processus légers. – Rapport de stage DESS Informatique, Université Louis Pasteur, Strasbourg, Sept. 1999.

[65] Ribler (R.), Mathur (A.) et Abrams (M.). – *Visualizing and Modeling Categorical Time Series Data*. – Technical Report nN° TR-95-08, Virginia Polytechnic Inst. and State University, Juin 19, 1995. `ftp://ftp.cslab.vt.edu/pub/local/reports/95/TR-95-08.ps.gz`.

[66] Ronsse (M.), Chassin de Kergommeaux (J.) et De Bosschere (K.). – Execution replay for an mpi-based multi-threaded runtime system. – ParCo'99, Août. 1999. Accepted for presentation.

[67] Stein (B.) et de Kergommeaux (J. C.). – Interactive visualization environment of multi-threaded parallel programs. *In : Parallel Computing : Fundamentals, Applications and New Directions*, éd. par D'Hollander (E.), Joubert (G.), Peters (F.) et Trottenberg (U.). – Amsterdam, 1998.

[68] Stevens (R. W.). – *UNIX Network Programming*. – Englewood Cliffs, NJ, Prentice Hall, 1990, *Software Series*.

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

[70] Wu (C. E.) et Franke (H.). – *UTE User's Guide for IBM SP Systems*, 1995. `http://www.research.ibm.com/people/w/wu/uteug.ps.Z`.

[71] Wylie (B. J. N.) et Endo (A.). – Annai/PMA multi-level hierarchical parallel program performance engineering. *In : Proceedings 1st Int'l Work. on High-level Programming Models and Supportive Environments (HIPS'96, Honolulu, USA)*. pp. 58–67. – IEEE Computer Society Press.

[72] Yan (J.), Sarukkai (S.) et Mehra (P.). – Performance measurement, visualization and modeling of parallel and distributed programs using the AIMS toolkit. *Software—Practice and Experience*, vol. 25, nN° 4, Avr. 1995, pp. 429–461.

[73] Yan (J. C.). – Performance tuning with AIMS — an automated instrumentation and monitoring system for multicomputers. *In : Proc. of the Twenty-Seventh Annual Hawai Conference on System Sciences*. pp. 625–633. – IEEE Computer Society Press.

[74] Yan (J. C.) et Schmidt (M. A.). – Constructing space-time views from fixed size trace files – getting the best of both worlds. *In : Parallel Computing : Fundamentals, Applications and New Directions, Proceedings of the Conference ParCo'97, 19-22 September 1997, Bonn, Germany*, éd. par D'Hollander (E. H.), Joubert (G. R.), Peters (F. J.) et Trottenberg (U.). pp. 633–640. – Amsterdam, Fév. 1998.

[75] Zaki (O.), Lusk (E.), Gropp (W.) et Swider (D.). – Toward scalable performance visualization with Jumpshot. *The International Journal of High Performance Computing Applications*, vol. 13, nN° 3, 1999, pp. 277–288. – à paraitre. `ftp://info.mcs.anl.gov/pub/tech_reports/reports/P763.ps.Z`.

[76] Zhao (Q. A.) et Stasko (J. T.). – *Visualizing the Execution of Threads-based Parallel Programs*. – Rapport technique nN° GIT-GVU-95-01, Georgia Institute of Technology, 1995.

Résumé

Cette thèse s'est déroulée au sein du projet[a] APACHE dont l'objectif est l'étude de l'ensemble des aspects liés à la mise en œuvre efficace et portable d'applications irrégulières et dont les études sont concrétisées par l'environnement ATHAPASCAN. Dans l'environnement ATHAPASCAN le « débogage pour les performances » est basé sur le traçage logiciel des exécutions des applications parallèles suivi de l'analyse des traces et de la visualisation des exécutions tracées. L'objectif de la thèse était de fournir aux programmeurs un outil de visualisation les aidant à identifier les « erreurs de performances » de leurs programmes en leur donnant une représentation aussi claire que possible de l'exécution de ces programmes. La principale contribution de la thèse est la conception et la réalisation d'un outil appelé Pajé combinant les trois propriétés essentielles d'interactivité, d'extensibilité et d'aptitude au passage à l'échelle. L'extensibilité permet de prendre en compte l'absence de stabilisation des modèles de programmation parallèles et d'offrir la possibilité d'ajouter à Pajé des visualisations non envisagées lors de sa conception. Elle est assurée par une architecture en graphe de modules génériques, communiquants par des protocoles bien spécifiés. L'interactivité donne au programmeur le contrôle sur la visualisation par des actions telles que déplacement dans le temps ou inspection du contenu des objets visualisés, etc. Pour limiter le volume de données qu'elle implique de conserver en mémoire, une structure de données appelée fenêtre de visualisation a été définie ainsi que les algorithmes permettant de la faire glisser efficacement dans le temps. L'aptitude au passage à l'échelle est liée à la capacité de représenter un nombre potentiellement important d'objets graphiques — processus légers, communications, tâches, etc. — évoluant dynamiquement. Elle est essentiellement assurée en facilitant la visualisation à différents niveaux d'abstraction, en sorte que le passage d'un niveau à un autre simule une action de zoom.

Mots clés : visualisation de programmes parallèles, mesure de performances, processus légers, interactivité, extensibilité et passage à l'échelle

Abstract

The research described in this dissertation was performed within the APACHE team[b] whose aim is to study all the issues raised by combining efficiency and portability in the implementation of irregular applications. The work done by the APACHE project is integrated in the ATHAPASCAN software environment. In ATHAPASCAN, performance debugging is based on software tracing of the executions of parallel applications, followed by trace analysis and visualisation of the traced executions. The aim of this thesis was to provide programmers with a visualisation tool helping them to identify the performance errors of their programs by providing them with the clearest possible representation of the execution of these programs. The most important claim of this thesis is the design and implementation of a visualisation tool called Pajé combining the three most important characteristics of such visualisations tools: extensibility, interactivity and scalability. Extensibility is necessary to cope with the lack of parallel programming standard and ease the implementation of non foreseen visualizations in Pajé. It is supported by the architecture of Pajé as a graph of generic components with clearly defined communication protocols. Interactivity allows programmers to control the visualisation by moving in time or inspecting the contents of the displayed objects. To limit the amount of data that need to be managed in memory to implement interactivity, a data structure called observation window was defined together with the algorithms to move it efficiently in time. Scalability is related to the possibility of representing a potentially important number of graphical objects — threads, communications, tasks, etc.— evolving dynamically. It is mainly supported by allowing the visualisation at several levels of abstraction, such that moving from one level to another simulates zooming.

Keywords: parallel program visualization, performance debugging, threads, interactivity, scalability, extensibility

[a] CNRS-INPG-INRIA-UJF

[b] sponsored by CNRS, INPG, INRIA and UJF