De la collecte de trace à la prédiction du comportement d'applications parallèles

Alexis Colin

▶ **To cite this version:**

Alexis Colin. De la collecte de trace à la prédiction du comportement d'applications parallèles. Informatique. Institut Polytechnique de Paris, 2022. Français. NNT : 2022IPPAS020 . tel-03954260

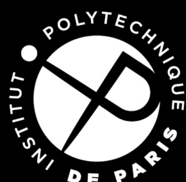
HAL Id: tel-03954260

https://theses.hal.science/tel-03954260

Submitted on 24 Jan 2023

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

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



NNT : 2022IPPAS020

Thèse de doctorat

De la collecte de trace à la prédiction du comportement d'applications parallèles

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom SudParis

École doctorale n°626, École doctorale de l'Institut Polytechnique de Paris (EDIPP)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Palaiseau, France, le 28 Novembre 2022, par

ALEXIS COLIN

Composition du Jury :

Camille Coti Professeure, Université de Québec	Rapporteuse
Samuel Thibault Professeur des universités, Université de Bordeaux	Rapporteur
Amel Bouzeghoub Professeure, Télécom SudParis	Présidente du jury
Patrick Carribault Chargé de recherche, CEA/DAM	Examinateur
Arnaud Legrand Directeur de recherche, Inria Grenoble	Examinateur
Denis Conan Maître de conférences HDR, Télécom SudParis	Directeur de thèse
François Trahay Maître de conférences HDR, Télécom SudParis	Co-encadrant de thèse

Résumé

La complexité des architectures parallèles modernes rend le développement de programmes à destination des serveurs et des supercalculateurs ardu. Pour exploiter les ressources de ces machines, les développeurs ont recours à des modèles de programmation spécifiques qui permettent d'exprimer le potentiel de parallélisme des programmes. Ces modèles de programmation introduisent un certain nombre d'abstractions telles que la notion de tâche et de communication. Ces abstractions, qui s'ajoutent à celles fournies par le système d'exploitation, permettent de rendre les programmes plus indépendants des machines qui les exécutent. Les modèles de programmation parallèle sont mis en œuvre par des supports d'exécution, ou *runtimes*, qui s'insèrent dans la pile logicielle entre le système d'exploitation et les programmes. Ces *runtimes* peuvent par exemple prendre la forme de bibliothèques de communication, d'ordonnancement de tâches ou de gestion mémoire. Leur rôle est de permettre à chaque programme d'exploiter pleinement les capacités de la machine qui l'exécute.

Pour adapter les programmes aux différentes machines parallèles, les *runtimes* doivent prendre des décisions qui ont un impact direct sur les performances. Pour cela, les *runtimes* utilisent généralement des heuristiques afin d'anticiper le comportement futur du programme à partir de données recueillies au cours de l'exécution. Ces heuristiques ont en particulier l'avantage d'être peu coûteuses et faciles à mettre en œuvre. Malheureusement, les heuristiques peuvent se tromper et mener à une dégradation des performances du programme.

Dans cette thèse, nous proposons un nouveau moyen de prédire le comportement futur d'une application. Nous développons PYTHIA, un oracle générique pouvant être utilisé au sein d'un *runtime* afin de prédire le comportement futur d'un programme et d'améliorer ses performances. Dans un premier temps, nous nous intéressons à la représentation d'une trace d'exécution sous la forme d'une grammaire. Nous montrons tout d'abord comment utiliser une grammaire pour représenter une trace d'exécution et nous décrivons quatre contraintes permettant d'utiliser une telle grammaire pour représenter la structure de l'exécution d'un programme. Ensuite, nous développons un algorithme capable de construire à la volée une telle grammaire à partir d'une séquence d'événements soumis à PYTHIA par le *runtime*. Nous implémentons cet algorithme et montrons que son surcoût est négligeable sur la durée d'exécution d'un panel de treize applications.

Dans un second temps, nous montrons comment utiliser une grammaire représentant une trace d'exécution pour prédire le comportement futur d'un programme. Nous introduisons la notion de « point de progression », qui permet à la fois de suivre l'avancement de l'exécution d'un programme dans une grammaire et de prédire son comportement du futur. Nous montrons comment explorer sous la forme d'un arbre probabilisé les potentielles actions à venir du programme. Ensuite, nous présentons comment utiliser les points de progression fournis par notre oracle pour enrichir ses prédictions avec des données adaptées aux besoins spécifiques des *runtimes*. Nous évaluons les performances et la correction des prédictions de notre oracle sur un panel de treize applications. L'évaluation montre que PYTHIA est capable de prédire précisément et efficacement le comportement futur de la plupart des programmes testés. Nous faisons aussi la démonstration de l'utilisabilité de PYTHIA en implémentant une stratégie de parallélisme adaptatif au sein d'un *runtime* OpenMP. Grâce aux prédictions de PYTHIA, cette stratégie permet d'optimiser les petites régions parallèles d'un programme.

Abstract

The complexity of modern parallel architectures makes the development of programs for servers and supercomputers difficult. In order to exploit the resources of these machines, developers use specific programming models that allow them to express the potential of parallelism in their programs. These programming models introduce abstractions such as the notion of task and communication. These abstractions, which are added to those provided by the operating system, make programs more independent of the machines that run them. Parallel programming models are implemented by runtime systems, which are inserted into the software stack between the operating system and the programs. These runtimes can, for example, take the form of communication, task scheduling or memory management libraries. Their role is to allow each program to fully exploit the capabilities of the machines that run it. To do this, runtimes exploit the abstractions of the programming models to adapt the programs to the machines that run them.

To adapt the programs to the different parallel machines, runtimes must take decisions that have direct impact on the performance. To make these decisions, runtimes generally use heuristics to anticipate the future behavior of the program from data collected during the execution. These heuristics have the particular advantage of being inexpensive and easy to implement. Unfortunately, heuristics can be wrong and can lead to a degradation of the performance.

In this thesis, we propose a new mechanism for predicting the future behavior of an application. We develop PYTHIA, a generic oracle that can be used within a runtime to predict the future behavior of a program and improve its performance. First, we focus on the representation of an execution trace under the form of a grammar. We first show how to use a grammar to represent an execution trace and we describe four constraints allowing to use such a grammar to represent the structure of the execution of a program. We then develop an algorithm capable of building such a grammar on the fly from a sequence of events submitted to PYTHIA by a runtime. We implement this algorithm and show that its overhead is negligible on the execution time of a panel of thirteen applications.

In a second step, we show how to use a grammar representing a trace of a program execution to predict its future behavior. We introduce the notion of "state of progress" that allows both to follow in a grammar the progress of the execution of a program and to predict its future behavior. We show how to explore the potential future actions of the program in the form of a probabilized tree. We then explain how to use the state of progress provided by our oracle to extend its predictions with data adapted to the specific needs of the runtimes. We evaluate the performance and the correctness of the predictions of our oracle on a panel of thirteen applications. The evaluation shows that PYTHIA is able to accurately and efficiently predict the future behavior of most of the tested programs. We also demonstrate the usability of PYTHIA by using it to implement an adaptive parallelism strategy within an OpenMP runtime. Thanks to PYTHIA, this strategy allows to optimize the small parallel regions of a program.

Table des matières

Table des matières vi

Table des figures viii

Table des tableaux ix

1 Introduction 1
1.1 Contexte et objectifs . . . 2
1.2 Problématique . . . 2
1.3 Contributions . . . 3
1.3.1 Réduction de traces à la volée sous forme de grammaire . . . 3
1.3.2 Utilisation d'une grammaire pour prédire le comportement futur d'un programme . . . 3
1.3.3 Résultats . . . 4
1.4 Organisation du document . . . 4

2 Supports d'exécution pour la programmation parallèle 6
2.1 Architectures matérielles parallèles . . . 7
2.1.1 Grappes de calcul . . . 7
2.1.2 Processeurs multicœurs . . . 8
2.1.3 Architectures à accès mémoire non uniforme (NUMA) . . . 8
2.1.4 Architectures hétérogènes . . . 9
2.1.5 Conclusion . . . 10
2.2 Programmation parallèle . . . 11
2.2.1 Programmation en mémoire partagée . . . 11
2.2.2 Programmation hétérogène . . . 15
2.2.3 Programmation en mémoire répartie . . . 15
2.3 Supports d'exécution et prises de décision . . . 16
2.3.1 Heuristiques . . . 17
2.3.2 Décision à l'aide d'annotations dans le code . . . 18
2.3.3 Décision à l'aide de modèles de performances . . . 18
2.3.4 Utilisation d'oracles . . . 19
2.4 Conclusion . . . 19

3 PYTHIA-RECORD : réduction de trace d'exécution à la volée pour la prévision 21
3.1 Enregistrer le déroulement de l'exécution d'un programme . . . 22
3.1.1 Échantillonnage . . . 23

3.1.2 Appels de fonction depuis le code source ou à la compilation . . . 23
3.1.3 Utilisation de sondes . . . 23
3.1.4 Injection de code machine . . . 24
3.1.5 Pré-chargement de bibliothèque dynamique avec `LD_PRELOAD` . . . 24
3.2 Représentation structurée d'une trace d'exécution . . . 26
3.2.1 Stockage séquentiel des événements . . . 26
3.2.2 Découverte de redondances *a posteriori* . . . 26
3.2.3 Réduction à la volée sous forme de grammaire . . . 27
3.2.4 Discussion . . . 27
3.3 Représentation du comportement d'un programme sous forme d'une grammaire . . . 28
3.3.1 Représentation d'une trace par une grammaire . . . 28
3.3.2 Contraintes sur la grammaire . . . 30
3.4 Réduction à la volée d'une trace d'exécution . . . 32
3.4.1 Principe de l'algorithme de réduction . . . 32
3.4.2 Notations . . . 33
3.4.3 Algorithme de réduction . . . 34
3.4.4 Implémentation . . . 36
3.5 Conclusion . . . 37

4 PYTHIA-PREDICT : prévision du comportement futur d'un programme 40
4.1 Suivi de l'exécution du programme : le cas de deux exécutions identiques . . . 41
4.1.1 Suivi de l'exécution du programme par reconstruction de la trace par substitutions . . . 41
4.1.2 Suivi de l'avancement de l'exécution sans décompression . . . 43
4.1.3 Représentation de l'état d'avancement du programme . . . 45
4.2 Comparaison d'exécutions avec des variations . . . 47
4.2.1 Adaptation de PYTHIA face à une variation du comportement du programme . . . 47
4.2.2 Recherche de séquences au sein d'une trace réduite . . . 48
4.2.3 Point de progression représentant une connaissance incomplète de l'avancement du programme 48
4.2.4 Discussion . . . 51
4.3 Prévoir le comportement futur du programme . . . 51
4.3.1 Prévision des événements à venir . . . 51
4.3.2 Calcul de la probabilité de chaque événement . . . 52
4.3.3 Enrichissement des prévisions . . . 54
4.4 Conclusion . . . 55

5 Évaluation de l'utilisation de PYTHIA au sein d'un runtime 56
5.1 Environnement expérimental . . . 57
5.1.1 Machines . . . 57
5.1.2 Applications et jeux de données . . . 57
5.1.3 Supports d'exécution . . . 58
5.2 Évaluation du surcoût de PYTHIA-RECORD . . . 59
5.3 Évaluation des performances de PYTHIA-PREDICT . . . 60
5.3.1 Correction des prédictions de PYTHIA-PREDICT . . . 61
5.3.2 Coût des prévisions de PYTHIA-PREDICT . . . 62
5.4 Évaluation de l'utilisabilité de PYTHIA . . . 63

5.4.1 Modifications apportées à GNU OpenMP . . . 63
5.4.2 Modifications apportées à Lulesh . . . 64
5.4.3 Impact de Pythia-OpenMP sur la durée d'exécution de Lulesh . . . 65
5.5 Conclusion . . . 68

6 Conclusion et perspectives **69**
6.1 Contexte et problématique de la thèse . . . 69
6.2 Contributions de la thèse . . . 70
6.2.1 Réduction à la volée d'une trace d'exécution sous la forme d'une grammaire . . . 70
6.2.2 Utilisation d'une grammaire pour prédire le comportement futur d'un programme . . . 71
6.3 Perspectives et travaux futurs . . . 71
6.3.1 Pistes d'amélioration de PYTHIA . . . 71
6.3.2 Généralisation d'un langage de requête pour l'oracle, annotation dynamique de la grammaire et programmation événementielle . . . 72
6.3.3 Analyse de fichiers de journalisation . . . 72
6.3.4 Recherche de bogues au sein d'applications . . . 73

Bibliographie **78**

A API de Pythia **79**

B Liste des fonctions interceptées par le *runtime* MPI-OpenMP modifié **81**

Table des figures

1.1 Vision d'ensemble de PYTHIA et de ses deux parties PYTHIA-RECORD et PYTHIA-PREDICT. 3

2.1 Schéma de principe d'une grappe de calcul. 7
2.2 Schéma de principe d'un processeur multicœur et de ses caches mémoire. 8
2.3 Schéma de principe d'une architecture NUMA. 9
2.4 Schéma de principe d'une architecture hétérogène. 10
2.5 Exemple d'utilisation de pthread pour accélérer une boucle dans un programme en C. 12
2.6 Exemple d'exécution d'un code OpenMP sur deux *threads*. 13
2.7 Graphe de dépendance des tâches constituant une factorisation de Cholesky. 14
2.8 Exemple d'exécution d'un code MPI avec trois processus. 16

3.1 Intégration de PYTHIA-RECORD dans l'outil PYTHIA. 22
3.2 Exemples de techniques d'instrumentation par injection de code. 25
3.3 Utilisation d'une bibliothèque dynamique. 25
3.4 Interception d'appels à une bibliothèque dynamique à l'aide de `LD_PRELOAD`. 25
3.5 Exemple de représentation de la séquence « $abcab$ » par une grammaire. 29
3.6 Exemple de représentation de la « $abbcbcab$ » par une grammaire. 29
3.8 Écriture dans un fichier de la grammaire représentant la séquence *abbcbcab*. 37
3.9 Représentation en mémoire de la grammaire représentant la séquence *abbcbcab*. 38

4.1 Intégration de PYTHIA-PREDICT dans l'outil PYTHIA. 41
4.2 Reconstruction par substitutions de la séquence « $abcabdabcabdabc$ » depuis la grammaire la représentant. 42
4.3 Suivi de l'exécution d'un programme générant la séquence « $abcab$ » par substitutions partielles. . . . 42
4.4 Intuition de la reconstruction d'une trace par parcours d'une grammaire. 43
4.5 Chemin représentant la troisième occurrence du symbole b de la séquence « $ababccabcab$ ». 43
4.6 Parcours d'une trace représentée par une grammaire sans décompression. 44
4.7 Parcours d'une grammaire contenant une boucle. 45
4.8 Recherche de toutes les occurrences de la séquence « $abda$ » au sein de la grammaire représentant la trace « $abcabdabe$ ». 49
4.9 Exemple d'un point de progression représentant les deux occurrences du symbole b dans la séquence ab au sein de la trace « $abcbcdbcabc$ ». 50
4.10 Deux points de progression sont nécessaires pour lister toutes les apparitions de la séquence « ab » dans la trace « $bcbdabcabd$ ». 50
4.11 Recherche de la séquence « $abcbcb$ » dans la trace « $abcbcbcbcbc$ ». 51
4.12 Recherche de la séquence « $cbcb$ ». 51

4.13 Association contextualisée de valeurs aux points de progression. 52
4.14 Arbre des possibles correspondant à l'exemple de la figure 4.13. 53
4.15 Calcul du nombre d'occurrences de points de progression. 53
4.16 Association contextualisée de valeurs aux points de progression. 54

5.1 Grammaire générée à partir des appels à MPI observés au sein d'un rang MPI de l'application BT avec la taille de problème *large*. (Le préfix `MPI_` des fonctions a été omis à des fins de concision.) . . . 60
5.2 Justesse de prédictions de PYTHIA sur Paravance. 62
5.3 Coût des prédictions de PYTHIA-PREDICT sur Paravance avec le jeu de données *large*. 63
5.4 Temps d'exécution de Lulesh en fonction de la taille du problème avec 24 *threads* sur Pudding. 66
5.5 Temps d'exécution de Lulesh en fonction de la taille du problème avec 16 *threads* sur Pixel. 66
5.6 Temps d'exécution de Lulesh en fonction du nombre de *threads* disponibles pour une taille de problème de 30 sur Pudding. 67
5.7 Temps d'exécution de Lulesh en fonction du nombre de *threads* disponibles pour une taille de problème de 30 sur Pixel. 67

A.1 Détail de l'API de PYTHIA-RECORD. 79
A.2 Détail de l'API de PYTHIA-PREDICT. 80

Liste des tableaux

5.1 Mesure des performances de PYTHIA-RECORD 60

B.1 Fonctions du *runtime* GNU OpenMP interceptées par Pythia-MPI+OpenMP 81
B.2 Fonctions du *runtime* MPI interceptées par Pythia-MPI+OpenMP 82

Chapitre 1

Introduction

Sommaire

1.1 Contexte et objectifs **2**
1.2 Problématique **2**
1.3 Contributions **3**
1.3.1 Réduction de traces à la volée sous forme de grammaire 3
1.3.2 Utilisation d'une grammaire pour prédire le comportement futur d'un programme 3
1.3.3 Résultats 4
1.4 Organisation du document **4**

En 1965, Gordon Moore observe que la puissance des ordinateurs croît de manière exponentielle. Cette tendance s'est maintenue depuis avec un doublement des performances des processeurs tous les 18 mois environ. Pendant des décennies, cette amélioration a été rendue possible par une gravure de plus en plus fine des puces électroniques. Aujourd'hui, malheureusement, cette amélioration des performances n'est plus permise par la simple amélioration de la vitesse des cœurs de calcul [65]. Pour continuer d'améliorer les performances, il est désormais nécessaire de complexifier les machines en leur permettant d'effectuer toujours plus de calculs en parallèle. Cette augmentation des capacités de parallélisation des machines peut être obtenu *via* diverses architectures matérielles : processeurs multicœurs, grappes de calculs, architectures NUMA, etc.

Écrire des programmes exploitant efficacement des machines parallèles n'est pas une chose aisée. Les machines parallèles sont complexes et soulèvent de nombreuses questions : quelles parties du code paralléliser ? comment partager les ressources ? comment synchroniser la lecture et la modification des données en mémoire ? Toutes ces questions posent des problèmes à la fois matériels et logiciels. Aux traditionnelles questions de portabilité des programmes d'un environnement logiciel à un autre, la diversité des architectures parallèles soulève la difficulté d'écrire des programmes capables de s'exécuter sur des machines mettant en œuvre des technologies très différentes. Optimiser un programme afin qu'il tire le meilleur parti des capacités d'une machine est une tâche ardue. Il s'agit d'écrire des programmes capables de s'exécuter efficacement sur des machines parfois très différentes. Cette capacité d'un programme à s'exécuter de manière efficace sur des machines différentes est appelée la « portabilité des performances » [58]. Cette thèse vise à développer un oracle permettant d'améliorer les supports d'exécution en charge de la portabilité des performances.

Dans ce chapitre, nous présentons brièvement le contexte et les objectifs de la thèse (section 1.1) puis la problématique considérée (section 1.2). Nous présentons ensuite nos contributions (section 1.3) et terminons le chapitre en présentant l'organisation de ce document (section 1.4).

1.1 Contexte et objectifs

Afin de garantir la portabilité des performances d'un programme, les développeurs ont recours à différents modèles de programmation. Ces modèles de programmation offrent des outils adaptés pour exprimer le potentiel de parallélisme d'une application. Ils introduisent un certain nombre d'abstractions, telles que la notion de tâche ou de communication. Ces nouvelles abstractions qui s'ajoutent à celles fournies par le système d'exploitation permettent aux développeurs et aux développeuses de penser leurs programmes à un plus haut niveau d'abstraction. Par ailleurs, en utilisant ces abstractions, ils rendent leurs programmes plus indépendants de la machine qui l'exécute.

Les modèles de programmation utilisés par les concepteurs d'applications parallèles sont mis en œuvre par des supports d'exécution, ou *runtimes*, tels que des bibliothèques de communication [30, 6, 67], d'ordonnancement de tâches [4, 36], ou de gestion mémoire [2, 57, 50]. C'est aux *runtimes* que revient dès lors d'adapter chaque programme à la machine qui l'exécute afin de lui fournir les meilleures performances possibles. Pour un *runtime*, cette adaptation du programme à la machine consiste à prendre de manière algorithmique des décisions sur la façon dont un programme doit s'exécuter : par exemple, dans quelle ordre exécuter les différentes tâches ? sur quel nœud NUMA allouer une page mémoire ? sur quel cœur du processeur placer un *thread* ?

Ces décisions prises par les supports d'exécution peuvent avoir un impact significatif sur les performances des programmes. Pour prendre ses décisions, un *runtime* utilise des heuristiques afin d'anticiper le comportement futur d'un programme à partir de données recueillies au cours de l'exécution. Par exemple, la politique d'allocation *first touch* [46] utilisée par GNU/Linux alloue chaque page mémoire utilisée par le programme sur le nœud NUMA exécutant le premier *thread* qui y accède. Cette heuristique suppose que ce *thread* continuera probablement à utiliser cette page mémoire à l'avenir et vise à lui garantir les meilleurs temps d'accès possibles. Cependant, cette heuristique peut se tromper : par exemple, si le *thread* chargeant les données depuis le disque n'est pas celui qui les exploite, les accès mémoire sont optimisés pour le chargement des données et seront ralentis pendant le reste de l'exécution de l'application [33].

Outre les heuristiques, les *runtimes* peuvent avoir recours à d'autres mécanismes afin de prendre des décisions éclairées. Le principe de ces mécanismes est d'apporter au *runtime* une connaissance sur le programme et sur son comportement futur. En connaissant le comportement d'un programme à l'avance, le *runtime* peut par exemple anticiper l'utilisation de certaines ressources et en optimiser l'accès. À titre d'exemple, le pré-chargement de fichiers est une technique souvent mise en œuvre. S'il peut prédire qu'un programme s'apprête à utiliser un fichier, un *runtime* peut charger en avance le contenu du fichier dans la mémoire principale et, le moment venu, faire économiser au programme le temps d'attente correspondant à la latence du périphérique de stockage [64, 49, 19].

Parmi les autres mécanismes pouvant être utilisés par un *runtime* pour prédire le comportement à venir d'un programme figurent les oracles. Un oracle est une boîte noire logicielle dédiée à la prédiction des prochaines actions d'un programme. Les oracles reposent sur l'hypothèse que le comportement de nombreuses applications parallèles varie peu, voire pas du tout, d'une exécution à l'autre. En capturant le comportement d'un programme, un oracle peut prédire, lors des exécutions suivantes, son comportement futur [26, 41, 49]. Pour cela, l'oracle enregistre le déroulement d'une exécution d'un programme puis compare les exécutions suivantes à cet enregistrement. En explorant la suite de l'enregistrement, l'oracle est en mesure de prédire les prochaines actions du programme.

1.2 Problématique

Actuellement, pour prendre des décisions, les *runtimes* n'ont recours qu'à des heuristiques ou à des annotations du code par les développeurs. Ces deux méthodes ont l'avantage d'être peu coûteuses en temps de calcul, mais restent limitées.

Dans cette thèse, nous cherchons ainsi à répondre à la question suivante : **en complétant un *runtime* avec un oracle, comment prédire le comportement futur d'une application parallèle de manière précise pour prendre de meilleures décisions.**

1.3 Contributions

Au cours de cette thèse, nous avons construit PYTHIA, un oracle capable de prédire le comportement futur d'une application parallèle. Les prédictions de PYTHIA peuvent être utilisées par des *runtimes* pour prendre des décisions. Comme montré dans la figure 1.1, cet oracle est composé de deux parties. La première, PYTHIA-RECORD, permet de capturer le comportement d'un programme. Lors des exécutions suivantes de l'application, la seconde, PYTHIA-PREDICT, compare une exécution de référence à l'exécution courante et informe le *runtime* du comportement futur du programme.

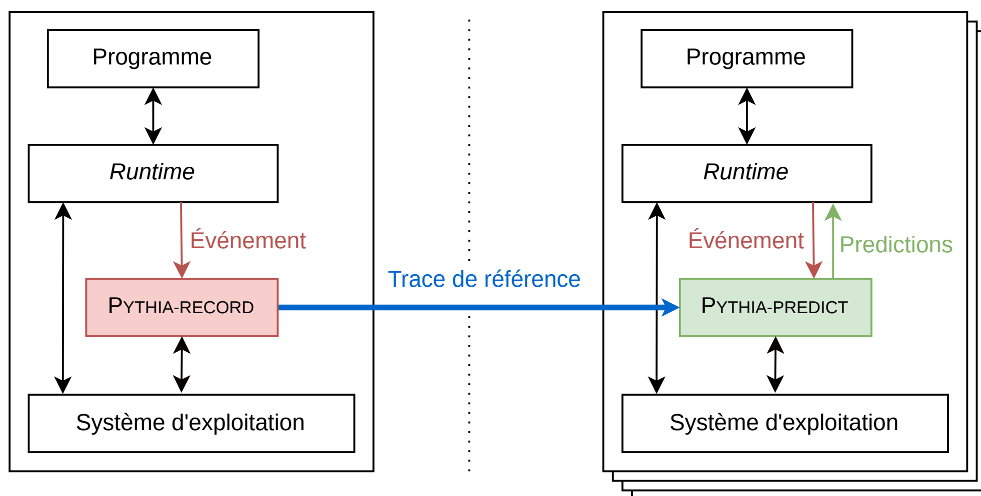


FIGURE 1.1 – Vision d'ensemble de PYTHIA et de ses deux parties PYTHIA-RECORD et PYTHIA-PREDICT.

1.3.1 Réduction de traces à la volée sous forme de grammaire

Notre première contribution est un algorithme performant permettant de capturer à la volée la structure d'une trace d'exécution sous la forme d'une grammaire.

Nous avons créé PYTHIA-RECORD, un outil permettant de capturer à la volée la structure de l'exécution d'un programme. PYTHIA-RECORD est capable, sans dégrader les performances d'un programme, d'enregistrer une trace d'exécution de celui-ci. En reconnaissant à la volée les différents schémas récurrents au sein d'une séquence d'événements soumis par le *runtime*, PYTHIA-RECORD construit une grammaire représentant la structure du programme.

L'évaluation de notre algorithme a montré qu'il permet de construire à la volée une grammaire représentant une trace de plusieurs millions d'événements avec un surcoût négligeable pour l'exécution du programme.

1.3.2 Utilisation d'une grammaire pour prédire le comportement futur d'un programme

Notre seconde contribution est un oracle générique. Cet oracle, appelé PYTHIA-PREDICT, compare la séquence d'événements décrivant l'exécution d'un programme à une grammaire représentant son exécution de référence, et

ce pour produire des prédictions à destination d'un *runtime*. Ce *runtime* peut alors utiliser les prédictions fournies pour guider ses décisions.

Pour cela, nous introduisons la notion de « point de progression » pour représenter l'état d'avancement d'un programme au sein d'une trace d'exécution réduite sous la forme d'une grammaire. Ces points de progression permettent de suivre l'exécution d'un programme et de retrouver des correspondances entre l'exécution courante et son exécution de référence. De plus, nous utilisons les points de progression pour explorer la grammaire sous la forme d'un arbre probabilisé des possibles comportements futurs.

Enfin, nous montrons que ces points de progression peuvent être utilisés par un *runtime* afin d'enrichir les prédictions fournies par PYTHIA. De cette façon, PYTHIA peut être utilisé comme oracle générique pour des prédictions spécifiques aux besoins de chaque *runtime*.

L'évaluation des prédictions montre que PYTHIA-PREDICT est capable de correctement prédire les comportements futurs de programmes parallèles variés, y compris lorsque ceux-ci sont exécutés avec des jeux de données de tailles différentes de ceux utilisés lors de l'exécution de référence. L'évaluation montre aussi que le coût de ces prédictions est suffisamment faible pour que les prédictions soient utilisées pour des optimisations au sein de *runtimes*.

1.3.3 Résultats

Ces deux contributions sont synthétisées au sein de l'implémentation de PYTHIA [1]. Nous évaluons l'utilisabilité de PYTHIA en implémentant une optimisation au sein du *runtime* GNU OpenMP.

Nos travaux ont mené à plusieurs publications :

— **Alexis Colin**, François Trahay, and Denis Conan. *Factorisation de traces d'exécutions de programmes pour l'analyse et la prédiction*. Présenté dans la Conférence francophone d'informatique en Parallélisme Architecture et Système (Compas), 2020. La tenue de la conférence a été annulée.

— Mohammed Islam Naas, François Trahay, **Alexis Colin**, Pierre Olivier, Stéphane Rubini, Frank Singhoff, and Jalil Boukhobza. *EZIOTracer : unifying kernel and user space I/O tracing for data-intensive applications*. Publié dans le *Workshop* international *Challenges and Opportunities of Efficient and Performant Storage Systems*, pages 1–11, 2021 [55]. Cet article a été republié dans la revue internationale ACM SIGOPS Operating Systems Review [38] ;

— **Alexis Colin**, François Trahay, and Denis Conan. *PYTHIA : un oracle pour guider les décisions des runtimes.* Présenté dans la Conférence francophone d'informatique en Parallélisme Architecture et Système (Compas), 2022.

— **Alexis Colin**, François Trahay, and Denis Conan. *PYTHIA : an oracle to guide runtime system decisions.* Présenté à la conférence *IEEE International Conference on Cluster Computing, 2022*

1.4 Organisation du document

La suite de ce document est organisée comme suit :

— dans le chapitre 2, nous présentons le contexte dans lequel s'inscrit le travail de cette thèse, et plus particulièrement, les architectures matérielles parallèles, la programmation parallèle, et les *runtimes* et la façon dont ils prennent des décisions ;

1. Disponible sous licence libre à cette adresse : https://gitlab.com/parallel-and-distributed-systems/easytraceanalyzer/eta-factorizer

- dans le chapitre 3, nous présentons PYTHIA-RECORD, qui est notre première contribution. Nous présentons les différentes techniques existantes permettant d'enregistrer le déroulement de l'exécution d'un programme et de représenter la structure de cette exécution. Nous détaillons ensuite la représentation sous forme de grammaire ainsi que l'algorithme que nous avons conçu et mis en œuvre ;
- dans le chapitre 4, nous présentons PYTHIA-PREDICT, qui est notre seconde contribution. En particulier, nous montrons comment PYTHIA-PREDICT procède pour suivre l'exécution d'un programme, comment il prédit le comportement futur du programme, et comment ces prédictions peuvent être enrichies par un *runtime* pour des optimisations spécifiques ;
- dans le chapitre 5, nous évaluons nos deux contributions. Plus précisément, nous évaluons les surcoûts de PYTHIA-RECORD et PYTHIA-PREDICT ainsi que la justesse des prédictions de PYTHIA-PREDICT. Nous faisons aussi la démonstration de l'utilisation de PYTHIA en implémentant une optimisation au sein d'un *runtime* OpenMP ;
- Enfin, dans le chapitre 6, nous concluons le manuscrit en résumant les contributions et en présentant différentes perspectives de recherche pour prolonger notre travail.

Chapitre 2

Supports d'exécution pour la programmation parallèle

Sommaire

2.1 Architectures matérielles parallèles **7**
2.1.1 Grappes de calcul 7
2.1.2 Processeurs multicœurs 8
2.1.3 Architectures à accès mémoire non uniforme (NUMA) 8
2.1.4 Architectures hétérogènes 9
2.1.5 Conclusion 10
2.2 Programmation parallèle **11**
2.2.1 Programmation en mémoire partagée 11
2.2.2 Programmation hétérogène 15
2.2.3 Programmation en mémoire répartie 15
2.3 Supports d'exécution et prises de décision **16**
2.3.1 Heuristiques 17
2.3.2 Décision à l'aide d'annotations dans le code 18
2.3.3 Décision à l'aide de modèles de performances 18
2.3.4 Utilisation d'oracles 19
2.4 Conclusion **19**

Le calcul parallèle est à la base de la simulation numérique et de l'intelligence artificielle. Cependant, les architectures parallèles sont complexes et variées. Afin d'atténuer cette complexité, les développeurs et développeuses ont recours à différents modèles de programmation. Ces modèles de programmation permettent d'abstraire une partie de la complexité des machines. Ils sont implémentés par des supports d'exécution, ou *runtimes*, qui doivent alors adapter les programmes à la machine qui les exécute.

Dans ce chapitre, nous étudions le contexte technique et scientifique dans lequel s'inscrit notre travail. Nous présentons dans un premier temps les principales architectures matérielles parallèles dans la section 2.1. Dans la section 2.2, nous introduisons les modèles de programmation implémentés par les principaux *runtimes* utilisés pour concevoir des programmes parallèles. Dans la section 2.3, nous détaillons les techniques à disposition de ces *runtimes* pour prendre des décisions. Enfin, nous concluons ce chapitre et expliquons l'approche adoptée dans nos travaux pour améliorer les décisions des *runtimes* dans la section 2.4.

2.1 Architectures matérielles parallèles

Un ordinateur parallèle est une machine ou un ensemble de machines capable d'exécuter plusieurs flots d'instructions en même temps. Différentes techniques, combinables entre elles, existent afin de produire de tels ordinateurs. Nous présentons ici les quatre principales architectures matérielles présentes sur les serveurs et les grappes de calcul (section 2.1.1), les processeurs multicœurs (section 2.1.2), les architectures à accès mémoire non uniforme (section 2.1.3), les architectures hétérogènes (section 2.1.4), puis nous concluons (section 2.1.5).

2.1.1 Grappes de calcul

Afin de disposer de davantage de puissance de calcul pour la résolution d'un problème, ce dernier peut être divisé en sous-problèmes dans le but de les faire résoudre par des ordinateurs différents. Cette approche permet de disposer théoriquement de la capacité de calcul cumulée de toutes les machines disponibles, aux coûts de synchronisation et des communications réseau près. Elle possède de plus l'avantage d'être modulaire : des programmes qui ne nécessitent pas la puissance de calcul de toutes les machines disponibles peuvent être exécutés sur des sous-ensembles de ces machines, ce qui permet d'économiser et de partager les ressources.

Un tel assemblage de machines capables de travailler de concert est généralement appelé *grappe de calcul* ou *cluster*. La figure 2.1 montre comment s'organise une grappe de calcul. Plusieurs ordinateurs autonomes formant autant de nœuds de calcul sont connectés en réseau. Ces machines peuvent charger les données dont elles ont besoin depuis des nœuds de stockage mutualisés, qui sont des équipements dédiés aux accès parallèles à un système de fichiers. Les différents nœuds de calcul travaillent ensuite conjointement à la résolution du problème en échangeant entre eux les données nécessaires.

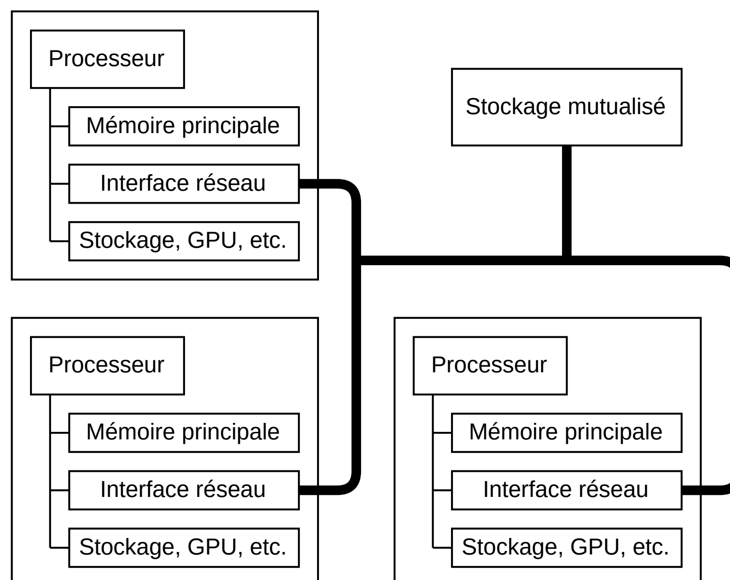


FIGURE 2.1 – Schéma de principe d'une grappe de calcul.

Il est à noter que les différentes machines d'une grappe de calcul ne partagent pas leur espace mémoire principal. Les données initiales, les résultats intermédiaires et les solutions agrégées doivent tous être dupliqués et transportés par le réseau. Ce dernier est par conséquent un élément crucial pour la performance d'une grappe de calcul. Aussi, les grappes de calcul utilisent généralement des réseaux dits « rapides » qui offrent une faible latence et une bande passante élevée (par exemple *infiniband* et *omnipath*). Malgré cela, le réseau reste sujet à la congestion, ce qui peut significativement dégrader les performances des programmes qui l'exploitent. Afin d'exécuter des

programmes parallèles sans avoir recours à une grappe de calcul, les ingénieurs ont développé des processeurs capables d'exécuter plusieurs fils d'exécution sur une même machine.

2.1.2 Processeurs multicœurs

Les processeurs multicœurs sont des processeurs capables d'exécuter plusieurs fils d'exécution en même temps. La figure 2.2 illustre comment un processeur multicœur est organisé. Un processeur multicœur est composé de plusieurs cœurs de calculs capables d'exécuter chacun un ou plusieurs (*hyperthreading*) fils d'exécution. Comme pour un processeur monocœur traditionnel, un processeur multicœur dispose d'une mémoire principale unique gérée par un unique contrôleur mémoire. Afin de réduire les temps d'accès à la mémoire, celle-ci est accédée par l'intermédiaire de différents caches mémoire, plus rapides, mais aussi plus coûteux, et donc plus petits.

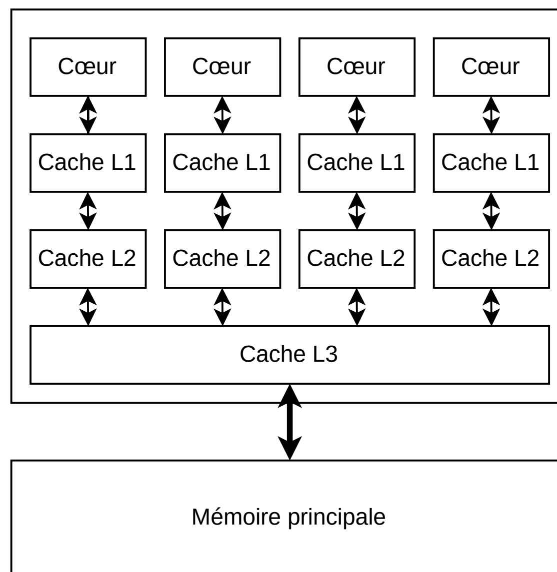


FIGURE 2.2 – Schéma de principe d'un processeur multicœur et de ses caches mémoire.

Contrairement aux grappes de calcul, les différents cœurs d'un processeur multicœur partagent leur espace d'adressage mémoire et ont accès à l'ensemble des ressources de la machine. Un *thread* peut accéder à la mémoire principale et aux périphériques indépendamment du cœur qui l'exécute. Il peut ainsi partager de la mémoire avec d'autres *threads*. Un *thread* peut aussi être déplacé de manière transparente par l'ordonnanceur du système d'exploitation d'un cœur à un autre. Un processeur multicœur n'a donc pas besoin de dupliquer la mémoire ou d'utiliser le réseau pour partager les données entre ses cœurs et les différents *threads* qu'il exécute. Malheureusement, la multiplication des cœurs au sein d'un processeur multicœur classique augmente la charge de travail du contrôleur mémoire, qui peut à son tour devenir une cause de dégradation des performances des programmes.

2.1.3 Architectures à accès mémoire non uniforme (NUMA)

Les processeurs multicœurs permettent d'exécuter des programmes parallèles sur une seule machine. Cependant, augmenter le nombre de cœurs au sein d'un processeur augmente d'autant le nombre de *threads* susceptibles d'accéder à la mémoire à un instant donné. Augmenter la capacité d'un processeur à exécuter du code parallèle augmente donc le risque de contention mémoire. Afin de limiter la contention mémoire, les architectures NUMA répartissent la mémoire principale au sein de plusieurs *nœuds NUMA* interconnectés. La figure 2.3 illustre le fonctionnement d'une telle architecture. Les différents cœurs sont répartis entre les différents nœuds NUMA. Chaque

nœud est associé à un contrôleur mémoire distinct. Les mémoires des différents nœuds NUMA sont reliées entre elles par un réseau d'interconnexion. Chaque cœur peut accéder de manière transparente à la mémoire locale de son nœud NUMA ou à la mémoire d'un autre nœud NUMA *via* le réseau d'interconnexion. Accéder à un bloc mémoire présent sur un autre nœud NUMA engendre cependant une latence supplémentaire. Cette différence de temps d'accès à la mémoire en fonction de son placement relativement au cœur qui y accède donne son nom aux architectures NUMA, qui est un acronyme pour « *Non Uniform Memory Access* » (pour « Accès Mémoire Non Uniformes »)

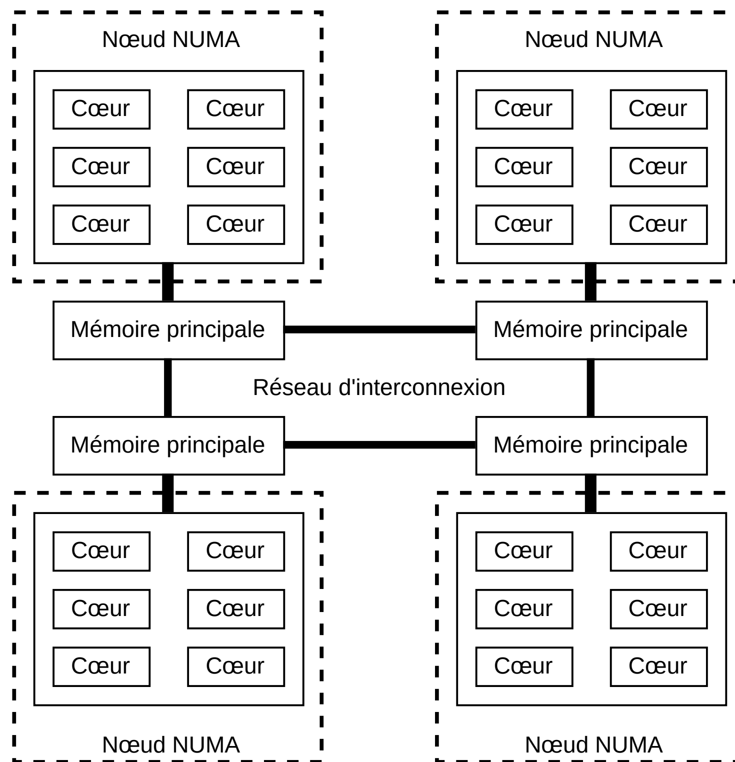


FIGURE 2.3 – Schéma de principe d'une architecture NUMA.

L'intérêt de cette architecture est de permettre d'augmenter le nombre de cœurs au sein d'un processeur en limitant la contention mémoire. En effet, la présence de plusieurs mémoires, chacune associée à un nœud NUMA différent *via* un contrôleur mémoire dédié, permet de répartir la charge des accès mémoire parallèles. De plus, le réseau d'interconnexion permet à tous les nœuds NUMA de partager le même espace d'adressage, ce qui permet aux architectures NUMA d'exécuter des programmes écrits pour un processeur multicœur habituel.

Les architectures NUMA sont cependant plus complexes que les architectures traditionnelles. Afin de limiter effectivement le risque de contention mémoire, encore faut-il que la mémoire allouée par les programmes soit effectivement réparties entre les différents nœuds du processeur. De plus, le temps d'accès à la mémoire dépend désormais du placement relatif de celle-ci et du *thread* qui y accède. Le placement relatif des *threads* et de la mémoire qu'ils utilisent doit donc être optimisé afin d'exploiter pleinement les capacités de calcul des processeurs NUMA.

2.1.4 Architectures hétérogènes

L'utilisation de coprocesseurs est une technique permettant d'augmenter les capacités de parallélisation d'une machine sans avoir recours à des processeurs plus complexes. Les coprocesseurs les plus courants sont sans

doute les processeurs graphiques, qui sont utilisés à la fois pour le calcul et pour le rendu d'images numériques. La figure 2.4 montre une architecture hétérogène composée d'un processeur multicœur classique et de trois processeurs graphiques. Un processeur graphique moderne peut être considéré comme un processeur doté d'un grand nombre de cœurs simplifiés, associés à une mémoire dédiée.

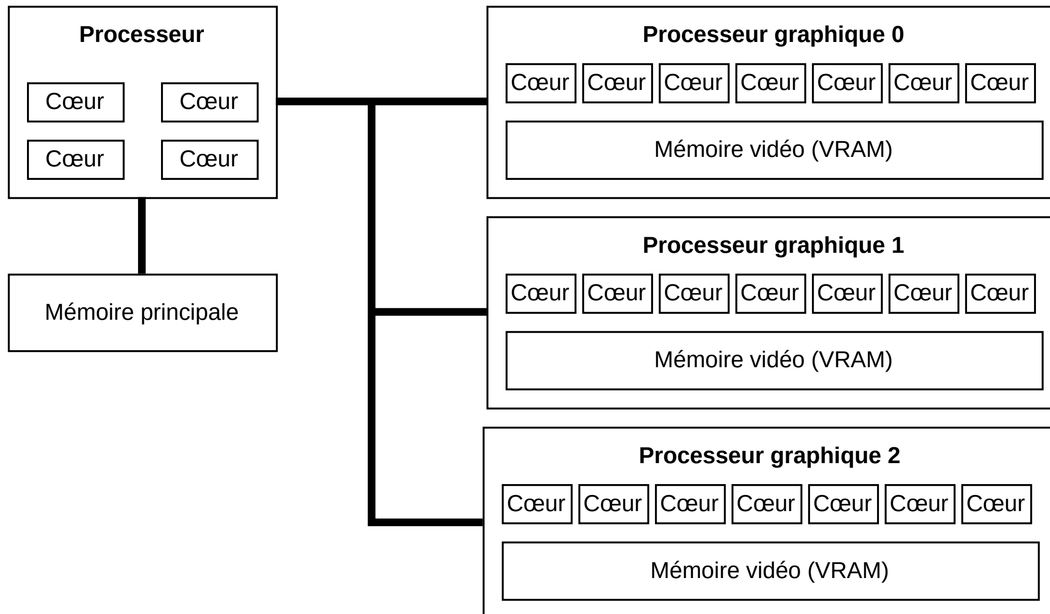


FIGURE 2.4 – Schéma de principe d'une architecture hétérogène.

Un avantage des architectures hétérogènes est la possibilité de démultiplier le nombre de cœurs présents sur une machine grâce à l'utilisation de composants « grand public », les processeurs graphiques étant par exemple couramment utilisés pour le jeu vidéo. Une carte graphique NVIDIA A100 dispose par exemple de 6912 coeurs de calcul. D'autres types de coprocesseurs ont été utilisés pour le calcul parallèle, tels que le processeur Cell/BE équipé de plusieurs processeurs spécifiques (Synergystic Processing Elements) ou le processeur Manycore Xeon Phi.

Cependant, les processeurs graphiques ne partagent pas les mêmes jeux d'instruction que les processeurs classiques. Deux programmes différents doivent donc être produits pour le processeur et pour les processeurs graphiques. Par ailleurs, si les processeurs graphiques permettent une parallélisation massive de certains calculs, ils réintroduisent des problèmes de duplication mémoire entre la mémoire principale de la machine et les mémoires des coprocesseurs. La bande passante entre les deux mémoires étant limitée, ces échanges peuvent entraîner des contentions et dégrader fortement les performances des programmes.

2.1.5 Conclusion

Dans cette section, nous avons décrit plusieurs techniques permettant de construire des architectures parallèles. En général, ces différentes techniques sont combinées afin de produire des machines hybrides. Par exemple, le supercalculateur Joliot-Curie comprend plusieurs partitions. La partition AMD est composée de 2292 machines NUMA à deux processeurs multicœurs de 64 cœurs chacun. La partition V100 est composée de 32 machines NUMA disposant chacune de deux processeurs multicœurs de 20 cœurs chacun et de quatre cartes graphiques NVIDIA Tesla V100.

Afin d'exploiter pleinement les capacités de ces machines, les programmes doivent être écrits et conçus de manière à exprimer pleinement les possibilités de parallélisation des calculs qu'ils effectuent. Pour cela, ils utilisent

différents modèles de programmation destinés à la conception de programmes parallèles. Ces modèles introduisent différentes abstractions qui permettent de masquer une partie de la complexité des machines et de rendre les programmes plus indépendants de ces dernières. Ainsi, les abstractions introduites par les différents modèles de programmation permettent aux *runtimes* qui les implémentent d'adapter chaque programme à la machine qui l'exécute afin de permettre aux programmes de s'exécuter efficacement sur chaque machine. Cette capacité des programmes à s'exécuter efficacement sur des machines différentes est appelée « portabilité des performances ».

2.2 Programmation parallèle

Dans la section 2.1, nous avons passé en revue les principales architectures matérielles utilisées dans le monde du calcul haute performance. Dans cette section, nous décrivons les principaux paradigmes de programmation permettant de les exploiter. Ces modèles de programmation ont pour objectif de permettre aux concepteurs et conceptrices d'exprimer les possibilités de parallélisation au sein des programmes. Ils permettent surtout, en introduisant différentes abstractions, de rendre les programmes plus indépendants des machines qui les exécutent. Cette indépendance des programmes vis-à-vis de l'architecture matérielle des machines est un prérequis à la portabilité des performances des programmes.

Dans la suite de cette section, nous présentons différents modèles de programmation utilisés dans le domaine du calcul haute performance. Nous présentons dans un premier temps les modèles dédiés aux machines disposant d'un espace mémoire partagé entre les différents nœuds de calcul (section 2.2.1), puis les modèles adaptés aux machines à mémoire répartie (section 2.2.3), et enfin, ceux dédiés à la programmation de machines hétérogènes (section 2.2.2).

2.2.1 Programmation en mémoire partagée

Dans le cas où un programme est exécuté par une machine dont toutes les unités de calcul partagent le même adressage mémoire, on parle de programmation *en mémoire partagée*. Cela correspond au cas d'une machine unique, multicœur ou NUMA, sur laquelle s'exécute un programme *multithreadé*. Dans ce cas, un programme est en mesure d'exploiter les capacités de parallélisation de la machine grâce aux processus légers, ou *threads*.

Au sein d'un processus, tous les *threads* partagent le même espace mémoire. Ce partage de l'espace mémoire a pour première conséquence que les données en mémoire ne doivent pas nécessairement être dupliquées pour être mises à disposition des différents nœuds de calcul. De plus, toute modification d'une valeur en mémoire est immédiatement visible par tous les *threads* en cours d'exécution. Un avantage de ce partage de la mémoire est de ne pas nécessiter de copie et transfert de données coûteux. Les communications et la synchronisation entre les différents fils d'exécution sont ainsi bien plus rapides que dans le cas d'une grappe de calcul ou d'une machine exploitant des processeurs graphiques par exemple.

Cependant, une conséquence du partage de l'espace mémoire est qu'un *thread* peut lire un bloc mémoire pendant que ce dernier est modifié par un autre *thread*. Dans un tel cas, même si les mécanismes de maintien de la cohérence des caches garantissent que les données visibles par les deux *threads* sont identiques, un *thread* peut lire la mémoire dans un état incohérent. Par exemple, si un *thread* met à jour les valeurs contenues dans un tableau, un autre *thread* peut entreprendre de lire ce tableau alors qu'une partie seulement des valeurs ont été modifiées. Le partage de l'espace d'adressage fait par conséquent apparaître la nécessité de synchroniser finement les différents accès concurrents à la mémoire entre eux.

Utilisation de primitives POSIX

Différentes techniques existent pour écrire le code qui doit être exécuté par chacun des *threads* et pour les synchroniser. Une possibilité est d'écrire des fonctions séparées pour chacun des *threads* à exécuter. C'est par exemple ce qui est permis par l'interface POSIX *pthread* [18] sur les systèmes Unix (ou Win32 sur les systèmes Windows). Le programmeur ou la programmeuse écrit dans des fonctions séparées les différents comportements possibles pour chaque *thread*.

La figure 2.5 montre un exemple de parallélisation de tâches sur deux *threads* en utilisant *pthread*. Le code consiste en une simple boucle remplissant un tableau de 1024 cases. Le code correspondant à la boucle est extrait dans la fonction `compute`. Le *thread* principal alloue la mémoire pour le tableau puis appelle `pthread_create` afin de créer un nouveau *thread*. L'appel spécifie que la fonction `compute` doit être exécutée sur le nouveau *thread*. Le *thread* principal exécute également cette fonction. Sous réserve que le système d'exploitation ordonnance les deux threads pour être exécutés simultanément sur deux cœurs séparés, le programme doit ainsi profiter des capacités parallèles de la machine. Une fois sa part de travail effectuée, le *thread* principal appelle `pthread_join` afin d'attendre que le *thread* secondaire ait effectué sa part également. Cet appel est donc bloquant. Le *thread* secondaire est détruit et l'exécution séquentielle du programme reprend.

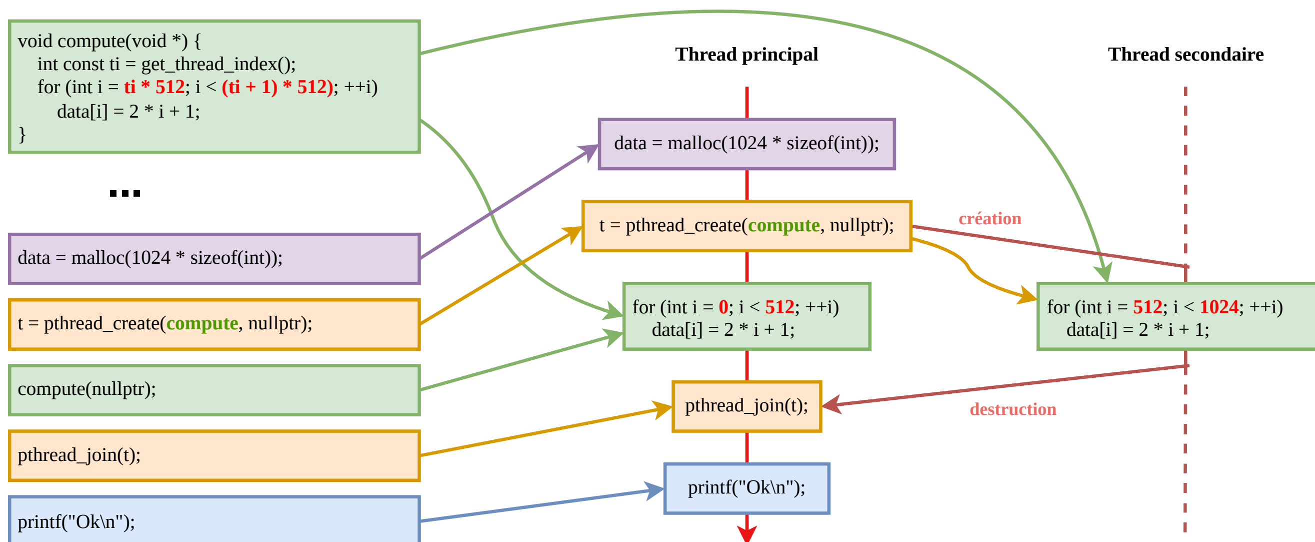


FIGURE 2.5 – Exemple d'utilisation de pthread pour accélérer une boucle dans un programme en C.

Dans cet exemple, les deux *threads* ne risquent pas d'accéder simultanément au même bloc mémoire grâce à l'utilisation de la fonction `get_thread_index`. Dans le cas contraire, il aurait été possible de définir dans le code une zone d'exclusion mutuelle afin de garantir que deux blocs de code qui entreraient en conflit d'accès mémoire ne peuvent pas s'exécuter simultanément. Une telle zone d'exclusion mutuelle est généralement obtenue par l'usage d'un verrou, sous la forme d'un sémaphore ou d'un *mutex*.

Avec *pthread*, le *runtime* prend en charge la gestion du *thread*. Lorsque la fonction `pthread_create` est appelée, le *runtime* crée un *thread* et le configure afin d'exécuter la fonction passée en argument. De même, il permet de synchroniser les *threads* lors de leur destruction avec la fonction `pthread_join` et offre différents mécanismes de synchronisation et de protection des données. L'utilisation de ces mécanismes, leur bon usage ainsi que le découpage du code et des données sur les différents *threads* restent cependant entièrement à la charge du programmeur ou de la programmeuse.

OpenMP

Une autre méthode de programmation en mémoire partagée est l'utilisation de OpenMP [20]. OpenMP est une interface de programmation multiplateforme disponible en Fortran, en C et en C++. OpenMP permet la conception de programmes parallèles en mémoire partagée à l'aide de directives insérées dans un code écrit dans un style séquentiel. Ces directives indiquent au compilateur quelles régions du code doivent être parallélisées, et de quelle façon. Par exemple, la figure 2.6 montre comment OpenMP peut être utilisé pour accélérer l'exécution d'une boucle `for` en C. La boucle est précédée d'une directive `#pragma omp parallel for` qui indique au compilateur que celle-ci doit être accélérée par parallélisation.

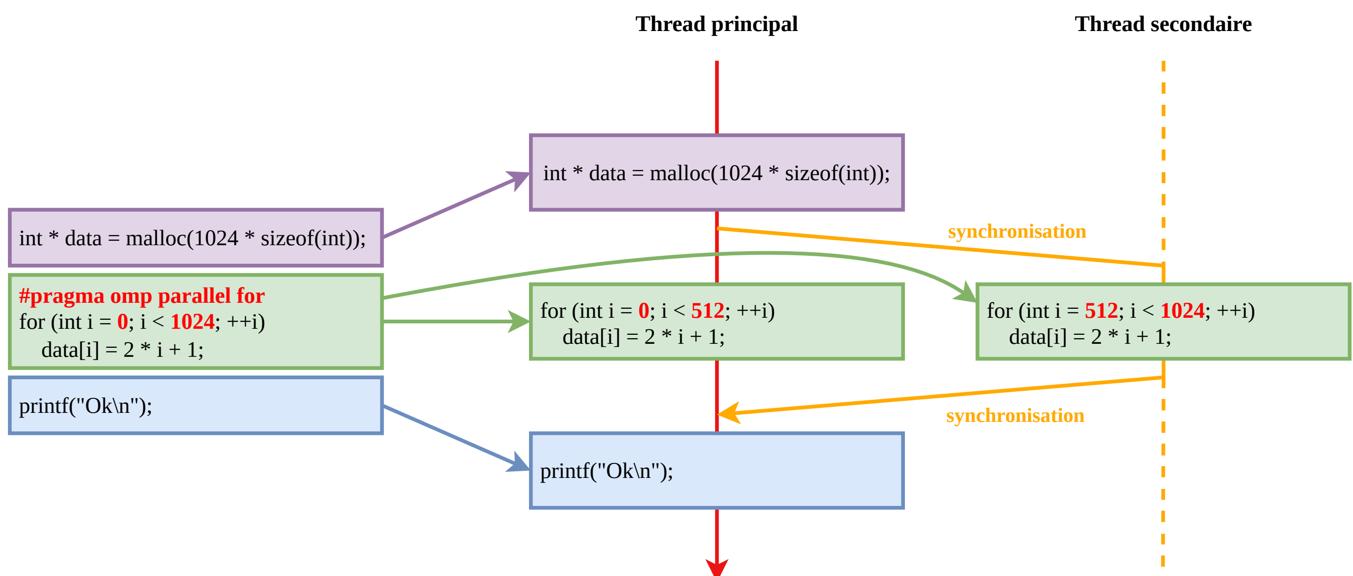


FIGURE 2.6 – Exemple d'exécution d'un code OpenMP sur deux *threads*.

OpenMP prend en charge la création des *threads*, leur destruction et leur synchronisation tout en permettant d'écrire du code parallèle dans un style proche du code séquentiel traditionnel. OpenMP permet également de spécifier si une variable doit être partagée par les différents *threads* et prend alors en charge la protection contre les accès concurrents.

Contrairement au *runtime pthread*, OpenMP ne s'arrête pas à la gestion des threads. Il prend en charge le découpage du code afin de le rendre parallélisable et répartit la charge de travail entre les différents *threads*. Il facilite aussi la protection des variables partagées entre les *threads* contre les accès concurrents. Enfin, il synchronise automatiquement les *threads* entre eux, et gère la communication entre les différents *threads* et la mise en commun des résultats, par exemple grâce à la clause `reduction`.

Programmation par tâches

La programmation par tâches consiste en un découpage des programmes en tâches également appelées *noyaux* ou *codelets*. Chaque tâche utilise des données en entrée et produit des données en sortie. Une tâche dépend d'une autre quand elle a besoin en entrée de données produites par celle-ci. L'ensemble des dépendances entre les tâches doit former un graphe orienté acyclique (ou *DAG* pour *Directed Acyclic Graph*). Chaque tâche peut alors être exécutée en parallèle des autres dès que les tâches dont elle dépend ont été exécutées.

La figure 2.7 montre un exemple de graphe de dépendance correspondant à la factorisation de Cholesky d'une matrice. Une factorisation de Cholesky consiste, pour une matrice symétrique définie positive A, à déterminer une

matrice triangulaire inférieure L telle que : $A = LL^T$. L'algorithme pour l'obtenir consiste en différentes étapes (triangulation, symétrisation, multiplication et factorisation) à effectuer successivement sur des parties de la matrice à factoriser. Un certain nombre de ces opérations dépendent les unes des autres tandis que d'autres peuvent être exécutées simultanément. En découpant le code de manière à séparer les différentes tâches et en explicitant leurs dépendances, on obtient un graphe de dépendance tel que celui représenté. Un ordonnanceur peut alors être utilisé afin de faire exécuter ces différentes tâches par des nœuds de calcul différents.

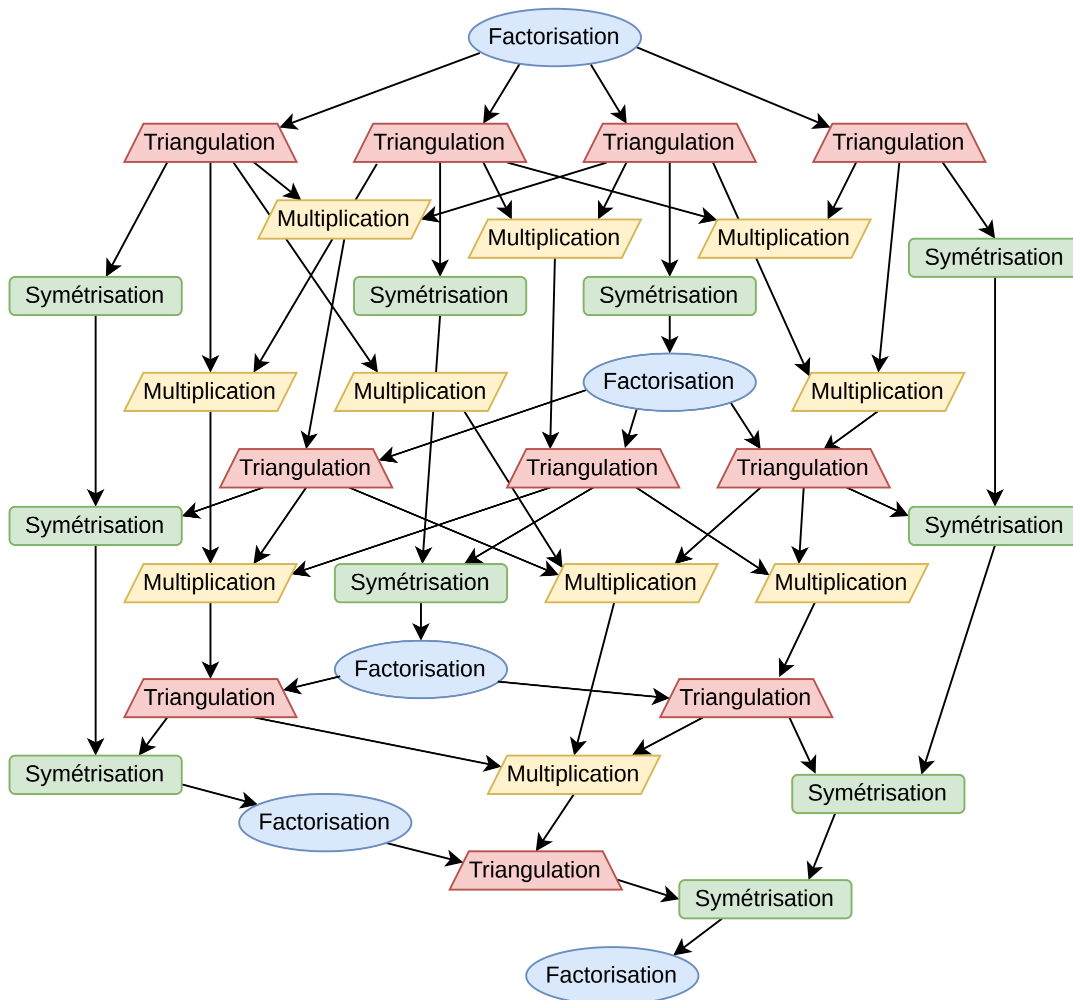


FIGURE 2.7 – Graphe de dépendance des tâches constituant une factorisation de Cholesky.

La programmation par tâches est intéressante à plus d'un titre. Contrairement aux modèles vus précédemment, la programmation par tâches exprime séparément les différentes tâches à exécuter au sein du programme. Contrairement à un code utilisant des régions parallèles, la programmation par tâches n'implique pas que les *threads* aient à se synchroniser au début et à la fin de chaque bloc parallélisable : quand un *thread* finit une tâche, il peut immédiatement en commencer une autre. Le système de dépendance permet également d'éviter que deux *threads* essaient d'accéder simultanément à la même donnée. Ces deux aspects permettent de réduire le nombre de synchronisation entre les *threads* et d'obtenir une meilleure utilisation des différentes unités de calcul.

Plusieurs *runtimes* implémentent la programmation par tâches en mémoire partagée, par exemple OpenMP (version $\geq$ 3.0) [7], CILK [11] et Intel TBB [44].

2.2.2 Programmation hétérogène

Afin d'améliorer les performances des programmes, des accélérateurs tels que des coprocesseurs ou des processeurs graphiques peuvent être utilisés. Ces accélérateurs ont la particularité de ne pas avoir la même architecture interne que les processeurs classiques. Par exemple, les processeurs graphiques sont des processeurs vectoriels disposant de milliers de cœurs de calcul. Ainsi, les accélérateurs utilisent généralement un langage machine différent de celui du processeur. Il faut donc écrire du code spécifique afin de pouvoir les utiliser. Par ailleurs, les données utilisées par les accélérateurs doivent être transférées entre la mémoire principale et l'accélérateur , et la bande passante est limitante.

Des environnements de développement dédiés existent pour programmer les accélérateurs. Par exemple, CUDA, SYCL [40] ou OpenCL [54] permettent d'écrire des programmes utilisant des coprocesseurs graphiques. OpenMP permet également d'exploiter une large gamme d'accélérateurs depuis sa version 4.0.

CUDA est un environnement de développement permettant d'exploiter des processeurs graphiques à partir de code écrit dans un style séquentiel classique. Le compilateur CUDA génère le code dédié au processeur et aux coprocesseurs graphiques, puis le *runtime* CUDA supervise les interactions ainsi que les transferts de données avec la mémoire des processeurs graphiques. Le *runtime* a aussi la responsabilité d'invoquer les noyaux de calcul et de synchroniser leur exécution avec le programme exécuté sur le processeur.

Il est également possible d'avoir recours à la programmation par tâches pour exploiter les accélérateurs. Par exemple, Taskflow [36], PaRSEC [13] et StarPU [4] permettent d'écrire des programmes pour architectures hétérogènes. Taskflow est un système permettant de répartir les tâches constituant un programme sur les différents nœuds de calcul, sur processeur ou coprocesseur graphique. StarPU permet de plus de proposer plusieurs implémentations pour chaque tâche. C'est alors StarPU qui choisira quelle implémentation utiliser en fonction de la topologie et des performances de l'architecture parallèle disponible. Il est ainsi possible d'écrire des tâches pour qu'elles soient exécutées sur CPU ou sur GPU, ou les deux, afin d'utiliser au maximum les ressources disponibles.

2.2.3 Programmation en mémoire répartie

Au sein des grappes de calcul, les différentes unités de calcul ne partagent pas leur espace d'adressage mémoire. Chaque unité de calcul exécute un processus distinct et l'utilisation de plusieurs *threads* ne suffit plus pour paralléliser un programme. Les données, les résultats intermédiaires et les résultats finaux sont échangés entre les nœuds de calcul en utilisant le réseau de la grappe. Les programmes s'exécutant sur des grappes de calcul doivent donc avoir recours à d'autres mécanismes que ceux utilisés en mémoire partagée. Dans cette section, nous présentons différents modèles de programmation utilisés pour écrire des programmes adaptés aux grappes de calcul.

Programmation répartie avec MPI

MPI (*Message Passing Interface*) [30] est une interface de programmation permettant d'écrire des programmes répartis sur une grappe de calcul. MPI prend en charge le fait de lancer un même exécutable sur différentes machines au sein d'une grappe. MPI fournit aussi les primitives nécessaires pour échanger les données entre les différents processus et les synchroniser. Ces primitives de communication sont de deux sortes :

- les primitives de communication point-à-point permettent aux développeurs d'envoyer des données d'un processus vers un autre processus ;
- les primitives de communication collectives permettent de répartir les données ou de collecter les résultats entre des groupes de processus.

La figure 2.8 montre une façon de calculer la moyenne des valeurs contenues dans un tableau de grande taille à l'aide de MPI. Tous les nœuds de calcul exécutent le même code. Pour cette raison, les sections de code qui ne doivent être exécutées qu'une seule fois sont conditionnées sur le rang du nœud de calcul. Les fonctions `MPI_Scatter` et `MPI_Gather` permettent respectivement de répartir les données entre les différentes unités de calcul de la grappe et de centraliser les résultats.

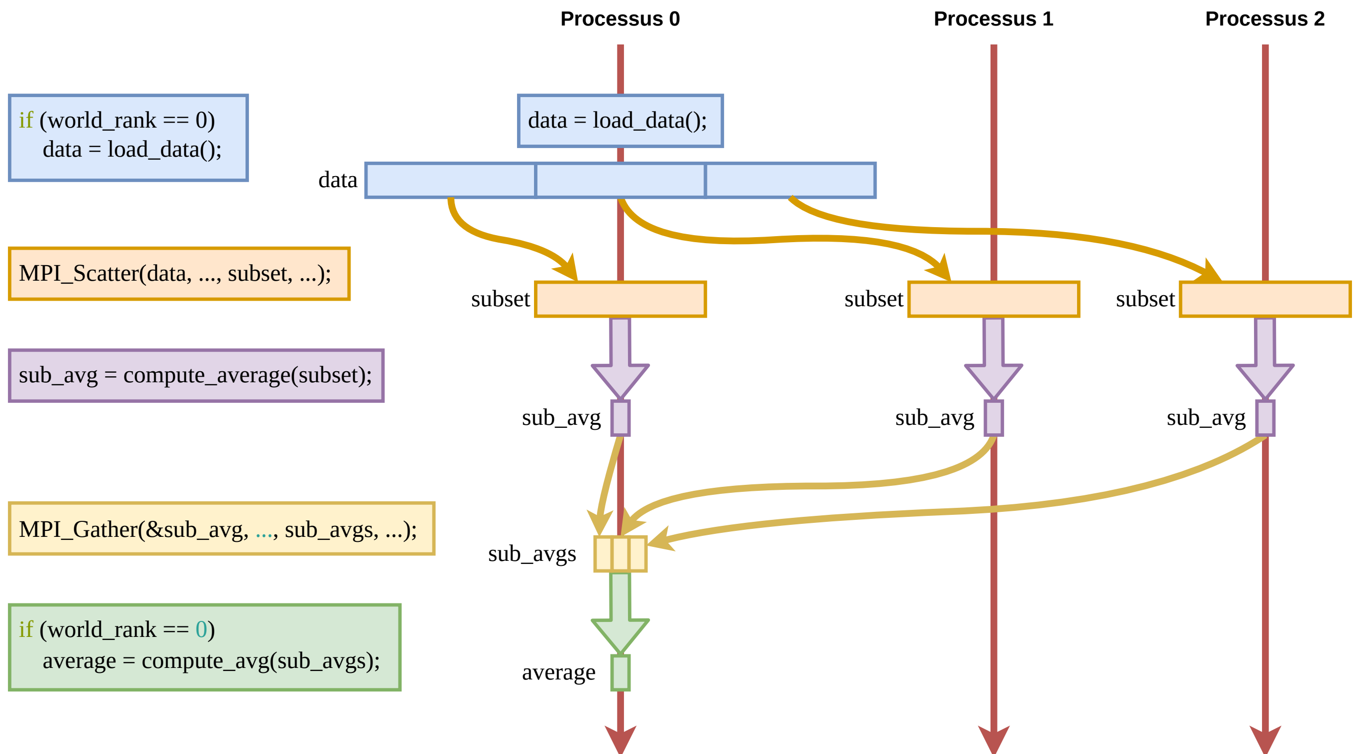


FIGURE 2.8 – Exemple d'exécution d'un code MPI avec trois processus.

Modèles de programmation hybride

Il est à noter que les grappes de calcul sont généralement composées de machines multicœurs ou NUMA, et dotées d'accélérateurs ou de GPU. Chaque machine correspond donc à plusieurs unités de calcul partageant le même espace mémoire. Il peut donc être intéressant de combiner plusieurs modèles de programmation en adoptant un paradigme hybride. Par exemple, il est possible de distribuer des calculs avec MPI sur une grappe de calcul et d'utiliser les accélérateurs de chaque machine avec CUDA. Combiner MPI et OpenMP permet de profiter à la fois de la capacité à distribuer de manière efficiente les calculs entre les différentes machines tout en profitant des avantages de la programmation en mémoire partagée sur chacune d'entre elle. Une partie du surcoût lié aux communications MPI peut ainsi être évité tout en limitant la duplication mémoire et en facilitant les synchronisations entre les différents *threads* sur chaque machine de la grappe de calcul.

2.3 Supports d'exécution et prises de décision

Nous l'avons vu, les différents modèles de programmation présentés dans la section 2.2 introduisent des abstractions afin de faciliter l'expression du parallélisme au sein des programmes et de rendre ces derniers plus in-

dépendants des machines qui les exécutent. Ces différents modèles de programmation sont implémentés par des *runtimes* qui, au sein de la pile logicielle, s'insèrent entre le système d'exploitation et les programmes. Grâce à l'utilisation des modèles de programmation, les programmes ne sont pas spécifiquement conçus pour s'exécuter sur une machine ou un type d'architecture parallèle particulier. Il résulte de ceci que les *runtimes* doivent prendre un certain nombre de décisions pour adapter chaque programme à la machine qui l'exécute.

Les décisions que prennent les *runtimes* permettent d'exploiter pleinement les ressources des machines. Si les décisions des *runtimes* sont mauvaises, l'exécution des programmes peuvent être ralenties. Pour prendre de bonnes décisions, les *runtimes* ont besoin d'informations sur la machine et sur le programme en cours d'exécution. Par exemple, en connaissant le comportement futur d'un programme, un *runtime* peut anticiper certaines opérations telles que l'ouverture ou le chargement d'un fichier. Autre exemple : si un *runtime* peut anticiper le comportement d'un programme exécuté sur une machine NUMA, il peut optimiser le placement des *threads* et de la mémoire afin de minimiser la latence d'accès.

Afin d'estimer le comportement futur d'un programme, les *runtimes* peuvent avoir recours à différentes méthodes que nous listons dans cette section. Dans la suite de la section, nous présentons l'utilisation d'heuristiques (section 2.3.1), d'annotation de code (section 2.3.2), de modèles de performances (section 2.3.3), et enfin d'oracles (section 2.3.4).

2.3.1 Heuristiques

Afin d'estimer le probable comportement futur d'un programme, un *runtime* peut s'appuyer sur une heuristique. Une heuristique est une stratégie permettant de trouver une solution approximée ou partielle à un problème donné. L'intérêt des heuristiques réside dans leur simplicité et leur coût réduit permettant d'éviter de devoir chercher des solutions exactes à des problèmes complexes.

En guise d'exemple, prenons l'heuristique *first touch* du noyau GNU/Linux pour le placement mémoire au sein des systèmes NUMA. Afin d'éviter de dégrader les performances du programme, il est préférable d'allouer la mémoire sur le nœud NUMA du cœur le plus susceptible d'y accéder. Pour cela, il faut, au moment de l'allocation de chaque page mémoire, que le système d'exploitation soit en mesure de déterminer quel *thread* va y accéder à l'avenir. Malheureusement, au moment de la demande d'allocation mémoire, il n'est pas possible de déterminer comment sera utilisé la mémoire dans le futur. L'heuristique *first touch* consiste à ne pas décider du placement mémoire au moment de l'allocation, mais à retarder la décision au moment de la première utilisation de la page mémoire. Linux considère qu'un *thread* qui accède maintenant à une page mémoire est davantage susceptible d'y accéder par la suite. En plaçant cette page sur le noeud NUMA qui exécute ce *thread*, Linux espère limiter les accès mémoire distants. Malheureusement, les heuristiques peuvent se tromper. Pour l'heuristique *first touch* par exemple, si le *thread* qui charge les données initiales du problème n'est pas celui qui utilise ces données par la suite, le placement mémoire est sous-optimal [31].

Autre exemple d'heuristique : au début de chaque région parallèle OpenMP, il est possible de choisir le nombre de *threads* à utiliser pour l'exécuter. En l'absence d'indication de la part des développeurs, le *runtime* doit faire un choix. L'implémentation GNU du *runtime* OpenMP dans sa version 12 fait le choix d'utiliser le maximum de *threads* disponibles. Pour les petites régions parallèles, ce choix peut, si le nombre de *thread* disponibles est assez grand, diminuer les performances du programme. En effet, synchroniser un grand nombre de *thread* peut devenir coûteux en comparaison des calculs à effectuer. Cette heuristique consiste à privilégier les grosses régions parallèles, en partant de l'hypothèse qu'elles représentent la majeure partie du temps d'exécution des programmes.

2.3.2 Décision à l'aide d'annotations dans le code

Lorsque le *runtime* ne peut pas deviner seul le comportement futur d'un programme, une solution possible pour le développeur est d'annoter le code pour y ajouter de l'information. En informant à la main le *runtime* du comportement futur du programme, le *runtime* est capable d'anticiper et d'optimiser son exécution.

Par exemple, le système d'annotation `posix_fadvise` permet au programmeur ou à la programmeuse de renseigner le système d'exploitation, qui remplit ici le rôle de *runtime*, à propos des prochaines utilisations de fichiers. La lecture d'un fichier stocké sur un disque dur est généralement une opération lente comparée à un accès à la mémoire centrale. Au moment où le programme demande à accéder à une partie d'un fichier, il est mis en pause le temps que les données soient chargées et rendues disponibles. Dans le cas d'un disque mécanique, cela correspond possiblement à plusieurs millisecondes d'attente, le temps de déplacer physiquement la tête de lecture le long du disque magnétique. `posix_fadvise` est une fonction permettant aux développeurs d'informer le *runtime* de futures opérations de lecture ou d'écriture sur des fichiers. Savoir à l'avance qu'un programme va accéder à une donnée peut permettre d'anticiper la lecture et donc de réduire significativement les temps d'attente au sein du programme. De même, savoir qu'un programme ne va plus utiliser un fichier permet de décharger les données pour économiser la mémoire.

Les annotations de code peuvent cependant être difficiles à maintenir au fur et à mesure de l'évolution d'un programme. Dans le cas de `posix_fadvise` par exemple, pour produire un effet, une annotation doit apparaître suffisamment en amont du code qui effectue l'opération décrite. Une personne qui modifie un programme utilisant `posix_fadvise` doit être attentive aux annotations qui peuvent ne pas se trouver dans la même portion de code que l'opération qu'elles cherchent à améliorer. Les annotations peuvent ainsi, sans qu'il existe de moyen efficace pour les déceler, devenir contre-productives, en conservant des informations obsolètes sur le comportement d'anciennes versions d'un programme, et ainsi mener le compilateur ou le *runtime* à des décisions sous-optimales.

2.3.3 Décision à l'aide de modèles de performances

Afin de prendre des décisions, le *runtime* peut construire un modèle du programme en cours d'exécution. En observant le comportement du programme, le *runtime* modélise les performances du programme dans l'objectif de prédire son comportement futur.

Prenons l'exemple de StarPU. Si la développeuse ou le développeur fournit le code d'une tâche en plusieurs versions, p.ex. adaptée aux processeurs graphiques ou adaptée aux machines disposant de nombreux cœurs de processeurs, ou encore adaptée à certains types de jeux de données, StarPU choisit pour chaque exécution de la tâche la version à effectivement utiliser. Pour ce faire, StarPU utilise le modèle de performance présenté dans [3] : il caractérise les tâches, les données, et les architectures matérielles, et essaye les différentes versions tour à tour pour construire le modèle de performance du programme.

Un autre exemple est NewMadeleine, qui offre différentes optimisations pour réduire la contention réseau dans les grappes de calcul. NewMadeleine peut agréger plusieurs messages ayant même destination [6] ou découper un messsage pour l'acheminer *via* plusieurs réseaux. Pour savoir quand appliquer une optimisation, il est nécessaire de prendre en compte le programme, le jeu de données utilisées, le matériel utilisé, la topologie du réseau, mais aussi les schémas de communication de l'application. Dans l'objectif d'ajuster les paramètres de la bibliothèque, les auteurs de [16] ajoutent à NewMadeleine la possibilité d'expérimenter différentes optimisations et de mesurer leur impact sur les performances des applications. À partir des résultats mesurés pour chaque paramétrage, NewMadeleine est capable de déterminer par la suite un ensemble de paramètres souhaitables pour chaque programme en fonction du jeu de données et de la machine.

Des modèles de performance sont également utilisés afin de configurer dynamiquement les différentes couches logicielles des entrées et sorties. Par exemple, les auteurs de [9] construisent des modèles de performance afin de

trouver le paramétrage des systèmes d'entrées/sorties parallèles permettant d'obtenir de meilleures performances pour chaque application.

2.3.4 Utilisation d'oracles

Les oracles sont une autre façon de prendre des décisions. Un oracle est un bloc logiciel pouvant être consulté afin de prédire le comportement à venir du programme. Les oracles s'appuient sur l'hypothèse selon laquelle le comportement de nombreux programmes varie peu d'une exécution à une autre. En enregistrant le comportement d'une application lors d'une de ses exécutions, un oracle peut ainsi prévoir le comportement à venir de ce programme lors de ses exécutions ultérieures.

À titre d'exemple, Omnisc'IO [26] est un oracle capable de prévoir les opérations à venir sur les fichiers au sein d'un programme. Pendant la première exécution du programme, Omnisc'IO enregistre toutes les opérations de lecture et d'écriture. En s'aidant des piles d'appels des fonctions correspondantes, Omnisc'IO compare les exécutions suivantes à celle qu'il a enregistrée et prédit les opérations à venir. Ceci est rendu possible par le fait qu'en pratique les paramètres d'appel d'une fonction travaillant sur un fichier sont stables pour une pile d'appel donnée.

Autre exemple : la philosophie Unix est de concevoir des programmes faisant peu de choses, mais le faisant bien (« *doing only one thing and doing it well* ») [59]. Ainsi, un programme complexe qui respecte cette philosophie tend à utiliser lui-même d'autres programmes plus petits. Hui Lei et Dan Duchamp proposent un oracle qui enregistre quels fichiers sont ouverts et par quel sous-programme [49]. Pendant l'exécution d'un programme, cet oracle construit l'arbre des sous-programmes exécutés et des fichiers qu'ils utilisent. En comparant cet arbre à celui qu'il a enregistré pendant les précédentes exécutions, l'oracle est en mesure de prédire quels seront les prochains fichiers utilisés. Cette information est utilisée afin de charger à l'avance les fichiers en mémoire. Les fichiers étant déjà chargés en mémoire au moment où un programme en a besoin, il n'est pas nécessaire d'attendre leur chargement depuis le disque et les performances du programme en sont améliorées.

2.4 Conclusion

Dans ce chapitre, nous avons passé en revue le paysage du calcul haute performance et de la programmation parallèle. Nous avons décrit les principales architectures matérielles modernes en usage dans le monde du HPC. Nous avons ensuite présenté les différents modèles de programmation utilisés pour écrire des programmes parallèles et les différentes méthodes de prises de décision utilisées par les supports d'exécution. Parmi ces méthodes de prise de décisions, les oracles se démarquent. En comparant entre elles deux exécutions d'un programme, les oracles disposent de davantage d'informations pour prédire le comportement futur, sans avoir recours à une modification du code source. Contrairement aux modèles de performance, les oracles permettent de prédire les prochaines actions d'un programme et pas seulement leur performance.

Malgré cela, nous constatons qu'actuellement les prises de décisions au sein des *runtimes* reposent majoritairement sur des heuristiques. Cela s'explique par leur simplicité de mise en œuvre. À l'inverse, les oracles sont plus difficiles à mettre en œuvre. Ceux qui existent sont spécifiques et ne sont pas réutilisables pour de nouveaux usages. À notre connaissance, il n'existe à ce jour aucun oracle générique et réutilisable disponible pour une utilisation au sein d'un *runtime*.

Dans le but d'ouvrir la voie à de nouvelles expérimentations et de nouvelles optimisations au sein des *runtimes*, nos travaux visent à concevoir un oracle générique et simple d'utilisation. Une première étape dans cette démarche est de doter notre oracle d'une représentation de la structure de l'exécution d'un programme qui soit générique et

indépendante de son utilisation. La seconde étape est d’utiliser cette structure pour prédire le comportement futur du programme.

Dans le chapitre suivant, nous étudions les différentes techniques existantes d’instrumentation et de représentation structurée du déroulement de l’exécution d’un programme avant de décrire en détail comment notre oracle capture à la volée la structure d’un programme.

Chapitre 3

PYTHIA-RECORD : réduction de trace d'exécution à la volée pour la prévision

Sommaire

3.1 Enregistrer le déroulement de l'exécution d'un programme 22

3.1.1 Échantillonnage 23

3.1.2 Appels de fonction depuis le code source ou à la compilation 23

3.1.3 Utilisation de sondes 23

3.1.4 Injection de code machine 24

3.1.5 Pré-chargement de bibliothèque dynamique avec `LD_PRELOAD` 24

3.2 Représentation structurée d'une trace d'exécution 26

3.2.1 Stockage séquentiel des événements 26

3.2.2 Découverte de redondances *a posteriori* 26

3.2.3 Réduction à la volée sous forme de grammaire 27

3.2.4 Discussion 27

3.3 Représentation du comportement d'un programme sous forme d'une grammaire 28

3.3.1 Représentation d'une trace par une grammaire 28

3.3.2 Contraintes sur la grammaire 30

3.4 Réduction à la volée d'une trace d'exécution 32

3.4.1 Principe de l'algorithme de réduction 32

3.4.2 Notations 33

3.4.3 Algorithme de réduction 34

3.4.4 Implémentation 36

3.5 Conclusion 37

Puisque les *runtimes* ont besoin de connaître le comportement futur des programmes, nous proposons de concevoir un oracle fournissant ce service. Comme présenté dans la figure 3.1, cet oracle, nommé PYTHIA, est composé de deux parties : PYTHIA-RECORD capture le comportement d'un programme tandis que PYTHIA-PREDICT prédit le comportement futur de l'exécution d'un programme.

La construction de cet oracle repose sur l'hypothèse que le comportement de nombreux programmes parallèles semble peu varier, voire pas du tout, d'une exécution à l'autre. En enregistrant le déroulement de l'exécution d'un

programme, il serait donc possible d'en déduire sa structure et de prédire son comportement lors des exécutions futures.

Dans ce chapitre, nous présentons PYTHIA-RECORD (à gauche dans la figure 3.1), un outil permettant de capturer le comportement d'un programme parallèle et de le stocker sous la forme d'une trace d'exécution. PYTHIA-RECORD doit permettre à un *runtime* d'enregistrer des événements décrivant le comportement d'un programme et d'en extraire la structure. En particulier, PYTHIA-RECORD doit être capable de repérer les boucles du programme et les schémas récurrents au sein de la séquence d'événements. Le surcoût lié à son utilisation au sein d'un *runtime* doit rester faible pour ne pas ralentir l'exécution du programme et ainsi risque de changer le comportement du programme.

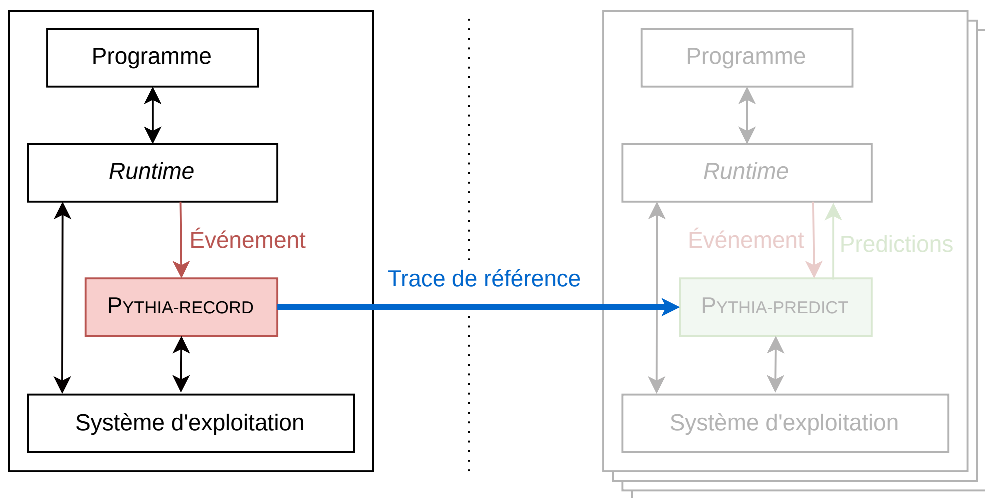


FIGURE 3.1 – Intégration de PYTHIA-RECORD dans l'outil PYTHIA.

Dans les sections 3.1 et 3.2, nous introduisons respectivement les principales techniques pour enregistrer le déroulement de l'exécution d'un programme et pour représenter de manière structurée la trace résultante. Ensuite, dans les sections 3.3 et 3.4, nous détaillons plus particulièrement comment une trace d'exécution peut être représentée sous la forme d'une grammaire et comment cette grammaire est construite à partir du flux d'événements qui constitue la trace. Enfin, la section 3.5 conclut le chapitre.

3.1 Enregistrer le déroulement de l'exécution d'un programme

Enregistrer le déroulement de l'exécution d'un programme est un procédé fréquemment utilisé. Un tel enregistrement peut permettre de comprendre le comportement et l'utilisation d'un programme [68]. Dans le domaine du calcul haute performance, une trace d'exécution permet notamment de trouver et d'étudier les causes de problèmes de performance [14] ou des bogues [66]. Nous présentons dans cette section les cinq principales techniques pouvant être mises en œuvre pour capturer le comportement d'un programme : l'échantillonnage de l'exécution d'un programme (section 3.1.1), l'ajout d'appels au sein du code source ou à la compilation (section 3.1.2), l'utilisation de sondes (section 3.1.3), l'injection d'instruction dans un exécutable (section 3.1.4) et l'interception d'appels de fonction par préchargement de bibliothèque dynamique (section 3.1.5).

3.1.1 Échantillonnage

L'échantillonnage (en anglais *sampling*) est une technique utilisée pour observer le comportement d'un programme sans avoir à modifier son code source. Cette technique est en particulier utilisée pour estimer le temps passé dans les différentes parties du code. Par exemple, `perf`, qui est un utilitaire présent sur Linux, suspend régulièrement l'exécution du programme pour sauvegarder différentes informations telles que la pile d'appels du programme. Ces piles d'appels donnent une estimation des fonctions les plus souvent appelées, de leur durées d'exécution ainsi que du contexte de leur appel. De nombreux outils d'analyse de performances permettent l'échantillonage de l'exécution d'un programme [63, 69, 60, 29, 21]. Cependant, comme l'échantillonnage est probabiliste, plus le temps effectivement passé dans une fonction est court, plus la probabilité de ne jamais suspendre le programme pendant son exécution est grande. Un appel à une fonction peut ainsi ne jamais être détecté. De plus, l'échantillonage permet de collecter les piles d'appel au moment de chaque échantillon mais ne donne aucune information sur l'état du programme entre deux échantillons. Il est par conséquent impossible de reconstruire le chemin de code emprunté par le programme par échantillonage.

3.1.2 Appels de fonction depuis le code source ou à la compilation

Le code d'un programme peut être modifié afin d'y introduire des instructions enregistrant le déroulement de son exécution. Un événement peut par exemple être enregistré à chaque fois que le code entre ou sort d'une fonction ou à chaque itération d'une boucle. Cette méthode possède l'avantage de permettre de choisir précisément quels événements enregistrer, et quand et quelles informations y adjoindre. Cependant, elle requiert l'accès au code source de l'application afin de recompiler le programme.

Certains outils d'analyse de performance mettent à disposition des utilisateurs des macros dédiées permettant de simplifier l'enregistrement du début et de la fin des fonctions [69, 63, 32, 71, 22]. Modifier « à la main » toutes les fonctions d'un programme peut toutefois être fastidieux. Aussi, des outils permettent d'effectuer cette opération à la compilation du programme [63, 32, 34, 10]. Par exemple, XRay [10], qui est un outil intégré à la suite d'outils LLVM [48], permet d'ajouter automatiquement du code au début et à la fin de chaque fonction afin d'enregistrer une trace contenant tous les appels. XRay permet aussi l'analyse des traces produites pour détecter différentes faiblesses comme la contention sur des verrous.

3.1.3 Utilisation de sondes

Certains logiciels tels que les systèmes d'exploitation fournissent un moyen alternatif à l'édition de leur code source afin de permettre l'altération et l'instrumentation de certaines de leurs fonctions. Les sondes logicielles sont placées par les concepteurs et conceptrices des programmes pour donner un accès à des informations internes au programme. Ce procédé est utilisé pour faciliter le développement d'outils compatibles.

La bibliothèque standard GNU pour le C est un exemple de bibliothèque fournissant des sondes logicielles. Ses sondes permettent d'instrumenter les appels à l'allocateur mémoire et éventuellement d'en modifier le comportement. Les utilisateurs et utilisatrices de cette bibliothèque peuvent fournir des fonctions qui seront appelées chaque fois que l'allocateur est utilisé. Sur le même principe, le mécanisme PBF (Berkeley Packet Filter) [52] permet de placer des sondes afin d'intercepter et de réagir à divers événements survenant au sein du noyau Linux. Par exemple, EZIOTracer [55] utilise IOTracer pour tracer les utilisations des entrées–sorties d'un programme, lequel s'appuie sur BPF pour instrumenter les entrées–sorties au sein du noyau Linux. Certains *runtimes* proposent également une interface pour l'instrumentation de programmes à l'aide de sondes. Par exemple, CUDA peut être instrumenté à l'aide de l'interface CUPTI et OpenMP à l'aide de l'interface OMPT [27].

Cette technique a l'avantage de ne pas nécessiter de modification du code source à instrumenter. En particulier, dans le cas du noyau Linux, BPF permet d'instrumenter et de traiter les informations relevées dans l'espace utilisateur sans développer un module noyau spécifique. Une limite importante à l'usage de sondes logicielles est que leur placement doit avoir été anticipé et doit permettre une couverture « complète » des besoins des utilisateurs et utilisatrices pour être utile.

3.1.4 Injection de code machine

Une autre alternative à l'ajout de code d'instrumentation dans le code source d'un programme avant ou pendant la compilation est de modifier le programme *a posteriori* par injection de code machine dans l'exécutable. Cette méthode consiste à injecter du code machine directement dans l'exécutable du programme. Cette technique est utilisée par de nombreux outils, parmi lesquels DynInst [17], DynamoRIO [15], EZTrace [5], MAQAO [25] ou PIN [51].

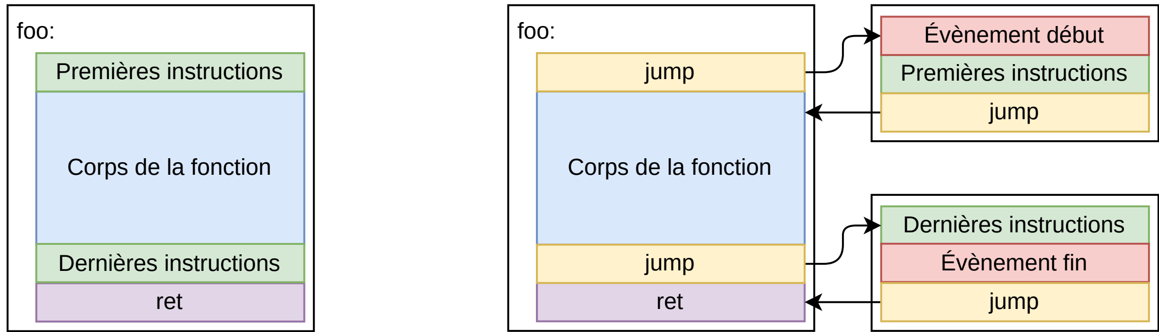
Si le principe général reste le même, l'injection de code peut être réalisée de différente manière. À titre d'exemple, la figure 3.2 illustre deux techniques mises en œuvres respectivement par DynIst et Eztrace. Pour simplifier, nous pouvons considérer une fonction dans un programme compilé comme un bloc de code machine dont le début est identifié par une adresse et la fin marquée par l'instruction `ret`, comme illustré par la figure 3.2a. La méthode mise en œuvre par DynIst est illustrée dans la figure 3.2b. Elle consiste à déplacer les premières instructions de la fonction dans un nouveau bloc, en y adjoignant des instructions supplémentaires, par exemple pour générer un événement quand la fonction est appelée. Des sauts sont insérés de sorte que lors d'un appel de la fonction le processeur commence par exécuter le nouveau bloc de code puis revienne exécuter la suite de la fonction. La même opération peut être effectuée à d'autres endroits de la fonction, par exemple juste avant l'instruction `ret`. La méthode utilisée par EZTrace, légèrement différente, est représentée par la figure 3.2c. Au lieu de faire de simple sauts entre le code injecté et le code initial, EZTrace a recours à une instruction `call`, comme si la fonction était appelée une seconde fois depuis le code injecté. De cette façon, au lieu de revenir au code qui l'avait appelé initialement, l'instruction `ret` de la fonction instrumentée faire revenir le processeur dans le bloc injecté, qui peut exécuter de nouvelles instructions avant d'utiliser `ret` à son tour.

En conséquence, tous les appels à une fonction sont tracés comme si le code d'instrumentation faisait partie du code initial, et ceci sans disposer du code source. Cependant, pour ce faire, nous devons connaître la liste des fonctions composant le programme ainsi que leurs positions, et tous les programmes ne fournissent pas ces informations. Par ailleurs, selon le jeu d'instructions du processeur, et donc en relâchant l'exigence de portabilité, l'injection de code peut être utilisée pour instrumenter des blocs de code plus fins que les fonctions tels que des boucles, comme le permettent PIN et DynInst.

3.1.5 Pré-chargement de bibliothèque dynamique avec `LD_PRELOAD`

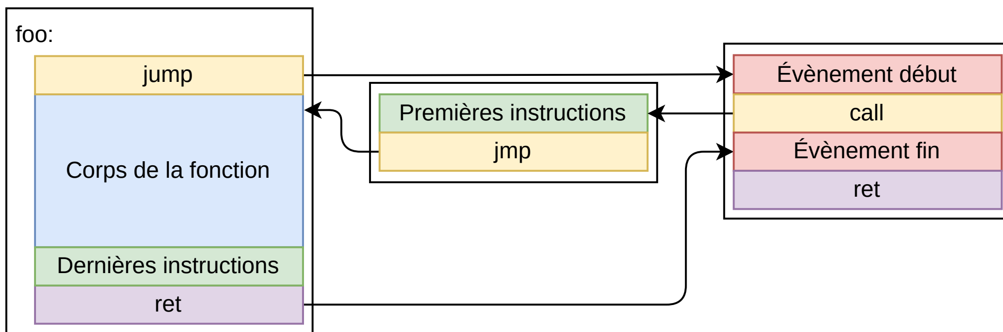
Les bibliothèques dynamiques sont un moyen couramment utilisé pour déployer du code utilisable par plusieurs applications sur un système. Lors de la phase d'édition des liens d'un programme, les fonctions des bibliothèques utilisées par celui-ci sont listées et identifiées par leur nom. Plus tard, lors de l'exécution du programme, les bibliothèques dynamiques sont chargées par le système et leurs fonctions sont mises à sa disposition comme illustré dans la figure 3.3.

Sur Linux, l'utilisateur du système peut spécifier manuellement des bibliothèques dynamiques devant être chargées et utilisées par un programme au moment de son lancement, et ce à l'aide de la variable d'environnement `LD_PRELOAD`. La figure 3.4 montre comment ce mécanisme peut être utilisé pour l'instrumentation. Les fonctions d'une bibliothèque dynamique étant identifiées par leur nom, si deux bibliothèques proposent une fonction différente mais ayant toutes les deux le même nom, le système d'exploitation privilégie la fonction de la bibliothèque



(a) Exemple d'une fonction non modifiée au sein d'un exécutable.

(b) Injection de code à la façon de DynIst.



(c) Injection de code à la façon d'EZTrace.

FIGURE 3.2 – Exemples de techniques d'instrumentation par injection de code.

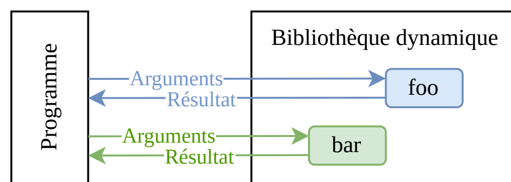


FIGURE 3.3 – Utilisation d'une bibliothèque dynamique.

venant en premier dans la variable d'environnement `LD_PRELOAD`. Ainsi, il est possible d'intercepter l'appel à une fonction en pré-chargeant une autre bibliothèque proposant une fonction de même nom. Cette nouvelle fonction peut elle-même appeler la fonction de la bibliothèque originale tout en sauvegardant différentes informations à propos de l'appel au sein d'une trace d'exécution, par exemple la date de début et de fin de l'appel, ses arguments et sa valeur de retour.

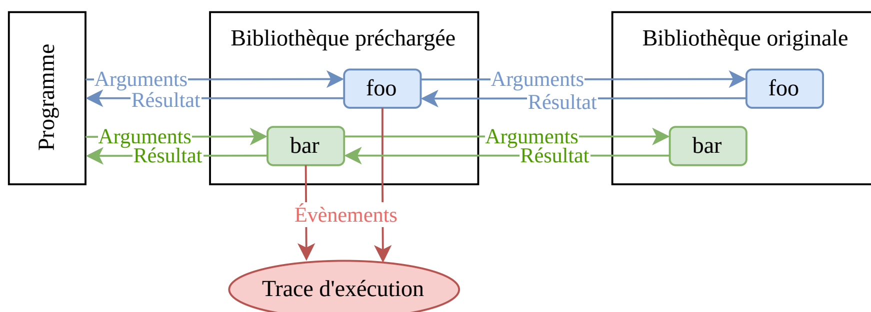


FIGURE 3.4 – Interception d'appels à une bibliothèque dynamique à l'aide de `LD_PRELOAD`.

Cette technique est très couramment utilisée par la plupart des outils d'analyse de performance [71, 53, 63, 32]. Elle a l'avantage de ne demander de modification ni du code source ni de l'exécutable du programme à instrumenter. Bien sûr, elle n'est utilisable que pour les bibliothèques dynamiques des programmes.

3.2 Représentation structurée d'une trace d'exécution

Une trace d'exécution est un enregistrement d'événements survenus au cours de l'exécution d'un programme. Ces événements reflètent les actions effectuées par un programme et l'état de celui-ci à un instant donné. Par exemple, un événement peut représenter un appel de fonction en spécifiant le contexte d'appel, les arguments et le résultat retourné. Un événement peut aussi représenter un appel système ou bien renseigner sur la valeur à un instant donné d'une variable ou d'un compteur matériel. Enfin, un événement peut représenter un changement d'état de la machine ou de son environnement, ou simplement une interaction entre les différents fils d'exécution du programme.

Un programme parallèle est constitué d'une multitude de fils d'exécutions. Ainsi, une trace est constituée de différents flux d'événements générés par chaque *thread* du programme. En général, le flux d'événements de chaque *thread* est représentatif de la structure du code exécuté. Les fonctions génèrent des schémas récurrents et les boucles engendrent des répétitions dans le flux d'événements.

Nous passons ici en revue les différentes manières existantes de représenter une trace d'exécution dans le but de déterminer la représentation de trace à utiliser au sein de PYTHIA. Nous présentons ainsi le stockage séquentiel des événements (section 3.2.1), la découverte de redondances *a posteriori* (section 3.2.2), et la réduction de trace à la volée sous forme de grammaire (section 3.2.3). Enfin, nous discutons de ces différentes techniques et de la possibilité de les utiliser pour notre oracle (section 3.2.4).

3.2.1 Stockage séquentiel des événements

La façon classique de représenter une trace d'exécution est de stocker séquentiellement les événements collectés au cours de l'exécution du programme dans un ou plusieurs fichiers [37, 63, 23, 42, 28, 22]. Ces différentes représentations mettent en œuvre différentes techniques afin de réduire le coût de l'instrumentation en temps de calcul ou en espace de stockage. Par exemple, certaines comme OTF [42] ou OTF2 [28] utilisent Zlib [24] pour compresser les traces d'exécution.

La plupart de ces formats permettent d'enregistrer les entrées et les sorties de fonction qui peuvent permettre de reconstituer des piles d'appel. Tous permettent de savoir combien de *threads* et de processus ont été utilisés au cours de l'exécution du programme. Cependant, ces différents formats ne permettent pas d'inférer ou de représenter la structure d'une exécution.

3.2.2 Découverte de redondances *a posteriori*

La découverte *a posteriori* de schémas récurrents et de répétitions au sein d'une trace d'exécution est une technique permettant de découvrir la structure de celle-ci depuis une trace séquentielle. Par exemple, Trahay *et al.* [70] proposent un algorithme réduisant une trace dans le but de la présenter sous une forme compacte et structurée. En éliminant les répétitions au sein de la trace d'exécution, cette méthode permet de réduire considérablement la quantité d'informations nécessaires pour décrire une trace d'exécution et rend celle-ci plus facilement lisible par un être humain.

L'algorithme proposé par Trahay *et al.* a l'avantage de ne pas nécessiter que la trace soit porteuse *a priori* d'informations sur la structure du programme. Elle est donc applicable à toute trace et tout fichier de journalisation.

Elle ne peut cependant pas être utilisée pendant l'exécution du programme. En effet, cette technique nécessite que la trace soit intégralement disponible sous sa forme séquentielle afin d'être parcourue pour y trouver les séquences redondantes. La découverte de redondances *a posteriori* ne peut ainsi pas être utilisée pour réduire la taille de la trace pendant l'exécution du programme et nécessite de pouvoir conserver la totalité de la trace séquentielle en mémoire pour être utilisée.

3.2.3 Réduction à la volée sous forme de grammaire

En 1997, Nevill-Manning et Witten introduisent Sequitur [56], un algorithme de compression par élimination de schémas récurrents au sein d'une séquence en utilisant une table de substitution. Sequitur permet de représenter une séquence d'événements sous la forme d'une grammaire. Appliqué à une trace d'exécution, Sequitur produit un résultat comparable à celui produit par la découverte de redondances *a posteriori*, à ceci près que Sequitur permet de construire ce résultat à la volée. Cependant, Sequitur ne reconnaît pas les boucles des programmes. En effet, Larus [47] remarque que Sequitur échoue à reconnaître certaines séquences récurrentes au sein des traces contenant des boucles et modifie l'algorithme pour corriger ce problème. Si ces modifications améliorent la détection de séquences récurrentes, elles ne sont toutefois pas suffisantes pour reconnaître les boucles d'un programme.

D'autres travaux modifient Sequitur afin de le doter de la capacité à reconnaître les boucles au sein d'une trace d'exécution. Par exemple, Cyclitur [1] est un algorithme basé sur Sequitur capable de reconnaître les boucles des programmes. Omnisc'IO [26] utilise quant à lui StarSequitur, une autre version modifiée de Sequitur capable de reconnaître les boucles à la volée. Malheureusement, Cyclitur repose sur une approche par fenêtre glissante qui limite la taille du corps de boucles pouvant être reconnues et dégrade les performances de l'algorithme. Omnisc'IO de son côté travaille sur des événements représentant directement les piles d'appel du programme au moment de l'émission des événements. En pratique, cette information n'est pas toujours disponible.

3.2.4 Discussion

Un pré-requis général à la représentation structurée d'une trace d'exécution est qu'au sein de la trace l'ordre des différents événements soit lui-même structuré. Du fait de la manière dont les programmes sont écrits, les traces d'exécution ont tendance à refléter la structure du code source du programme instrumenté. Cependant, cette structure peut être occultée de différentes façons. D'une part, il est nécessaire de séparer au sein de traces différentes les flux d'événements provenant des différents *threads* d'une application. Si cela n'est pas fait, les différents flux d'événements se trouvent entrelacés au gré de l'ordonnancement des *threads* par le système d'exploitation. D'autre part, certains modèles de programmation peuvent également destructurer l'ordre d'exécution des différentes parties d'une programme. Par exemple, l'utilisation de co-routines ou de fibres, des processus légers collaboratifs, permet la suspension et l'entrelacement des fonctions du programme. De même, la programmation par tâches [4, 13, 36] découpe le programme de telle sorte que le *runtime* dispose d'une certaine liberté quant à l'ordre d'exécution des tâches. Par conséquent, la trace d'exécution ne capture pas le graphe de dépendance des tâches mais l'ordre dans lequel l'ordonnanceur a décidé de les exécuter. À l'inverse, les autres modèles de programmation décrits dans la section 2.2 ne présentent généralement pas cet inconvénient. PYTHIA s'adresse donc *a priori* aux programmes qui n'utilisent pas le modèle de programmation par tâches.

Par ailleurs, nous remarquons que dans la pratique la structure du code du programme n'est pas toujours représentative de la structure des événements utiles pour un *runtime*. Par exemple, un *runtime* peut vouloir instrumenter l'usage de fichiers au sein d'un programme de simulation physique. La structure du code qui charge les fichiers peut ne pas avoir d'incidence sur l'ordre dans lequel sont chargés les fichiers. Cet ordre pourrait par exemple être dicté par la façon dont ceux-ci sont référencés au sein de fichiers de données. Dans la mesure où PYTHIA a voca-

tion à être un oracle générique, nous ne pouvons pas imposer une sémantique aux événements de la trace. Par exemple, nous ne pouvons pas nous appuyer sur des piles d'appel. Par conséquent, nous ne nous sommes pas intéressés aux travaux où ce type d'information est primordial, tels que les graphes d'appels complets (*Complete Call Graphs* [62, 43]).

Enfin, les programmes parallèles modernes peuvent être exécutés sur de très nombreux nœuds de calcul et contenir un très grand nombre d'événements. Une réduction à la volée de la trace d'exécution, en ce qu'elle permet de minimiser l'utilisation mémoire de PYTHIA-RECORD est une exigence.

La nécessité d'utiliser des événements abstraits et l'intérêt de réduire la trace à la volée ont orienté nos recherches dans la direction de la représentation des traces d'exécution sous la forme de grammaires.

3.3 Représentation du comportement d'un programme sous forme d'une grammaire

Au cours de l'exécution d'un programme, un *runtime* peut utiliser PYTHIA pour enregistrer le déroulement de celle-ci. Pour ce faire, le *runtime* informe PYTHIA de la survenue d'événements qui sont collectés par PYTHIA-RECORD. Dans le but de capturer la structure de l'exécution du programme, PYTHIA-RECORD construit une grammaire représentant la trace d'exécution constituée des événements survenus. Dans cette section, nous précisons comment la grammaire est utilisée pour représenter la structure de l'exécution du programme. En particulier, nous détaillons comment une grammaire peut représenter une trace d'exécution (section 3.3.1) et quelles contraintes appliquer à la grammaire pour représenter la structure de la trace (section 3.3.2).

3.3.1 Représentation d'une trace par une grammaire

Pour le dire simplement, une grammaire est une structure de données récursive permettant de représenter des séquences ordonnées de symboles. L'ensemble des symboles de la grammaire est constitué de deux ensembles de symboles disjoints : l'ensemble des symboles terminaux et l'ensemble des symboles non terminaux. Dans le cas d'une grammaire représentant une trace d'exécution, les *symboles terminaux* représentent les événements de la trace tandis que les *symboles non terminaux* représentent les séquences redondantes au sein de celle-ci.

Représentation des événements

Pour enregistrer le comportement d'un programme, le *runtime* notifie PYTHIA de la survenue d'événements tels que l'entrée ou la sortie d'une fonction ou d'une région parallèle. Chaque événement est associé par le *runtime* à un symbole terminal de la grammaire. Cette association entre événements et symboles permet au *runtime* de définir des classes d'équivalence entre les événements. Par exemple, un *runtime* souhaitant enregistrer la quantité de mémoire allouée peut faire le choix d'utiliser un unique symbole terminal pour représenter un appel à `malloc` ou un appel à `calloc`, rendant par là même ces deux événements indiscernables du point de vue de PYTHIA. Pour PYTHIA, une trace d'exécution est donc une séquence de symboles terminaux abstraits. Par convention, nous représentons les symboles terminaux par des lettres minuscules.

Représentation des séquences redondantes

Chaque symbole non terminal est associé par une règle grammaticale à une séquence de symboles terminaux et non terminaux. Ces règles de grammaire permettent de décrire les séquences d'événements qui sont alors représentées par le symbole non terminal associé. En particulier, la *règle grammaticale racine* associe la séquence

d'événements correspondant à la trace d'exécution complète au *symbole racine*, noté R. Par convention, nous représentons toujours les symboles non terminaux par des lettres majuscules. La figure 3.5 montre comment une règle grammaticale permet de représenter une séquence redondante au sein d'une grammaire. Au sein de la trace « $abcab$ », la séquence « ab » se répète deux fois. Dans la figure, la séquence « ab » est représentée par la règle grammaticale $A \mapsto ab$. Les deux occurrences de la séquence « ab » peuvent ainsi être remplacées par le symbole non terminal A au sein de la règle grammaticale racine R, qui représente alors la séquence « $abcab$ » par la règle grammaticale $R \mapsto AcA$.

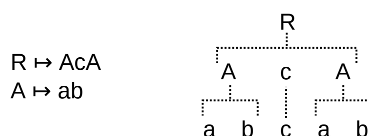


FIGURE 3.5 – Exemple de représentation de la séquence « $abcab$ » par une grammaire.

Représentation des boucles

La plupart des programmes mettent en œuvre des boucles qui permettent de répéter un certain nombre de fois une même section de code. Au sein d'une trace d'exécution, ces boucles peuvent apparaître sous la forme d'un grand nombre de répétitions consécutives d'un même événement ou d'une même séquence d'événements. Afin de représenter ces boucles dans la grammaire, le nombre de répétitions est noté en exposant du symbole correspondant. Par exemple, la séquence « $abbbbbbbc$ » peut être écrite « ab^7c ». Il est aussi possible de représenter la répétition d'une séquence à l'aide d'un symbole non terminal. Par exemple, la figure 3.6 montre comment la séquence « $abbcbcab$ » est représentée. Les séquences « ab » et « bc » apparaissent deux fois chacune et sont respectivement associées aux symboles non terminaux A et B. La règle racine peut ainsi s'écrire $R \mapsto ABBA$. Le symbole B se répétant deux fois consécutivement, il est remplacé par B^2 et la règle racine s'écrit alors $R \mapsto AB^2A$.

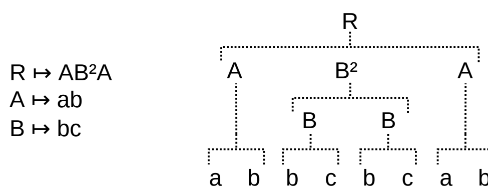


FIGURE 3.6 – Exemple de représentation de la « $abbcbcab$ » par une grammaire.

Il est à noter que nos grammaires sont déterministes et n'ont aucun moyen de représenter un embranchement conditionnel. À titre d'exemple, considérons un programme qui contient une boucle effectuant neuf itérations générant chacune un événement a si l'indice de boucle est un multiple de 3 et un événement b sinon. Exécuter ce programme génère la séquence « $abbabbabb$ » qui sera représentée par la grammaire suivante :

$$R \mapsto A^3$$
$$A \mapsto ab^2$$

Il est important de remarquer que les événements sont regroupés par trois au sein de la règle associée au symbole A. Par conséquent la règle racine n'affiche que trois répétitions du symbole A.

Discussion

Nous avons présenté ici les différents mécanismes pouvant être utilisés au sein d'une grammaire pour représenter une séquence d'événements. Cependant, la façon d'utiliser ces mécanismes pour représenter une séquence d'événements n'est pas unique. Par exemple, les trois grammaires ci-dessous représentent toutes trois la séquence « *abababab* ».

$$R \mapsto AbB \quad A \mapsto aB \quad B \mapsto ba$$

$$R \mapsto AAa \quad A \mapsto BB \quad B \mapsto aa$$

$$R \mapsto A^4a \quad A \mapsto ab$$

L'une de ces grammaires est plus concise et semble mieux représenter la façon dont la séquence peut être humainement interprétée. Il est par conséquent utile de préciser de quelle manière les différents mécanismes mis à disposition par les grammaires doivent être utilisés pour représenter la structure d'une trace d'exécution.

3.3.2 Contraintes sur la grammaire

Dans cette section, nous présentons les différentes contraintes que nous appliquons aux grammaires dans l'objectif de représenter la structure d'une trace d'exécution. Ces contraintes ne sont pas originales et ont déjà été présentées explicitement par certains travaux [1, 56, 26] et apparaissent implicitement dans d'autres [70].

Unicité des digrammes au sein de la grammaire

La contrainte d'unicité des digrammes au sein de la grammaire vise à ce que toutes les séquences redondantes d'une trace d'exécution soient reconnues par une règle grammaticale unique. Si une séquence d'au moins deux symboles apparaît plus d'une fois dans la grammaire, celle-ci doit être remplacée par un nouveau symbole non terminal la représentant. Il n'est pas nécessaire que les deux occurrences de la séquence apparaissent au sein de la même règle de grammaire. Par exemple, dans la grammaire ci-dessous, la séquence « *ab* » apparaît deux fois, dans les règles associées aux symboles R et A.

$$R \mapsto Aab$$
$$A \mapsto abc$$

Afin de respecter l'unicité du digramme « *ab* », il est nécessaire d'introduire une nouvelle règle grammaticale $B \mapsto ab$. La grammaire devient alors :

$$R \mapsto AB$$
$$A \mapsto Bc$$
$$B \mapsto ab$$

Cette règle doit être appliquée en tenant compte du nombre de répétitions de chaque symbole. Par exemple, dans la grammaire ci-dessous, le digramme « *ab* » apparaît deux fois dans la règle racine de la grammaire, respectivement sous la forme « a^3b » et « a^2b^2 ».

$$R \mapsto a^3bca^2b^2$$

Une nouvelle règle grammaticale $A \mapsto a^2b$ doit donc être introduite et la grammaire devient alors :

$$R \mapsto aAcAb$$
$$A \mapsto a^2b$$

Grâce à cette règle, nous garantissons que toutes les séquences redondantes sont représentées par une règle

grammaticale. Cependant, elle ne garantit nullement que toutes les règles grammaticales représentent une séquence redondante au sein de la trace.

Pertinence des règles grammaticales

La contrainte de pertinence des règles grammaticales garantit que toutes les règles grammaticales représentent une séquence redondante au sein de la trace. Par ce faire, elle stipule que tout symbole non terminal de la grammaire est utilisé au moins deux fois au sein de celle-ci. Par exemple, dans la grammaire ci-dessous, le symbole non terminal A n'apparaît qu'une seule fois et n'apporte donc aucune information sur la structure de la trace.

$$R \mapsto Ac$$
$$A \mapsto ab$$

La règle grammaticale associée au symbole A doit alors être supprimée et la grammaire devient :

$$R \mapsto abc$$

Le nombre d'apparitions d'un symbole non terminal au sein de la grammaire prend en compte le nombre de répétitions consécutives noté en exposant. Par exemple, dans la grammaire suivante, le symbole A est utilisé deux fois au travers de son exposant et la contrainte de pertinence des règles est donc respectée.

$$R \mapsto A^2c$$
$$A \mapsto ab$$

Utilité des règles grammaticales

La règle d'utilité des règles grammaticales indique que toute règle grammaticale doit associer un symbole non terminal à une séquence d'au moins deux symboles, en tenant compte des nombres de répétitions. En effet, un symbole terminal qui ne représenterait qu'un seul symbole n'agirait que comme un *alias* pour ce dernier. Cela complexifierait la grammaire sans apporter d'information sur sa structure et rendrait plus difficile la reconnaissance de redondances.

Élimination des répétitions

Enfin, la règle d'élimination des répétitions permet de reconnaître les boucles et répétitions au sein de la trace. Aucun symbole n'apparaît deux fois côte à côte dans la grammaire : toutes les répétitions de la forme « a^na^m » sont fusionnées en « a^{n+m} ». En appliquant cette contraintes à la grammaire concernant les symboles non terminaux, on permet à celle-ci de représenter des boucles constituées de plusieurs événements. Par exemple, dans la grammaire ci-dessous, la séquence « A^5A » doit être simplifiée.

$$R \mapsto aA^5A$$
$$A \mapsto bc$$

Cette grammaire devient alors :

$$R \mapsto aA^6$$
$$A \mapsto bc$$

3.4 Réduction à la volée d'une trace d'exécution

Dans cette section, nous présentons l'algorithme qui permet à PYTHIA de représenter dans une grammaire une trace d'exécution à partir des événements envoyés par le *runtime*. La grammaire est construite pendant l'exécution du programme et respecte les différentes contraintes présentées dans la section 3.3.2. Nous exposons dans un premier temps le principe de fonctionnement au travers d'un exemple (section 3.4.1), puis, après avoir introduit diverses notations (section 3.4.2), nous décrivons l'algorithme (section 3.4.3).

3.4.1 Principe de l'algorithme de réduction

Afin d'illustrer le principe de fonctionnement de l'algorithme de réduction de trace, nous présentons l'exemple de deux ajouts successifs du même symbole c à la fin de la trace représentée par la grammaire suivante :

$$\begin{aligned} R &\mapsto ...Bb^5 \\ A &\mapsto b^3c^2 \\ B &\mapsto b^2A \end{aligned}$$

Insertion d'un premier symbole c à la fin de la trace

En ajoutant le symbole terminal c à la fin de R comme ci-dessous (cf. la notation « $\leftarrow c$ »), nous remarquons que le digramme « bc » est déjà présent dans la séquence associée au symbole A. Ajouter le symbole c à la fin de cette séquence ferait apparaître le digramme « bc », déjà présent au sein de la grammaire dans la séquence associée au symbole A.

$$\begin{aligned} R &\mapsto ...Bb^5 \quad \leftarrow c \\ A &\mapsto b^3c^2 \\ B &\mapsto b^2A \end{aligned}$$

Afin de respecter la contrainte d'unicité des digrammes, un nouveau symbole non terminal C est créé pour représenter la séquence b^3c. La séquence « b^3 » est retirée de la fin de la séquence racine et doit être désormais remplacée par le symbole C.

$$\begin{aligned} R &\mapsto ...Bb^2 \leftarrow C \\ A &\mapsto Cc \\ B &\mapsto b^2A \\ C &\mapsto b^3c \end{aligned}$$

La séquence « bC » n'est pas présente dans la grammaire ; le symbole C est donc simplement ajouté à la fin de la séquence associée au symbole R.

$$\begin{aligned} R &\mapsto ...Bb^2C \\ A &\mapsto Cc \\ B &\mapsto b^2A \\ C &\mapsto b^3c \end{aligned}$$

Insertion d'un deuxième symbole c à la fin de la trace

Continuons notre exemple en ajoutant un deuxième symbole terminal c à la fin de la trace.

$$\begin{aligned} R &\mapsto ...Bb^2C \leftarrow c \\ A &\mapsto Cc \\ B &\mapsto b^2A \\ C &\mapsto b^3c \end{aligned}$$

À la suite de l'ajout de c à R, observons que R se termine désormais par la séquence « Cc », déjà représentée par le symbole A. Par conséquent, retirons le symbole C de la fin de la séquence associée à R et essayons d'ajouter le symbole A à la place. À ce stade, remarquons cependant que le symbole C n'est désormais plus utilisé qu'une seule fois au sein de la grammaire.

$$\begin{aligned} R &\mapsto ...Bb^2 \leftarrow A \\ A &\mapsto Cc \\ B &\mapsto b^2A \\ C &\mapsto b^3c \end{aligned}$$

Donc, pour respecter la contrainte de pertinence des règles grammaticales, le symbole C est supprimé de la grammaire avec la règle grammaticale associée pour obtenir la grammaire qui suit.

$$\begin{aligned} R &\mapsto ...Bb^2 \leftarrow A \\ A &\mapsto b^3c^2 \\ B &\mapsto b^2A \end{aligned}$$

Enfin, nous souhaitons ajouter le symbole A. La séquence « b^2B » étant déjà représentée par le symbole B, retirons la séquence « b^2 » à la fin de la séquence associée à R et essayons d'ajouter le symbole B à la place. Nous pouvons alors incrémenter le nombre de répétitions de B et nous obtenons la grammaire suivante :

$$\begin{aligned} R &\mapsto ...B^2 \\ A &\mapsto b^3c^2 \\ B &\mapsto b^2A \end{aligned}$$

3.4.2 Notations

Afin de faciliter la lecture de la suite de ce chapitre, et en particulier de l'algorithme de construction de grammaire, nous introduisons différentes notations.

Grammaire

Étant donné une grammaire G, on note $\mathcal{T}$ l'ensemble de ses symboles terminaux et $\mathcal{V}$ l'ensemble de ses symboles non terminaux. L'ensemble des règles de la grammaires est noté $\mathcal{R}$. Chaque règle grammaticale associe à un symbole non terminal une séquence de symboles terminaux ou non terminaux associés à un exposant. Ainsi, $\mathcal{R}$ est une fonction de $\mathcal{V}$ dans $\left((\mathcal{T} \cup \mathcal{V}) \times \mathbb{N}^*\right)^+$. Avec ces notations, une grammaire est un quadruplet $\langle \mathcal{T}, \mathcal{V}, \mathcal{R}, R \rangle$ avec $R \in \mathcal{V}$ le symbole racine à partir duquel la trace d'exécution complète peut être reconstruite.

Pour chaque symbole s de $\mathcal{T} \cup \mathcal{V}$, nous distinguons chacune de ses apparitions dans la grammaire par un indice différent. Ainsi, la règle $r = s \mapsto \langle \langle A, 1 \rangle, \langle B, 2 \rangle, \langle A, 1 \rangle \rangle$ peut être écrite $s \mapsto A_0^1 B_0^2 A_1^1$ avec en exposant de chaque symbole le nombre de répétitions successives et en indice un nombre permettant d'identifier de manière

unique les différentes apparitions d'un même symbole, par exemple ici A_0 et A_1. Selon le contexte, les exposants égaux à 1 ainsi que les indices peuvent être omis, de sorte que nous pouvons également l'écrire $s \mapsto A_0 B_0^2 A_1$, ou encore $s \mapsto AB^2A$. De plus, pour une utilisation s_i du symbole s, nous notons $repeats(s_i)$ son nombre de répétitions successives, c'est-à-dire son exposant.

Pour $a, b \in \mathcal{T} \cup \mathcal{V}$, nous notons $a \prec b$, le fait que a soit immédiatement suivi de b dans une séquence. Par exemple, dans la séquence $w = \langle w_1 w_2 w_3 ... w_{len(w)} \rangle$, nous avons $\forall i \in [1, len(w) - 1], w_i \prec w_{i+1}$.

Reformulation des contraintes sur la grammaire

Ce formalisme permet de reformuler les quatre contraintes présentées dans la section 3.3.2 :

— unicité des digrammes : $\forall a, b \in \mathcal{T} \cup \mathcal{V}, \forall i, j, k, l \in \mathbb{N}, a_i \prec b_j \land a_k \prec b_l \implies i = k \land j = l$;

— pertinence des règles : $\forall \langle s, \alpha \rangle \in \mathcal{R}, s \neq R \implies \sum_i repeats(s_i) > 1$;

— utilité des règles : $\forall \langle s, \alpha \rangle \in \mathcal{R}, s \neq R \implies len(\alpha) \geq 2$;

— réduction des boucles : $\forall a, b \in \mathcal{T} \cup \mathcal{V}, \forall i, j \in \mathbb{N}, a_i \prec b_j \implies a \neq b$.

3.4.3 Algorithme de réduction

Étant donnée une séquence de symboles terminaux envoyés par le *runtime*, PYTHIA construit une grammaire représentant la trace d'exécution résultante. Cette grammaire est construite incrémentalement par l'algorithme 1 conformément aux contraintes énoncées à la section 3.3.2.

Algorithme 1 Réduction d'une trace.

```
1: fonction REDUCETRACE(w = ⟨w_0 w_1 ... w_len(w)⟩)
2:     Créer une grammaire G = ⟨T, V = {R}, R = {R ↦ w_0}, R⟩
3:     pour i ← 1, len(w) faire
4:         appendSymbol(R, w_i)
5:         i ← i + 1
6:     fin pour
7:     retourner G
8: fin fonction
```

L'initialisation consiste en la création d'une grammaire ne reconnaissant que le premier symbole de la trace. Les symboles sont ensuite ajoutés un par un à la fin de la règle racine à l'aide de la fonction appendSymbol, qui est décrite dans l'algorithme 2. Elle prend en entrée deux arguments : une règle r de la grammaire et un symbole n. Si r reconnaît une séquence $w = \langle w_0 w_1 ... w_{len(w)} \rangle$, appendSymbol modifie la grammaire afin que r reconnaisse désormais la séquence $w' = \langle w_0 w_1 ... w_{len(w)} n \rangle$. La fonction appendSymbol est récursive (cf. lignes 21, 24, 29, 34 et 42).

Soit $r = (s \mapsto w_0^{i_0} w_1^{i_1} ... w_k^{i_k}) \in \mathcal{R}$, nous nommons « dernier élément de r » (avant insertion de n) le symbole $p_i = w_k^{i_k}$. Si $p_i = n$, nous répétons simplement le dernier élément p_i de r en incrémentant son nombre de répétitions successives afin de respecter la contrainte d'absence de répétitions successives (lignes 3 à 4). Si $p_i \neq n$, mais qu'un digramme $p_j \prec n_k$ est déjà présent dans la grammaire, nous ne pouvons pas insérer n à la fin de r sans enfreindre la contrainte d'absence de répétition des digrammes. Nous cherchons donc à utiliser une règle de substitution représentant ce digramme (lignes 5 à 44). Enfin, si $p \neq n$ et que le digramme $p \prec n$ est absent de la grammaire, nous ajoutons simplement n après p_i dans la définition de r (ligne 46).

Dans le cas où $p \neq n$ et qu'un digramme $p_j \prec n_k$ est déjà présent dans la grammaire, nous modifions la grammaire pour que les nombres de répétitions successives de p_i et p_j soient égaux, et que le nombre de répétitions

Algorithme 2 Ajout d'un symbole à une séquence.

1: **fonction** APPENDSYMBOL($r \in \mathcal{R}, n \in \mathcal{T} \cup \mathcal{V}$)
2: Soit p_i le dernier élément de r
3: **si** $p = n$ **alors** → Simplification des répétitions
4: $repeats(p) \leftarrow j + 1$
5: **sinon si** $\exists j, k \in \mathbb{N}$ tels que $p_j \prec n_k$ **alors** → p_j garanti unique par la règle d'unicité des digrammes
6: $l_p \leftarrow \min(repeats(p_i), repeats(p_j))$
7: **si** $repeats(p_i) < repeats(p_j)$ **alors** → Égalisation des nombres de répétitions de p_i et p_j
8: Remplacer p_j par $p^{repeats(p_j)-l_p} p_j^{l_p}$
9: **sinon si** $repeats(p_i) > repeats(p_j)$ **alors**
10: Remplacer p_i par $p^{repeats(p_i)-l_p} p_i^{l_p}$
11: **fin si**
12: **si** $repeats(n_k) > 1$ **alors**
13: Remplace n_k par $n_k^1 n^{repeats(n_k)-1}$
14: **fin si**
15:
16: **si** $p \notin T \wedge m = j = 1 \wedge |\{p_x\}| = 2$ **alors** → Règle utilisée deux fois
17: **si** $\exists s \in \mathcal{V}$ tel que $(r_{pn} = s \mapsto p_j n_k) \in \mathcal{R}$ **alors** → Fusion de règles
18: Soit α tel que $(p \mapsto \alpha) \in \mathcal{R}$
19: Supprimer p_i de r et remplacer p_j par α dans r_{pn}
20: Retirer p de $\mathcal{V}$ et $(p \mapsto \alpha)$ de $\mathcal{R}$
21: appendSymbol(r, s)
22: **sinon** → Extension de règle
23: Soit $r_p = (p \mapsto \alpha) \in \mathcal{R}$
24: appendSymbol(r_p, n)
25: **si** $n_k \prec p_i$ **alors**
26: Remplacer $p_j n_k p_i$ par p^2
27: **sinon**
28: Supprimer p_i et n_k
29: appendSymbol(r, p)
30: **fin si**
31: **fin si**
32: **sinon si** $\exists s \in \mathcal{V}$ tel que $(s \mapsto p_j n_k) \in \mathcal{R}$ **alors** → Réutilisation d'une règle
33: Supprimer p_i de r
34: appendSymbol(r, s)
35: **sinon** → Création d'une nouvelle règle de substitution
36: Ajouter un nouveau symbole s à $\mathcal{V}$
37: Ajouter $s \mapsto p^{l_p} n_n^1$ à $\mathcal{R}$
38: **si** $n_k \prec p_i$ **alors**
39: Remplacer $p_j n_k p_i$ par s^2
40: **sinon**
41: Remplacer $p_j n_k$ par s^1 et supprimer p_i
42: appendSymbol(r, s)
43: **fin si**
44: **fin si**
45: **sinon**
46: Ajouter n^1 à la fin de r → Ajout sans simplification
47: **fin si**
48: **fin fonction**

successives de n_k soit égal à un (lignes 6 à 14). Par exemple, pour $repeats(p_i) = 3$, la séquence $p_j^5 n_k^2$ est remplacée par $p^2 p_j^3 n_k^1 n^1$ afin de faire apparaître le digramme $p_j^3 \prec n_k^1$. Ceci amène à enfreindre temporairement la contrainte d'absence de répétitions. Trois cas sont ensuite à distinguer :

— si le symbole p n'est utilisé que deux fois dans la grammaire, alors la règle de grammaire associée à p peut être modifiée pour y intégrer n. Cependant, s'il existe une règle de substitution $q = s \mapsto p_j n_k$ reconnaissant exactement le digramme $p \prec n$, l'intégration de n à la règle de substitution associée à p transforme la règle q en $s \mapsto p$, en violation de la contrainte d'utilité des règles. Dans ce cas, nous fusionnons les règles q et p (lignes 17 à 21) pour supprimer la règle p. Dans le cas où il n'existe pas de règle $q = s \mapsto p_j n_k$ reconnaissant exactement le digramme $p \prec n$, nous appelons récursivement appendSymbol sur p et n (lignes 23 à 30) ;

— si le symbole p est utilisé plus de deux fois dans la grammaire, mais qu'il existe une règle de substitution $s \mapsto p_j n_k$ reconnaissant exactement le digramme $p \prec n$, alors nous réutilisons le symbole s en retirant p_i de r et en appelant récursivement appendSymbol sur r et s (lignes 32 à 34) ;

— si le symbole p est utilisé plus de deux fois dans la grammaire et qu'il n'existe pas de règle reconnaissant exactement le digramme $p \prec n$, nous créons un nouveau symbole non-terminal s et une nouvelle règle de substitution $s \mapsto pn$ (lignes 36 à 43).

3.4.4 Implémentation

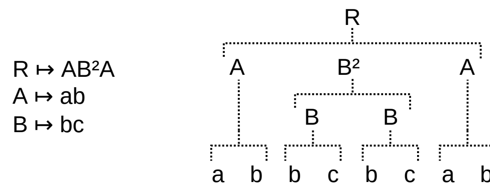
PYTHIA[1] est implémenté en C++ avec une interface compatible avec les programmes écrits en langage C. L'API de PYTHIA est décrite à l'annexe A. L'API dédiée à l'enregistrement de message consiste en deux fonctions. La fonction `pythia_new_terminal` permet au *runtime* de déclarer un nouveau symbole terminal. La fonction `pythia_append_symbol` permet au *runtime* d'ajouter un nouvel événement à la trace en ajoutant le symbole terminal correspondant à l'événement.

Représentation d'une grammaire en mémoire

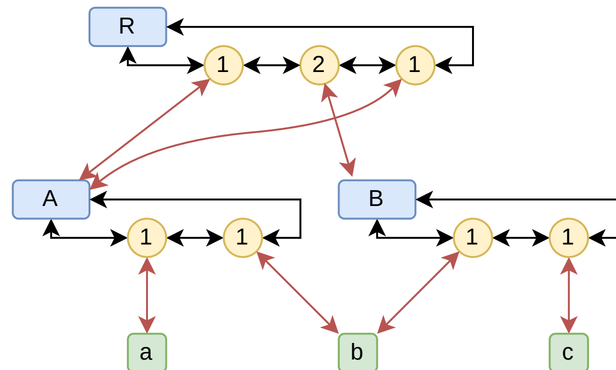
La figure 3.8 montre par un exemple comment PYTHIA représente une trace sous forme de grammaire. La grammaire considérée représente la séquence « $abbcbcab$ » (cf. figure 3.8a). La structure de donnée en mémoire est représentée par la figure 3.8b. Les symboles terminaux sont représentés en vert dans des petites boîtes et les symboles non terminaux sont représentés en bleu dans des grandes boîtes. Les symboles non terminaux référencent le début et la fin de la séquence de symboles qu'ils représentent, et qui est stockée sous la forme d'une liste doublement chaînée. Chaque maillon de cette liste chaînée représente l'utilisation d'un symbole dans la règle grammaticale correspondante. Cette utilisation d'un symbole est matérialisée par un pointeur vers le symbole correspondant (les flèches verticales en rouge sur la figure) et un entier représentant son nombre de répétitions consécutives (les disques en jaune).

Chaque symbole, qu'il soit terminal ou non terminal, contient une liste de ses utilisations sous la forme d'une table de hachage. Chaque utilisation d'un symbole est indexée dans cette table par le symbole qui la suit dans la règle grammaticale correspondante. Ainsi, à partir d'un symbole a, il est possible de savoir en temps constant si un digramme dont a est le premier élément existe dans la grammaire et où. Par ailleurs, chaque symbole non terminal contient un pointeur vers le dernier symbole de la séquence associée. Ceci permet d'accéder quasi-directement à la fin d'une séquence, par exemple, pour y ajouter un symbole. De même, il est possible d'accéder au symbole non terminal depuis le dernier symbole de la séquence associée. Ceci permet par exemple, une fois arrivé à la fin d'une séquence, de chercher toutes les occurrences de cette séquence dans la grammaire.

1. Disponible sous licence libre à cette adresse : https://gitlab.com/parallel-and-distributed-systems/easytraceanalyzer/eta-factorizer



(a) Représentation sous forme de grammaire et dépliage de celle-ci.



(b) Structure de donnée en mémoire.

FIGURE 3.8 – Écriture dans un fichier de la grammaire représentant la séquence *abbcbcab*.

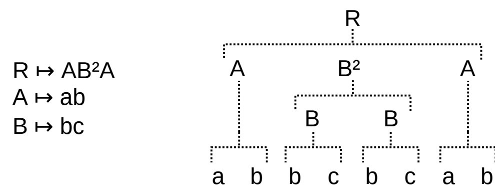
Représentation d'une trace d'exécution réduite dans un fichier

À la fin de l'exécution du programme, PYTHIA sauvegarde la trace d'exécution réduite sous la forme d'une grammaire. La figure 3.9b montre comment PYTHIA écrit la trace représentant la séquence « $abbcbcab$ » (figure 3.9a) dans un fichier. En entête du fichier, PYTHIA inscrit le nombre de symboles terminaux et non terminaux à lire. PYTHIA associe ensuite à chaque symbole un identifiant unique sous la forme d'un entier attribué dans l'ordre, en commençant par les symboles terminaux et en terminant par les symboles non terminaux. Ainsi, si la grammaire comprend n symboles terminaux et m symboles non terminaux, les symboles terminaux seront numérotés de 0 à $n-1$ et les symboles terminaux de n à $n+m-1$.

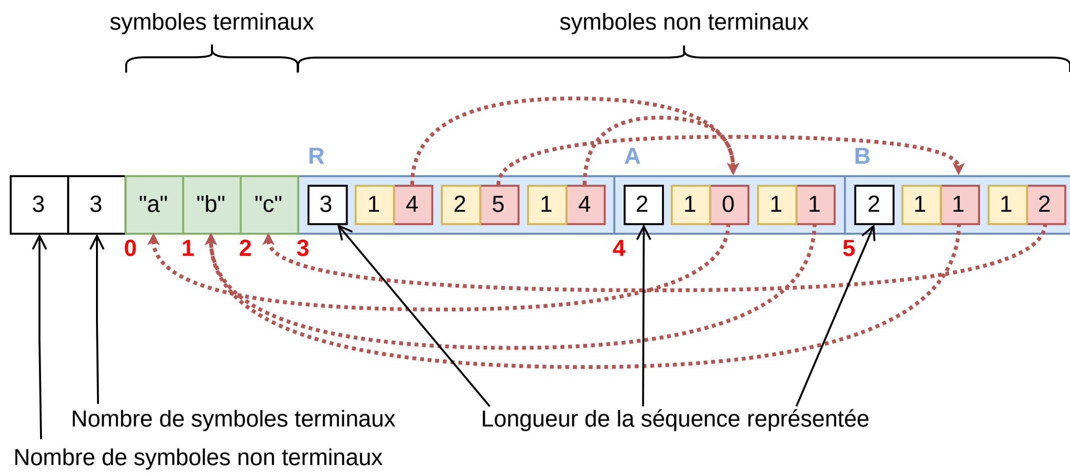
Le reste du fichier est séparé en deux sections. La première section concerne les symboles terminaux (avec des guillemets anglais et en vert dans la figure). Pour chaque symbole terminal, une description du symbole est écrite afin de permettre lors des exécutions futures de faire correspondre le symbole à l'événement correspondant dans les prédictions fournies par PYTHIA. La seconde section décrit les règles grammaticales de la grammaire et les séquences correspondantes à chaque symbole non terminal (écrites en bleu au dessus des zones coloriées en bleu dans la figure). Chaque séquence est linéarisée, avec, pour chaque symbole utilisé dans la séquence, son identifiant unique (écrit dans la seconde case, qui est coloriée en rouge) précédé de son nombre de répétitions successives (écrit dans la première case, qui est coloriée en jaune).

3.5 Conclusion

Dans ce chapitre, nous avons passé en revue différentes méthodes permettant d'instrumenter un programme et de capturer la structure de son exécution. Au regard de nos objectifs, nous avons alors choisi d'utiliser des



(a) Représentation sous forme de grammaire et dépliage de celle-ci.



(b) Représentation dans un fichier.

FIGURE 3.9 – Représentation en mémoire de la grammaire représentant la séquence *abbcbcab*.

grammaires pour représenter les traces d'exécution collectées par PYTHIA-RECORD. Dans la suite du chapitre, nous avons présenté la première contribution de cette thèse.

Dans un premier temps, nous avons décrit précisément comment une grammaire pouvait être utilisée pour représenter une trace d'exécution. Nous avons alors formulé quatre contraintes que les grammaires représentants des traces d'exécution doivent respecter afin de représenter la structure de l'exécution du programme.

Dans un second temps, nous avons présenté l'algorithme que nous avons conçu et qui permet de réduire à la volée une trace d'exécution sous la forme d'une grammaire respectant les quatre contraintes que nous avons énoncées.

Enfin, nous avons décrit l'implémentation des grammaires et de l'algorithme de réduction au sein de PYTHIA-RECORD, ainsi que la façon dont les grammaires sont sauvegardées pour être réutilisées par notre oracle lors des exécutions ultérieures du programme.

Chapitre 4

PYTHIA-PREDICT : prévision du comportement futur d'un programme

Sommaire

4.1 Suivi de l'exécution du programme : le cas de deux exécutions identiques . . . 41
4.1.1 Suivi de l'exécution du programme par reconstruction de la trace par substitutions . . . 41
4.1.2 Suivi de l'avancement de l'exécution sans décompression . . . 43
4.1.3 Représentation de l'état d'avancement du programme . . . 45
4.2 Comparaison d'exécutions avec des variations . . . 47
4.2.1 Adaptation de PYTHIA face à une variation du comportement du programme . . . 47
4.2.2 Recherche de séquences au sein d'une trace réduite . . . 48
4.2.3 Point de progression représentant une connaissance incomplète de l'avancement du programme 48
4.2.4 Discussion . . . 51
4.3 Prévoir le comportement futur du programme . . . 51
4.3.1 Prévision des événements à venir . . . 51
4.3.2 Calcul de la probabilité de chaque événement . . . 52
4.3.3 Enrichissement des prévisions . . . 54
4.4 Conclusion . . . 55

Dans le chapitre précédent, nous avons décrit comment PYTHIA-RECORD permet à un *runtime* d'enregistrer des événements afin de constituer une trace d'exécution. Pour cela, PYTHIA-RECORD utilise une grammaire permettant de représenter la structure de l'exécution du programme. À partir de maintenant, cette exécution dont le comportement a été enregistré est nommée « exécution de référence ». Dans ce chapitre, nous présentons comment le module de prédiction de PYTHIA, nommé PYTHIA-PREDICT (à droite dans la figure 4.1), utilise cette grammaire pour suivre l'avancement de l'exécution d'un programme et prévoir son comportement futur.

Le principe d'utilisation de PYTHIA-RECORD et PYTHIA-PREDICT est le suivant. À la fin de l'exécution de référence, PYTHIA-RECORD sauvegarde la grammaire représentant la structure de la trace d'exécution constituée par tous les événements qui lui ont été rapportés par le *runtime*. Lorsqu'un utilisateur ré-exécute la même application, il peut fournir à PYTHIA-PREDICT la trace de cette exécution de référence. Cette grammaire est chargée par PYTHIA-PREDICT au début de la nouvelle exécution du programme. Lors de cette nouvelle exécution, le *runtime* soumet des événements à PYTHIA-PREDICT, qui suit l'avancement de l'exécution du programme relativement à l'exécution de référence.

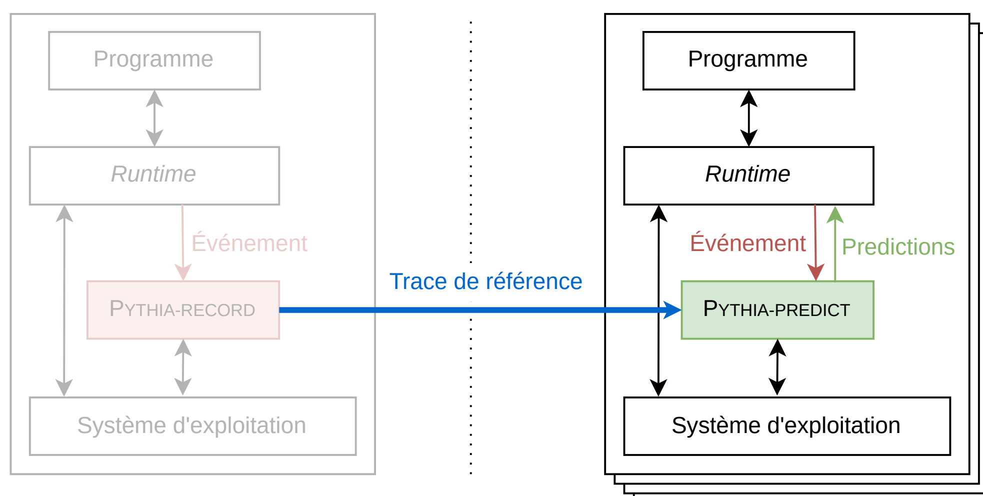


FIGURE 4.1 – Intégration de PYTHIA-PREDICT dans l'outil PYTHIA.

En comparant l'exécution du programme à son exécution de référence, PYTHIA-PREDICT est en capacité de fournir au *runtime* des prévisions sur le comportement futur du programme. Ces prédictions indiquent les prochains événements à venir, ce qui peut être exploité par le *runtime* pour sa prise de décision.

Dans les sections 4.1 et 4.2, nous présentons comment PYTHIA-PREDICT suit l'avancement de l'exécution du programme dans le cas où le comportement du programme est identique à son exécution de référence et dans le cas où les deux exécutions diffèrent. Ensuite, dans la section 4.3, nous montrons comment PYTHIA-PREDICT prévoit le comportement futur du programme et comment ces prévisions peuvent être enrichies. Enfin, nous concluons le chapitre dans la section 4.4.

4.1 Suivi de l'exécution du programme : le cas de deux exécutions identiques

Lors de l'exécution d'un programme, PYTHIA-PREDICT utilise la grammaire de référence produite par PYTHIA-RECORD pour suivre le déroulement de l'exécution relativement à celle de référence. Dans le cas où l'exécution en cours du programme est similaire à l'exécution de référence, la séquence d'événements générée par le *runtime* est identique à celle enregistrée dans la trace d'exécution. Suivre l'avancement du programme consiste alors à faire correspondre les deux séquences d'événements.

Nous voyons dans cette section comment procéder pour suivre l'avancement de l'exécution d'un programme quand cette dernière est identique à son exécution de référence. Nous montrons comment suivre l'exécution d'un programme avec une grammaire par reconstruction (section 4.1.1) puis sans reconstruction (section 4.1.2) avant de présenter plus précisément comment représenter et mettre à jour l'avancement de l'exécution d'un programme (section 4.1.3).

4.1.1 Suivi de l'exécution du programme par reconstruction de la trace par substitutions

Par définition, reconstruire la séquence complète d'événements représentée par la grammaire consiste à effectuer, au sein de la séquence associée au symbole racine, les substitutions correspondant aux règles grammaticales, et ce jusqu'à obtenir une séquence constituée exclusivement de symboles terminaux. La figure 4.2 illustre

ce procédé pour la trace « $abcabdabcabdabc$ ». La grammaire initiale est composée de quatre règles grammaticales (figure 4.2a). Tout d'abord, les répétitions consécutives doivent être dépliées (figure 4.2b). Les différentes règles grammaticales sont ensuite utilisées pour remplacer les différents symboles non terminaux jusqu'à ce que la règle racine R ne comporte plus aucun symbole non terminal (figures 4.2c, 4.2d et 4.2e). La séquence associée au symbole racine est alors la trace représentée initialement par la grammaire.

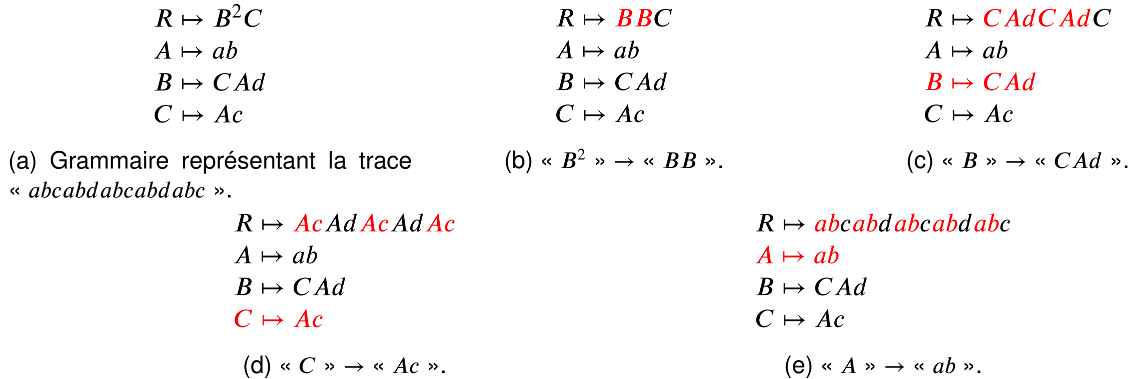


FIGURE 4.2 – Reconstruction par substitutions de la séquence « $abcabdabcabdabc$ » depuis la grammaire la représentant.

Mais, cette méthode possède le double inconvénient de nécessiter une phase de traitement sur les données chargées afin de les rendre utilisables. De plus, elle fait perdre l'avantage du faible encombrement mémoire de la trace réduite par rapport à une trace non réduite.

Ces inconvénients peuvent être amoindris en appliquant cette méthode de manière partielle tout au long de l'exécution. En effet, pour suivre l'exécution du programme, il n'est pas nécessaire de disposer de la totalité de la trace à tout instant. La figure 4.3 montre comment procéder avec l'exemple d'une trace représentant la séquence « $abcab$ ». La grammaire correspondante comprend deux règles grammaticales (figure 4.3a). Nous souhaitons lire au sein de cette grammaire la séquence « $abcab$ » représentée par la règle racine R. Dans un premier temps, la séquence ab (en bleu) associée au symbole A doit être substituée à A_0 du symbole non terminal A. Le premier symbole, que nous soulignons dans la figure 4.3b, est alors lisible : il s'agit du symbole a. Afin de lire le deuxième symbole de la trace, il suffit de lire le symbole suivant au sein de la règle racine (figure 4.3c). Ensuite, pour lire le symbole c_0, il n'est plus nécessaire de décompresser A_0 qui vient à nouveau remplacer la séquence « ab » au début de la règle racine (figure 4.3d). Les deux symboles suivants peuvent être lus en substituant la séquence « ab » à A_1 (figures 4.3e et 4.3f).

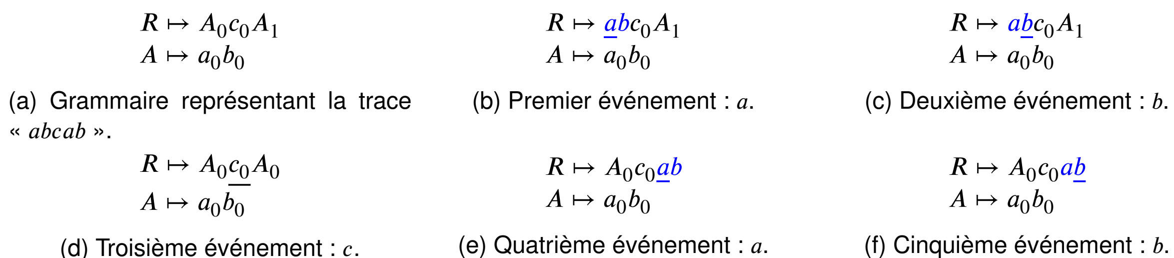


FIGURE 4.3 – Suivi de l'exécution d'un programme générant la séquence « $abcab$ » par substitutions partielles.

Cette méthode, qui consiste à suivre l'exécution du programme par reconstruction de la trace, permet de réduire les problèmes sous-jacents à la reconstruction complète de la trace d'exécution représentée par la grammaire. Cependant, elle peut impliquer de nombreuses modifications en mémoire pour chaque symbole lu, en particulier

dans le cas de grammaires présentant de nombreuses règles grammaticales imbriquées ou des symboles avec de grands nombres de répétitions. Dans la suite, nous présentons une méthode alternative permettant de parcourir une grammaire sans la modifier.

4.1.2 Suivi de l'avancement de l'exécution sans décompression

Une solution alternative à la reconstruction par substitutions est d'effectuer un parcours en profondeur de la grammaire. L'idée est illustrée par la figure 4.4. Dans la représentation en forme de graphe de la grammaire, le successeur d'un symbole peut être trouvé en remontant parmi ses parents jusqu'au premier symbole ayant un successeur dans une règle grammatical. Ce parcours est en fait conceptuellement très proche du parcours de la trace par substitution partielle présenté dans la section 4.1.1.

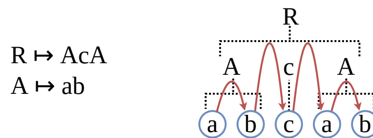


FIGURE 4.4 – Intuition de la reconstruction d'une trace par parcours d'une grammaire.

Par souci de clareté de l'exposé, nous traitons séparément le cas des grammaires sans répétitions du cas des grammaires avec répétitions.

Le cas des grammaires sans répétions

Nous nous intéressons tout d'abord au cas de grammaires ne mettant pas en jeu de répétitions, c'est-à-dire dans lesquelles tous les symboles utilisés ont un nombre de répétions égal à 1. Dans ce cas, l'algorithme de parcours de la grammaire dérive directement du parcours par reconstruction partielle. Dans cet algorithme, nous représentons une occurrence particulière d'un symbole terminal par un « chemin » dans la grammaire. À titre d'exemple, le troisième symbole b de la séquence « $ababccabcab$ » peut être représenté par le chemin dessiné par la figure 4.5.

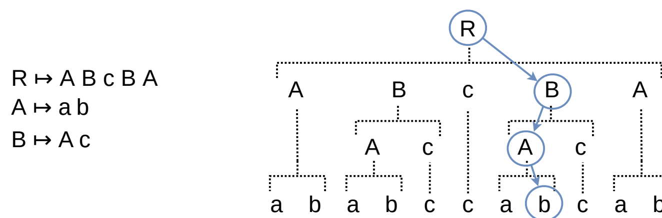


FIGURE 4.5 – Chemin représentant la troisième occurrence du symbole b de la séquence « $ababccabcab$ ».

Toujours avec un exemple, la figure 4.6 illustre comment parcourir et ainsi suivre l'exécution d'un programme à l'aide de chemins représentants les différents événements d'une trace. La trace représentée dans l'exemple est la séquence « $abcab$ ». La grammaire correspondante consiste en deux règles (figure 4.6a). Le chemin représentant le premier événement de la trace est composé du symbole racine R, puis de A_0, premier élément de la séquence associée à R, puis de a_0, premier élément de la séquence associée à A (figure 4.6b). Le terminal b_0 est le successeur de a_0 au sein de la séquence associée à A. Par conséquent, pour représenter le second événement de la trace, nous pouvons remplacer a_0 par b_0 au sein du chemin (figure 4.6c). Contrairement à a_0, b_0 n'a pas de successeur au sein de cette séquence. Donc, pour représenter le troisième événement de la trace, nous devons retirer b_0 puis

A_0 afin d'atteindre le successeur c_0 (figure 4.6d). Ensuite, comme le successeur de c_0 est A_1, avec le symbole A qui est un symbole non terminal, nous remplaçons c_0 par le chemin A_1 complété de a_0 (figure 4.6e). Pour finir, nous remplaçons a_0 par b_0 pour représenter le dernier événement de la trace (figure 4.6f).

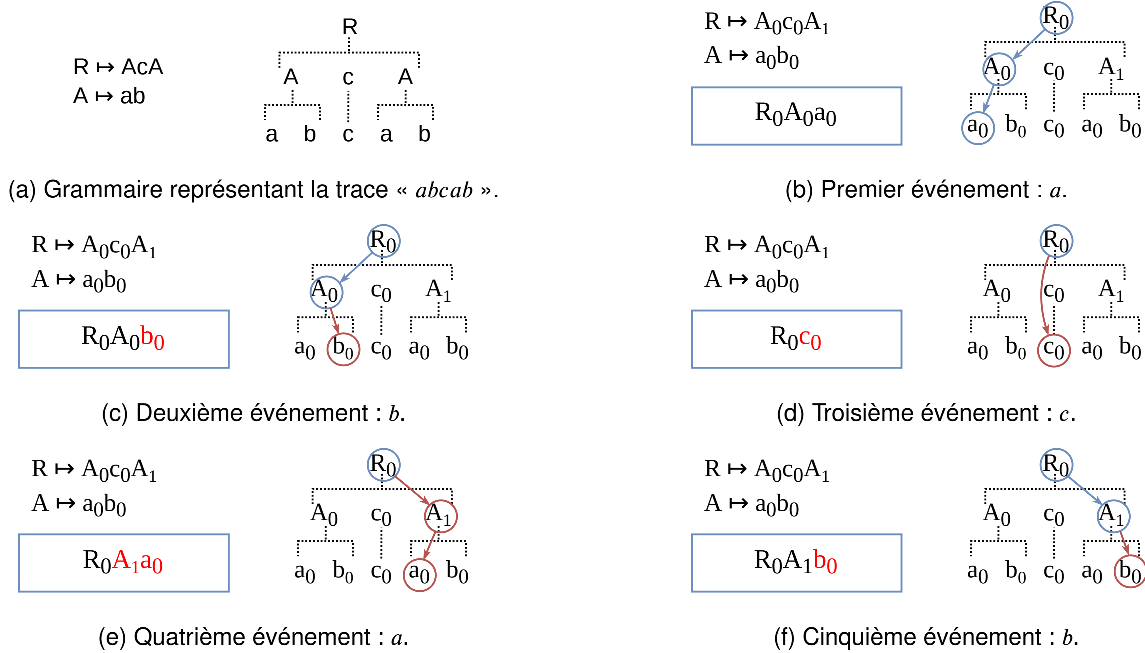
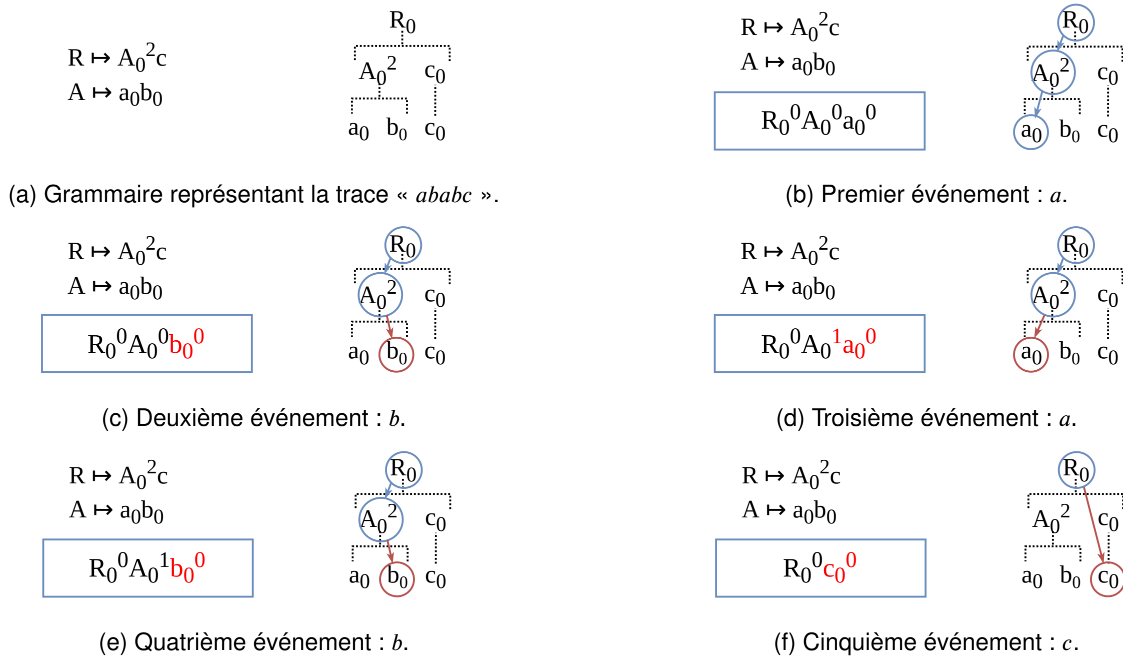


FIGURE 4.6 – Parcours d'une trace représentée par une grammaire sans décompression.

Suivi de l'exécution avec des répétitions

La prise en compte des répétitions requiert l'ajout, à chaque maillon du chemin, d'un nombre de répétitions « déjà rencontrées ». Le nombre de répétitions permet de décider si le passage d'un événement au suivant demande de remplacer un élément du chemin par son successeur, ou de relire une nouvelle fois le chemin associé. Comme pour les symboles au sein des séquences, nous notons ce nombre de répétitions, mais cette fois-ci en exposant.

La figure 4.7 illustre ce fonctionnement par un exemple. La grammaire considérée représente la séquence « *ababc* » et contient deux règles grammaticales (figure 4.7a). Comme dans l'exemple précédent, le premier événement est représenté par un chemin correspondant à une « descente en profondeur par la gauche » de la représentation sous forme de graphe de la grammaire. Chaque élément de ce chemin est associé à un nombre de répétitions valant 0 puisque chacun est rencontré pour la première fois dans le parcours de la trace (figure 4.7b). Le terminal a_0 étant répété une seule fois, le passage au deuxième élément de la trace est effectué comme dans l'exemple précédent (figure 4.7c). Ensuite, puisque b_0 n'a pas de successeur dans la grammaire, celui-ci est retiré du chemin qui se termine désormais par A_0^0. Le non-terminal A_0 devant être répété deux fois, la séquence correspondante doit être lue une seconde fois et nous remplaçons donc A_0^0 par A_0^1. En outre, pour relire la séquence associée à A, nous ajoutons a_0^0 au chemin, signifiant ainsi que le troisième événement de la trace correspond au symbole terminal a qui est, à ce moment, rencontré pour la première fois dans le contexte de la deuxième lecture de A_0 (figure 4.7d). Le passage au quatrième événement de la trace est similaire à la lecture du second événement de la trace (figure 4.7e). Comme lors de la lecture du troisième événement de la trace, b_0 n'a pas de successeur au sein de la grammaire et est retiré du chemin. Le dernier élément du chemin est alors A_0^1, correspondant à l'élément A_0 de la grammaire dont le nombre de répétitions est 2. La séquence associée à A a déjà été parcourue deux fois ; A_0^1 est donc remplacé par c_0, son successeur dans la grammaire (figure 4.7f).



(a) Grammaire représentant la trace « *ababc* ».

(b) Premier événement : a.

(c) Deuxième événement : b.

(d) Troisième événement : a.

(e) Quatrième événement : b.

(f) Cinquième événement : c.

FIGURE 4.7 – Parcours d'une grammaire contenant une boucle.

Nous avons vu qu'il était possible de lire la trace d'exécution représentée par une grammaire, et par conséquent de suivre le déroulement d'une exécution qui serait similaire à l'exécution de référence. Dans la section qui vient, nous définissons comment est représenté l'état d'avancement du programme dans la grammaire en introduisant le concept de point de progression.

4.1.3 Représentation de l'état d'avancement du programme

Dans cette section, nous détaillons l'algorithme permettant de suivre l'exécution d'un programme en utilisant un point de progression. Nous définissons dans un premier temps la représentation et les opérations de base permettant de manipuler ces points de progression,puis nous expliquons comment initialiser un point de progression et comment le mettre à jour au fil de l'exécution.

Point de progression

Rappelons qu'une utilisation d'un symbole dans une règle grammaticale est représentée comme un élément de l'ensemble $(\mathcal{T} \cup \mathcal{V}) \times \mathbb{N}^*$, et est noté a_i, avec $a \in \mathcal{T} \cup \mathcal{V}$ et $i \in \mathbb{N}^*$. Un point de progression est une suite finie d'utilisations de symboles au sein de la grammaire, chacune associée à un nombre de répétitions. Un point de progression est donc un élément de l'ensemble $((\mathcal{T} \cup \mathcal{V}) \times \mathbb{N}^* \times \mathbb{N})^*$. Formellement, nous notons le point de progression σ sous la forme $\sigma = a_1^{i_1} ... a_n^{i_n}$. Le point de progression σ peut être vide et est alors noté $\sigma = \emptyset$. Remarquons que jusqu'à présent nous n'avons discuté que de points de progression commençant par le symbole racine de la grammaire : un tel point de progression repère l'état d'avancement de la trace depuis le début de l'exécution. Dans la suite, nous étudions des exemples de points de progression ne représentant pas l'avancement du programme depuis le début de son exécution, mais contenant seulement les derniers événements survenus.

Étant donné le point de progression $\sigma = a_1^{i_1} ... a_n^{i_n}$, nous définissons trois opérations sur les points de progression :

— $push(\sigma, a_i^j)$ permet d'ajouter un nouvel élément a_i^j à la fin du point de progression σ ;

— $pop(\sigma)$ permet de supprimer le dernier élément du point de progression σ s'il n'est pas vide ;

— $last(\sigma)$ permet d'accéder au dernier élément du point de progression sans le supprimer.

Initialisation de l'état d'avancement

L'initialisation d'un point de progression sur le premier événement consiste à ajouter récursivement le premier élément de la séquence racine, puis le premier élément de la séquence associée au symbole non terminal correspondant, et ainsi de suite jusqu'à atteindre un symbole terminal. Au sein du graphe représentant la grammaire dépliée, cela consiste à descendre de la racine vers le terminal « le plus à gauche ».

Dans l'algorithme 3, nous montrons comment initialiser un point de progression pour représenter le premier événement d'une trace réduite. La fonction correspondante, `InitialisationAvancement`, prend en argument une grammaire $G = \langle \mathcal{T}, \mathcal{V}, \mathcal{R}, R \rangle$ (ligne 1) et retourne le point de progression construit à l'aide de la fonction `DescenteAGauche` à partir de son symbole racine (algorithme 4, ligne 2).

Algorithme 3 Initialisation de l'état d'avancement du programme.

1: **fonction** INITIALISATIONAVANCEMENT($G = \langle \mathcal{T}, \mathcal{V}, \mathcal{R}, R \rangle$)
2: **retourner** $DescenteAGauche($« R_0^0 »$)$
3: **fin fonction**

Dans l'algorithme 4, nous décrivons comment compléter un point de progression afin de représenter le premier symbole terminal de la séquence associée à son dernier élément. La fonction `DescenteAGauche` prend en argument un point de progression non vide (ligne 1). Si le dernier élément du point de progression est un symbole non terminal (ligne 3), nous ajoutons le premier élément de la séquence associée (ligne 5), nous appelons récursivement `DescenteAGauche`, et nous retournons le résultat (ligne 6). Dans le cas contraire, le point de progression est retourné car un symbole terminal a été atteint (ligne 8).

Algorithme 4

1: **fonction** DESCENTEAGAUCHE($\sigma \neq \emptyset$)
2: Soit $s_i^j = last(\sigma)$
3: **si** $s \in \mathcal{V}$ **alors**
4: Soit a_k^l le premier élément de la séquence associée à s
5: $push(\sigma, a_k^0)$
6: **retourner** $DescenteAGauche(\sigma)$
7: **sinon**
8: **retourner** σ
9: **fin si**
10: **fin fonction**

Algorithme pour le suivi de l'état d'avancement d'une exécution

Dans le cas où la séquence d'événements soumise par le *runtime* est identique à celle représentée par la grammaire, suivre l'exécution du programme consiste simplement à lire la trace depuis la grammaire, événement par événement. Dans l'algorithme 5, nous détaillons comment mettre à jour le point de progression événement après événement afin qu'il représente l'état d'avancement du programme.

La fonction `ProchainEvenement` prend en argument un point de progression représentant un événement et renvoie un point de progression représentant l'événement suivant dans la trace représentée par la grammaire (ligne 1). Nous notons x_i^j le dernier élément du point de progression et nous l'en retirons (ligne 3 et 4). Si x_i représente une boucle et que celle-ci n'a pas été entièrement parcourue, nous le rajoutons à la fin du point de progression avec un nombre de répétitions augmenté de 1, et nous retournons le point de progression complété

Algorithme 5

```
1:  fonction PROCHAINEVENEMENT(σ)
2:      tant que true faire
3:          Soit x_i^j = last(σ) et n le nombre de répétitions de x_i dans la grammaire
4:          pop(σ)
5:          si j + 1 < n alors
6:              push(σ, x_i^{j+1})
7:              retourner DescenteAGauche(σ, x)
8:          sinon si ∃s_k, x_i ≺ s_k alors
9:              push(σ, s_k^0)
10:             retourner DescenteAGauche(σ, s)
11:         sinon si σ = ∅ alors
12:             retourner ∅
13:         fin si
14:     fin tant que
15: fin fonction
```

avec la fonction `DescenteAGauche` (lignes 5 à 7). Sinon, si x_i possède un successeur s_k dans la grammaire, nous ajoutons ce successeur au point de progression avec un nombre de répétitions égal à 0, et nous retournons le point de progression complété avec la fonction `DescenteAGauche` (lignes 8 à 10). Enfin, si x_i n'a pas de successeur, nous recommençons avec l'élément précédent du point de progression (ligne 2), excepté si $\sigma = \emptyset$, ce qui signifie que la fin de la trace a été atteinte (lignes 11 et 12).

Cet algorithme indique comment relire une trace d'exécution depuis une grammaire et permet donc de suivre l'exécution d'un programme dont l'exécution est identique à l'exécution de référence. Cet algorithme ne permet pas de suivre l'exécution d'un programme dont le comportement est différent de celui capturé pendant son exécution de référence. Dans la suite, nous étudions comment suivre l'exécution d'un tel programme.

4.2 Comparaison d'exécutions avec des variations

Le comportement d'un programme peut varier d'une exécution à l'autre. Par exemple, en changeant de jeu de données, certains chemins de code qui n'avaient pas été exécutés pendant une première exécution peuvent désormais être activés. Pour certaines applications, le comportement peut varier en fonction de paramètres difficilement prédictibles tels que l'ordre de réception de messages, ou de la position calculée d'un élément au sein d'une simulation.

Dans cette section, nous décrivons comment PYTHIA-PREDICT s'adapte aux variations de comportement des programmes (section 4.2.1) en cherchant des similitudes entre leurs exécutions (section 4.2.2). Nous redéfinissons la notion de points de progression (section 4.2.3) et discutons de l'applicabilité de cette approche (section 4.2.4).

4.2.1 Adaptation de PYTHIA face à une variation du comportement du programme

Du point de vue de PYTHIA-PREDICT, une variation du comportement du programme par rapport à son exécution de référence se manifeste par une séquence d'événements rapportée par le *runtime* différente de celle de la trace de référence. Certaines parties de la trace de référence peuvent par exemple être absentes de la séquence d'événements. Certaines parties de la trace d'exécution peuvent aussi être réordonnées et des événements qui n'étaient pas apparus lors de l'exécution de référence peuvent survenir.

Face à une telle variation, PYTHIA-PREDICT doit s'adapter. Il n'est plus possible de simplement faire coïncider événement pour événement la nouvelle séquence avec la trace de référence. PYTHIA-PREDICT cherche donc dans la trace de référence quelles parties pourraient correspondre à l'exécution d'un morceau de code similaire à celui en cours d'exécution. Pour cela, PYTHIA-PREDICT cherche, événement après événement, si la séquence d'événements

courante apparaît dans la grammaire. En effet, nous conjecturons que, même si le comportement du programme a changé, il est probable que les fonctions exécutées l'aient été aussi pendant l'exécution de référence, mais potentiellement plus tôt, ou plus tard dans l'exécution courante du programme.

En pratique, PYTHIA-PREDICT essaie de faire correspondre les deux séquences d'événements issues de l'exécution de référence et de l'exécution courante avec la méthode décrite dans la section 4.1.3. Si à un moment de l'exécution courante, le point de progression censé représenter l'avancement du programme représente un événement différent du dernier événement rapporté par le *runtime*, PYTHIA-PREDICT prend acte de la divergence et cherche dans quel contexte cet événement apparaît dans la grammaire. Ensuite, PYTHIA-PREDICT affine, événement par événement, la liste des endroits de la trace de référence pouvant correspondre à la nouvelle séquence d'événements rapportée par le *runtime*.

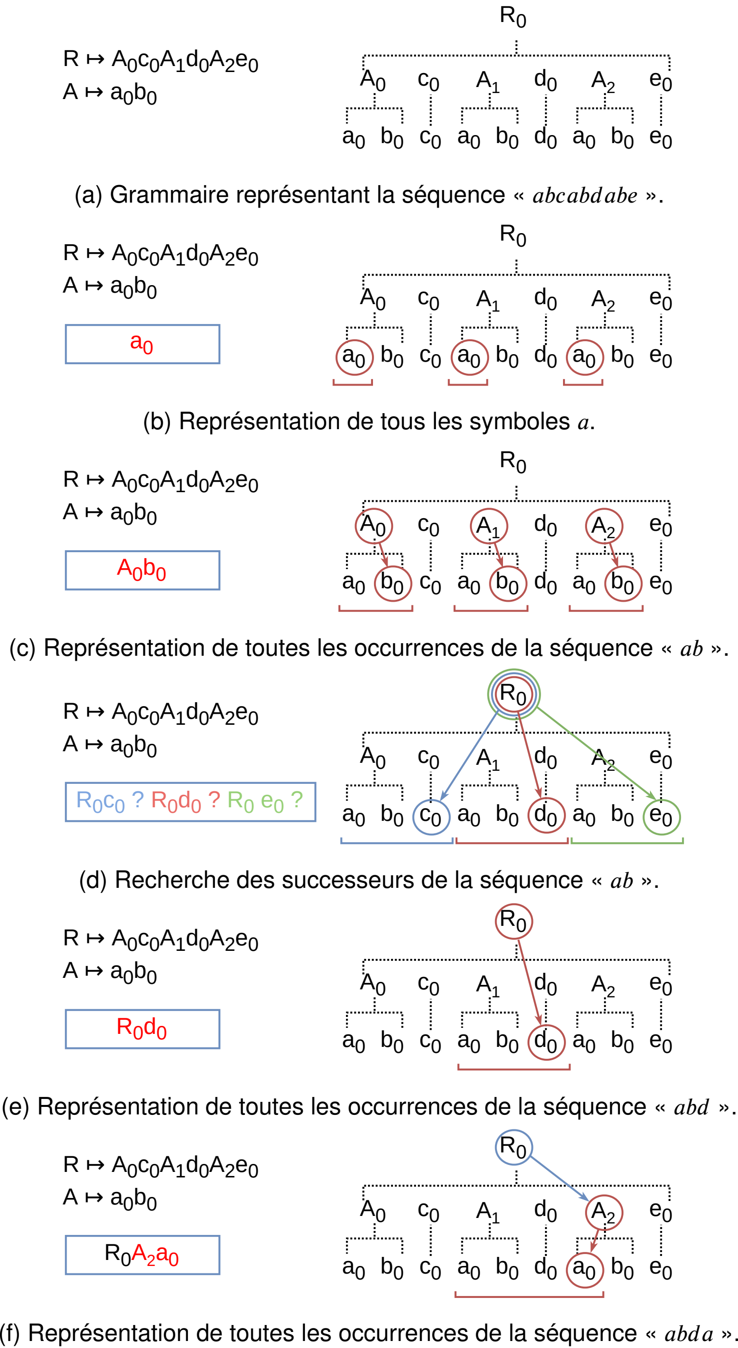
4.2.2 Recherche de séquences au sein d'une trace réduite

Considérons un événement a qui n'est attendu par aucun des points de progression courants. Quand un tel événement a survient, PYTHIA-PREDICT recherche la séquence constituée de ce seul événement dans la grammaire. PYTHIA-PREDICT liste donc toutes les occurrences du symbole a au sein des différentes règles grammaticales. Si l'événement survenant ensuite est associé au symbole b, PYTHIA-PREDICT cherche parmi les apparitions du symbole a repérées précédemment lesquelles sont suivies d'une séquence commençant par le symbole b. Ainsi, PYTHIA-PREDICT repère toutes les occurrences de la séquence « ab » au sein de la grammaire. Événement après événement, PYTHIA-PREDICT cherche ainsi à retrouver dans la trace la nouvelle séquence d'événement générée par le programme.

Pour illustrer le fonctionnement de cette recherche, prenons l'exemple de la séquence « $abda$ » au sein de la trace « $abcabdabe$ ». Cet exemple est dessiné dans la figure 4.8. Puisque la grammaire correspondante (figure 4.8a) ne contient aucune boucle, nous faisons ici l'économie de noter tout nombre de répétitions. Pour trouver une séquence au sein d'une trace, nous cherchons son premier symbole au sein de la grammaire. Ici, le symbole a est présent une seule fois dans la grammaire et représente trois occurrences de l'événement associé au sein de la trace (figure 4.8b). L'unique occurrence du symbole a, a_0, est suivi de b_0. Il est possible de représenter cette information en dessinant un chemin dans la grammaire reliant A_0 et b_0 (figure 4.8c). En procédant comme à l'étape précédente, nous construisons trois chemins dans la grammaire qui correspondent aux successeurs respectifs (figure 4.8d). Un seul de ces chemins représente une occurrence du symbole d. Les autres chemins ne correspondant pas à la séquence « abd », ils sont ignorés (figure 4.8e). On remarque que ce chemin est de la même forme que les points de progression vus dans la section 4.1.2. En appliquant l'algorithme permettant de passer d'un événement représenté par un point de progression à son successeur, nous trouvons le symbole a (figure 4.8f). Ceci nous permet de dire que la trace « $abcabdabe$ » contient bien la séquence « $abda$ ». De plus, nous sommes capable de situer cette séquence dans la trace à l'aide du point de progression.

4.2.3 Point de progression représentant une connaissance incomplète de l'avancement du programme

Dans l'exemple de la section précédente, nous pouvons remarquer que des chemins dans la grammaire ont été utilisés. Ces chemins diffèrent des points de progression tels que nous les avons utilisés jusqu'à présent en ce qu'ils ne remontent pas jusqu'à la racine de la grammaire. Un tel chemin permet de décrire la connaissance partielle dont dispose PYTHIA-PREDICT sur l'état d'avancement d'un programme quand celui-ci a dévié du comportement enregistré durant l'exécution de référence. Dans la suite du document, nous gardons le terme « point de progression », que le chemin relie ou non un terminal au symbole racine de la grammaire.



(a) Grammaire représentant la séquence « *abcabdabe* ».

(b) Représentation de tous les symboles *a*.

(c) Représentation de toutes les occurrences de la séquence « *ab* ».

(d) Recherche des successeurs de la séquence « *ab* ».

(e) Représentation de toutes les occurrences de la séquence « *abd* ».

(f) Représentation de toutes les occurrences de la séquence « *abda* ».

FIGURE 4.8 – Recherche de toutes les occurrences de la séquence « *abda* » au sein de la grammaire représentant la trace « *abcabdabe* ».

Représenter tous les occurrences d'une séquence au sein d'une grammaire sans répétition

Un point de progression permet d'identifier l'usage d'un symbole terminal dans un contexte donné, qui correspond à certains schémas identifiés au sein de la trace de référence. Un point de progression peut ainsi représenter plusieurs occurrences d'un symbole au sein d'une trace. Par exemple, dans la figure 4.9, nous montrons toutes les occurrences du symbole b qui sont précédées du symbole a au sein de la trace « $abcbcdbcabc$ ».

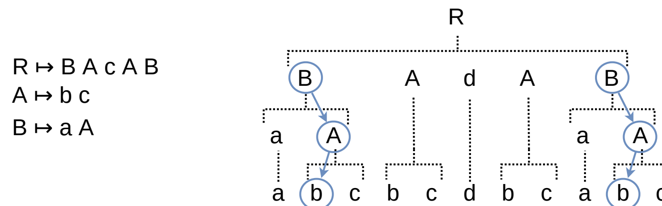


FIGURE 4.9 – Exemple d'un point de progression représentant les deux occurrences du symbole b dans la séquence ab au sein de la trace « $abcbcdbcabc$ ».

Nous observons également qu'un point de progression unique n'est pas toujours suffisant pour représenter toutes les occurrences d'une même séquence au sein d'une grammaire. Par exemple, dans la figure 4.10, la trace « $bcbdabcabd$ » contient deux occurrences de la séquence « ab ». Malgré cela, cette séquence n'est pas représentée par une règle grammaticale. Il en résulte que deux points de progressions sont nécessaires pour désigner les deux occurrences de cette séquence dans la trace.

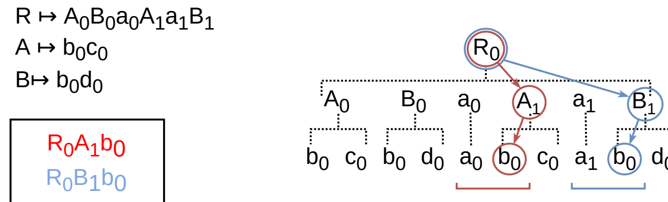


FIGURE 4.10 – Deux points de progression sont nécessaires pour lister toutes les apparitions de la séquence « ab » dans la trace « $bcbdabcabd$ ».

Représenter toutes les occurrences d'une séquence dans le cas des grammaires avec répétitions

Afin de suivre l'avancement du programme dans le cas des grammaires avec répétitions, nous procédons comme décrit dans la section 4.1.2. Chaque élément d'un point de progression se voit adjoint un nombre de répétitions indiquant combien de fois la séquence représentée a été parcourue depuis le début de la séquence actuellement cherchée dans la grammaire. La figure 4.11 illustre l'utilisation de nombres de répétitions afin de désigner une occurrence particulière d'une répétition.

Nous avons vu dans la section 4.2.3 qu'un point de progression peut permettre de désigner plusieurs événements ou séquences au sein d'une trace réduite sans répétition. Cela est d'autant plus vrai dans le cas où la grammaire représentant la trace contient des répétitions. La figure 4.12 représente l'exemple d'une grammaire représentant la séquence « $abcbcbcbcbc$ ». Au sein de cette trace, le point de progression « $A_0^2 b_0$ » peut désigner trois occurrences différentes de la séquence « $cbcb$ », au début, au milieu et à la fin de la boucle A_0^5

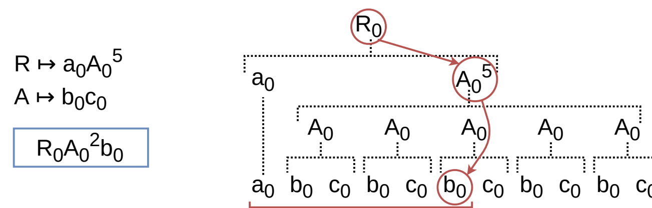


FIGURE 4.11 – Recherche de la séquence « $abcbcb$ » dans la trace « $abcbcbcbcbc$ ».

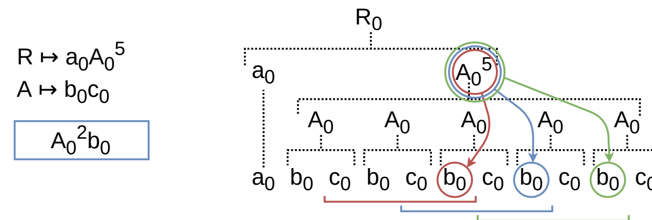


FIGURE 4.12 – Recherche de la séquence « $cbcb$ ».

4.2.4 Discussion

Nous avons montré comment PYTHIA-PREDICT reconstruit sa connaissance de l'état d'avancement du programme par recherche de similarités entre la trace de référence et la séquence d'événements courante. Si l'exécution courante fait intervenir des symboles terminaux qui n'apparaissent pas dans la grammaire, PYTHIA-PREDICT ne peut pas établir une correspondance entre les exécutions.

Par ailleurs, PYTHIA-PREDICT peut utiliser plusieurs points de progressions pour reconstruire sa connaissance de l'état d'avancement du programme. L'apparition de ce type de situation est d'autant plus probable et leur nombre d'autant plus grand que le nombre de règles grammaticales au sein de la grammaire est important. Ce type de grammaire est caractéristique des applications dont l'exécution est irrégulière. À l'inverse, cette méthode est particulièrement efficace dans le cas des programmes dont l'exécution est très structurées et qui utilisent une large variété de symboles terminaux permettant de décrire finement les différentes phases de l'exécution du programme. Dans ce cas, PYTHIA-PREDICT utilise moins de points de progression, divisant d'autant le coût de la mise à jour de ceux-ci.

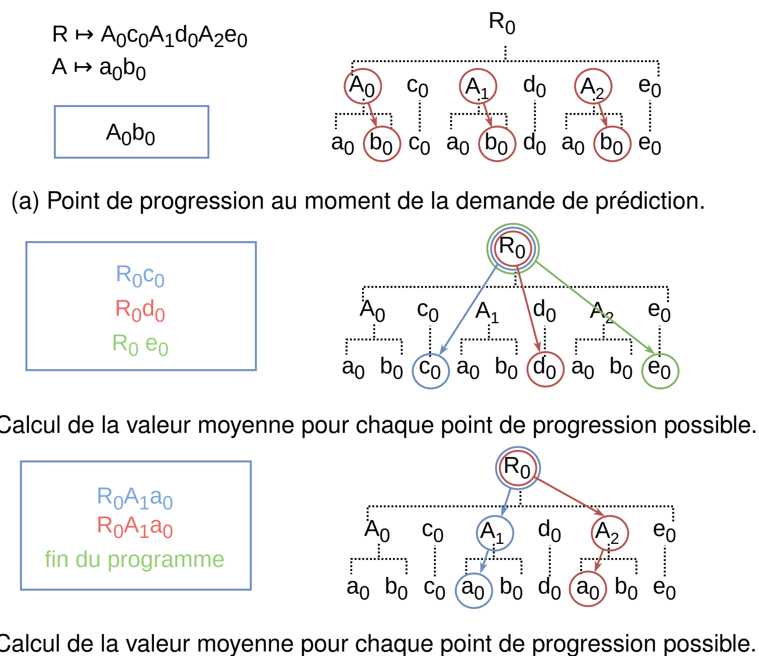
4.3 Prévoir le comportement futur du programme

Lors de l'exécution d'un programme, un *runtime* peut demander à PYTHIA-PREDICT une prévision sur les prochains événements susceptibles de se produire à l'avenir. Pour cela, PYTHIA-PREDICT explore la grammaire dite de référence à partir d'un ou plusieurs points de progression. Dans cette section, nous présentons comment PYTHIA-PREDICT explore l'avenir des programmes pour produire des prédictions (section 4.3.1) et les probabiliser (section 4.3.2), puis comment ces prédictions peuvent être enrichies par le *runtime* afin de prédire des informations plus complexes que la seule succession d'événements à venir (section 4.3.3).

4.3.1 Prévision des événements à venir

PYTHIA-PREDICT suit l'exécution d'un programme en mettant à jour une liste de points de progression à mesure que le *runtime* soumet des événements. En utilisant ces points de progressions, PYTHIA-PREDICT explore les

événements à venir dans la grammaire, comme illustré par la figure 4.13. Par soucis de lisibilité et puisque la grammaire ne présente pas de répétition, nous faisons ici encore l'économie des nombres de répétitions. L'état d'avancement du programme au moment où le *runtime* demande une prédiction est représenté par un unique point de progression : « $A_0 b_0$ » (figure 4.13a). Pour connaître les événements susceptibles de survenir à l'étape suivante, PYTHIA-PREDICT cherche les successeurs au point de progression représentant l'avancement actuel du programme. Trois points de progression sont possibles pour représenter le prochain événement, correspondant respectivement aux symboles c, d et e (figure 4.13b). PYTHIA-PREDICT est ainsi en mesure de prévoir que le prochain événement soumis par le *runtime* doit correspondre à l'un de ces trois symboles. Afin de prévoir la suite de l'exécution, PYTHIA-PREDICT répète la même opération avec chacun des points de progression (figure 4.13c). PYTHIA-PREDICT est ainsi en mesure de prédire que, si le prochain événement correspond au symbole c ou au symbole d, le suivant doit correspondre au symbole a, et si le prochain événement correspond au symbole e, celui-ci doit être le dernier rapporté par le *runtime* avant la fin de l'exécution du programme.



(a) Point de progression au moment de la demande de prédiction.

(b) Calcul de la valeur moyenne pour chaque point de progression possible.

(c) Calcul de la valeur moyenne pour chaque point de progression possible.

FIGURE 4.13 – Association contextualisée de valeurs aux points de progression.

Afin de faciliter cette exploration, PYTHIA-PREDICT explore ses prédictions comme un arbre des possibles, chaque point de progression correspondant à un futur possible. PYTHIA-PREDICT conserve en interne une liste de ses points de progressions. Le *runtime* peut ainsi parcourir ces futurs alternatifs et demander à PYTHIA-PREDICT d'en sélectionner un afin de connaître les événements qui doivent suivre. À titre d'exemple, dans la figure 4.14, nous construisons l'arbre des possibles correspondant à l'exemple de la figure 4.13. Le point de progression représentant l'état d'avancement du programme au moment de la demande de prédiction est figuré dans une boîte sur fond gris. Les trois points de progression correspondant à la prédiction à un événement d'avance sont représentés de haut en bas respectivement sur fond bleu, rouge et vert. Chacun peut être exploré indépendamment des autres.

4.3.2 Calcul de la probabilité de chaque événement

La méthode vue dans la section 4.3.1 permet à PYTHIA-PREDICT de parcourir la grammaire pour suivre l'exécution d'un programme et pour prévoir les événements à venir. Cependant, un *runtime* peut avoir besoin de probabiliser les différents futurs proposés par PYTHIA-PREDICT. Pour cela, PYTHIA-PREDICT procède par dénombrement :

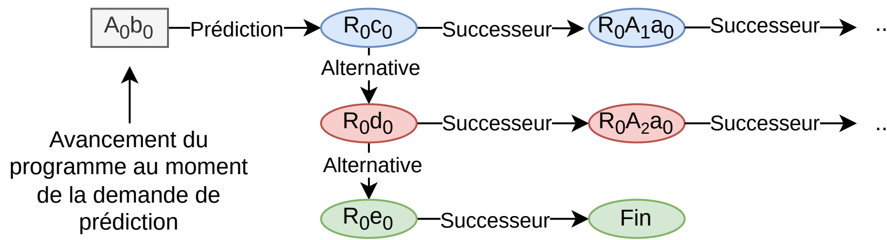


FIGURE 4.14 – Arbre des possibles correspondant à l'exemple de la figure 4.13.

en comptant le nombre d'événements représentés par chaque point de progression, PYTHIA-PREDICT peut estimer la probabilité des événements représentés.

Ce nombre d'événements peut être calculé à partir du premier élément d'un point de progression et de son nombre de répétitions. Dans la figure 4.15, nous montrons trois exemples. Dans ces exemples , la trace représentée est trop longue pour que les boucles soient « dépliées ». La grammaire de l'exemple consiste en quatre règles grammaticales avec des boucles imbriquées.

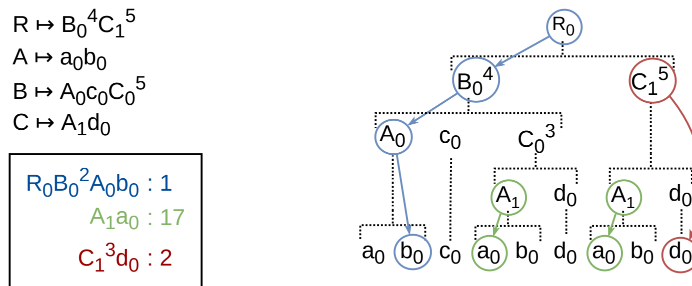


FIGURE 4.15 – Calcul du nombre d'occurrences de points de progression.

Le premier exemple est le point de progression « $R_0B_0^2A_0b_0$ », représenté en bleu sur la figure. Ce point de progression relie un symbole terminal à la racine de la grammaire. Il identifie donc un événement unique au sein de la trace. Le nombre de symboles désignés par le point de progression est donc 1.

Le deuxième exemple est le point de progression « A_1a_0 », représenté en vert sur la figure. Le premier élément du point de progression, A_1, apparaît au sein de la séquence associée au symbole C. Connaître le nombre d'apparitions de la séquence représentée par C permet donc de connaître le nombre d'événements représentés par le point de progression. Le non-terminal C apparaît deux fois dans la grammaire : C_0 se répète dans une boucle à 5 itérations. C_1 apparaît 3 fois dans la séquence B et B_0 est répété 4 fois. Le nombre d'événements représentés par le point de progression « A_1a_0 » est donc de $5+3*4$, soit 17 événements.

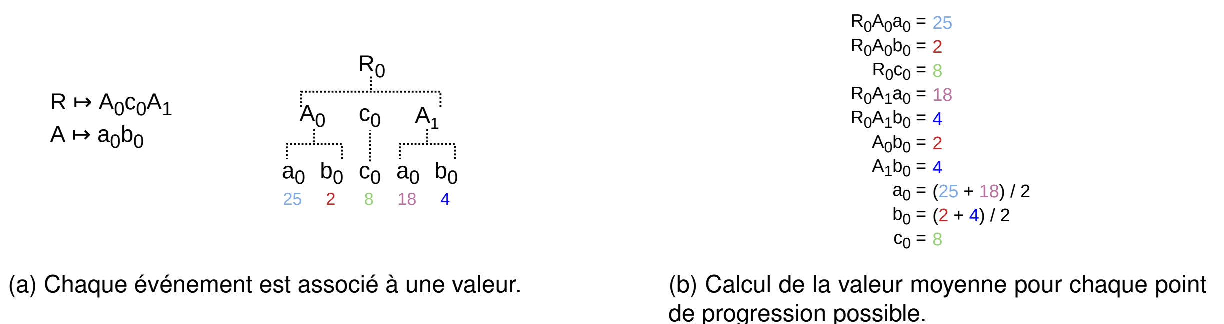
Enfin, le troisième exemple est le point de progression « $C_1^3d_0$ ». C_0 apparaît comme une boucle de cinq itérations au sein de la règle racine. Cependant, le premier élément du point de progression, C_1^3, indique que la séquence associée à C est actuellement dans sa quatrième exécution consécutive. Le point de progression ne peut donc représenter que la dernière ou l'avant dernière itération de la boucle C_1^5. Le nombre d'événements représentés par le point de progression « $C_1^3d_0$ » est donc 2.

Pour une prédiction donnée, en calculant le nombre d'événements représentés par chaque point de progression et en le divisant par leur somme, PYTHIA-PREDICT calcule la probabilité pour chaque événement représenté.

4.3.3 Enrichissement des prévisions

En plus de connaître les séquences d'événements à venir avec leur probabilité, certains *runtimes* peuvent vouloir prédire d'autres informations complémentaires telles que la durée d'exécution séparant deux événements ou la valeur d'une variable au moment où surviendra un événement. De telles prédictions sont spécifiques à l'utilisation que chaque *runtime* peut avoir de PYTHIA, et ne peuvent donc pas être directement intégrées au sein de l'oracle, qui a vocation à rester générique. De telles prédictions peuvent cependant être implémentées avec PYTHIA-PREDICT grâce à l'utilisation des points de progressions. Quand un *runtime* demande à PYTHIA-PREDICT une prédiction, PYTHIA-PREDICT n'indique pas seulement un événement à venir, mais fournit les points de progression décrivant dans quel contexte cet événement doit avoir lieu. L'API de PYTHIA-PREDICT est décrite à l'annexe A.

La figure 4.16 montre par un exemple comment utiliser les points de progression pour prédire la durée séparant un événement du précédent. La grammaire considérée représente la trace « $abcab$ » et chacun des cinq événements de cette trace est associé à la durée en microsecondes depuis le dernier événement ou, dans le cas du premier événement, depuis le début de l'exécution (figure 4.16a). Le parcours de cette grammaire par l'algorithme décrit dans la section 4.1.3 met en œuvre cinq points de progression : « $R_0A_0a_0$ », « $R_0A_0b_0$ », « R_0c_0 », « $R_0A_1a_0$ » et « $R_0A_1b_0$ ». En étudiant attentivement l'algorithme présenté dans la section 4.3, nous voyons que seulement cinq autres points de progression peuvent être manipulés par PYTHIA-PREDICT. Trois de ces points de progression permettent de représenter respectivement les séquences « a », « b » et « c » : « a_0 », « b_0 » et « c_0 ». Les deux autres, « A_0b_0 » et « A_1b_0 », permettent de représenter respectivement la séquence « ab » au début et à la fin de la trace. La valeur moyenne des durées associées à chaque événement représenté par chacun de ces points de progression correspond à la durée moyenne écoulée entre ces événements et leurs prédécesseurs respectifs. Par exemple, nous voyons que la durée séparant la première apparition de a et la première apparition de b est de 2ms. La durée séparant la seconde apparition de a de la seconde apparition de b est de 4ms. Donc, la durée moyenne écoulée entre un événement a et un événement b est associée au point de progression « b_0 », et est de 3 millisecondes (figure 4.16b). Cette méthode permet à un *runtime* d'enregistrer des valeurs pour chaque événement lorsque cela est pertinent ainsi que de pré-calculer une table similaire. En rechargeant cette table en même temps que la grammaire de référence, le *runtime* peut ré-associer une valeur contextuelle aux prévisions de PYTHIA-PREDICT.



(a) Chaque événement est associé à une valeur.

(b) Calcul de la valeur moyenne pour chaque point de progression possible.

FIGURE 4.16 – Association contextualisée de valeurs aux points de progression.

Puisque la structure finale de la grammaire n'est pas connue pendant l'exécution de référence, les valeurs associées à chaque événement doivent être stockées dans une trace séquentielle. Après l'exécution de référence de l'application, lorsque la grammaire représentant la trace complète est connue, les valeurs collectées peuvent être attachées aux symboles de la grammaire.

4.4 Conclusion

Dans ce chapitre, nous avons présenté le fonctionnement de PYTHIA-PREDICT qui permet, à partir de la trace d'une exécution de référence fournie par l'utilisateur, de suivre l'avancement d'une application, ainsi que d'en prévoir le comportement futur. Pour cela, PYTHIA-PREDICT charge en mémoire la grammaire de la trace de référence et recueille les événements soumis par le *runtime* lors de la nouvelle exécution de l'application. Ces événements sont comparés à ceux de la grammaire de référence et PYTHIA-PREDICT établit des points de progression indiquant l'état d'avancement du programme.

Si l'exécution courante diffère de l'exécution de référence, PYTHIA-PREDICT calcule les similarités qui peuvent correspondre par exemple à différents appels d'une même fonction à différents moments de l'exécution. Cette recherche de similarités est effectuée par raffinage d'une liste de points de progression au fur et à mesure des événements.

En utilisant les points de progression comme représentation de l'avancement de l'exécution du programme, PYTHIA-PREDICT est capable de parcourir les événements susceptibles de survenir et de les probabiliser. Les points de progression ne permettent pas seulement d'explorer l'exécution à venir du programme. Ils permettent aussi de décrire dans quel contexte un événement futur surviendra, ce qui permet à un *runtime* de précalculer des données : par exemple la durée d'exécution séparant un événement de son prédécesseur ou la valeur d'une variable en fonction du contexte. Ce procédé permet au *runtime* d'utiliser PYTHIA-PREDICT pour établir des prédictions correspondant à ses besoins particuliers. PYTHIA est donc un oracle générique.

Nous avons ainsi montré comment utiliser une grammaire représentant une exécution d'un programme pour construire un oracle capable de produire des prédictions contextualisées adaptées aux besoins d'un *runtime*, ce qui constitue notre deuxième contribution.

Chapitre 5

Évaluation de l'utilisation de PYTHIA au sein d'un runtime

Sommaire

5.1 Environnement expérimental . . . 57
5.1.1 Machines . . . 57
5.1.2 Applications et jeux de données . . . 57
5.1.3 Supports d'exécution . . . 58
5.2 Évaluation du surcoût de PYTHIA-RECORD . . . 59
5.3 Évaluation des performances de PYTHIA-PREDICT . . . 60
5.3.1 Correction des prédictions de PYTHIA-PREDICT . . . 61
5.3.2 Coût des prévisions de PYTHIA-PREDICT . . . 62
5.4 Évaluation de l'utilisabilité de PYTHIA . . . 63
5.4.1 Modifications apportées à GNU OpenMP . . . 63
5.4.2 Modifications apportées à Lulesh . . . 64
5.4.3 Impact de Pythia-OpenMP sur la durée d'exécution de Lulesh . . . 65
5.5 Conclusion . . . 68

Dans les chapitres précédents, nous avons présenté comment PYTHIA-RECORD produit, sous la forme d'une grammaire, une représentation structurée de la trace d'exécution d'un programme. Nous avons ensuite présenté PYTHIA-PREDICT et détaillé comment utiliser la trace d'exécution d'un programme pour prédire son comportement futur. Nous avons aussi montré comment un *runtime* peut enrichir les prédictions produites par PYTHIA, oracle générique, afin de répondre à des besoins spécifiques pour une optimisation particulière. Dans ce chapitre, nous évaluons la capacité de PYTHIA à capturer le comportement de l'exécution d'un programme, à prédire son comportement futur, et à être utilisé au sein d'un *runtime* pour mettre en œuvre une optimisation.

Dans la section 5.1, nous présentons l'environnement expérimental et les applications utilisées pour nos expériences. Dans les sections 5.2 et 5.3, nous évaluons respectivement le surcoût de PYTHIA-RECORD et les performances de PYTHIA-PREDICT. Ensuite, dans la section 5.4, nous évaluons l'utilisation de PYTHIA au sein d'un *runtime* par l'implémentation d'une stratégie de parallélisme dynamique pour OpenMP. Nous concluons le chapitre dans la section 5.5.

5.1 Environnement expérimental

Nous présentons dans un premier temps les machines utilisées (section 5.1.1) puis nous listons les applications et les jeux de données utilisés (section 5.1.2).

5.1.1 Machines

Nous présentons ici les trois machines parallèles que nous avons utilisées dans le cadre de nos expérimentations. Deux de ces machines, Pudding et Pixel, sont des machines disponibles au sein du département Informatique de Télécom SudParis où cette thèse s'est déroulée. La troisième machine, Paravance, est un *cluster* de machines de Grid'5000 [12].

Pudding. Pudding est une machine NUMA équipée de deux processeurs Intel Xeon Silver 4116, disposant chacun de 12 cœurs physiques *hyperthreadés* (24 cœurs au total, 2 *threads* par cœur). Les processeurs sont cadencés à 2,1GHz. La machine dispose de 64Go de DRAM et fonctionne avec le noyau Linux 5.4.0.

Pixel. Pixel est une machine NUMA équipée de deux processeurs Intel Xeon E5-2630 v3 cadencés à 2,4GHz. La machine dispose au total de 16 cœurs physiques *hyperthreadés* (deux *threads* par cœur) et de 32 Go de DRAM. La machine fonctionne avec le noyau Linux 5.4.0.

Paravance. Paravance est un cluster de 72 machines faisant partie de l'ensemble Grid'5000. Chaque nœud du cluster est une machine NUMA équipée de deux processeurs Intel Xeon E5-2630 v3 cadencés à 2,4GHz. Chaque nœud dispose de 16 cœurs physiques au total (deux *threads* par cœur) et de 128Go de RAM. Les nœuds sont reliés entre eux par un réseau Ethernet avec un débit de 10Gbps et fonctionnent avec le noyau Linux 5.10.0.

5.1.2 Applications et jeux de données

Nous avons mené nos expérimentations sur treize applications différentes, et pour chacune, nous avons utilisé trois tailles de problème différentes désignées par leur nom : *small*, *medium* et *large*. Nous listons ici les applications testées et les paramètres utilisés pour chaque taille de problème.

NAS Parallel Benchmarks

Les *NAS Parallel Benchmarks* [8] (NPB) sont un ensemble de programmes développés par la division *Advanced supercomputers* de la NASA. Leur objectif est de permettre d'évaluer les performances des supercalculateurs. Plusieurs implémentations des *NAS Parallel Benchmarks* utilisant des modèles de programmation différents sont disponibles. Dans nos expérimentations, nous utilisons les implémentations MPI des programmes BT, CG, EP, FT, IS, LU, MG et SP dans leur version 3.3.1. Les jeux de données *small*, *medium*, et *large* correspondent aux tailles de problèmes A, B, et C.

AMG, Kripke, Lulesh et Quicksilver

AMG [72], Kripke [45], Lulesh [39] et Quicksilver [61] sont quatre programmes développés au *Lawrence Livermore National Laboratory*. Les quatre applications utilisent à la fois MPI et OpenMP dans leur implémentation.

AMG est un solveur algébrique multigrille parallèle. Les trois jeux de données utilisés correspondent aux paramètres de l'application suivants : `-n 100 100 100` pour *small*, `-n 150 150 150` pour *medium*, `-n 200 200 200` pour *large*.

Lulesh est une application d'hydrodynamique qui résout un problème de souffle de Sedov. La version de Lulesh utilisée est une version légèrement modifiée de la version 2.0. Les modifications apportées sont détaillées dans la section 5.4.2. Les trois tailles de problème utilisées correspondent aux paramètres suivants : `-s 10` pour *small*, `-s 30` pour *medium* et `-s 50` pour *large*.

Quicksilver est une application qui résoud un problème de transport de particules par une méthode de Monté-Carlo. Cette application possède un comportement non déterministe. Les trois tailles de problème utilisées correspondent aux paramètres suivants : `--lx 500 --ly 500 --lz 500 -n 1000000` pour *small*, `--lx 500 --ly 500 --lz 500 -n 10000000` pour *medium* et `--lx 500 --ly 500 --lz 500 -n 20000000` pour *large*.

Kripke est une application qui résout un problème de transport de particules. Contrairement à Quicksilver, Kripke utilise une méthode de résolution déterministe. Les trois tailles de problème utilisées correspondent aux paramètres suivants : `--procs 2,2,2 --groups 128` pour *small*, `--procs 2,2,2 --groups 512` pour *medium* et `--procs 2,2,2 --groups 1024` pour *large*.

miniFE

MiniFE [35] est une application *proxy* développée par le *National Energy Research Scientific Computing Center* (NERSC) qui résout un problème d'éléments finis implicites non structurés. Les trois tailles de problème utilisées correspondent aux paramètres suivants : `-nx 100 -ny 100 -nz 100` pour *small*, `-nx 200 -ny 200 -nz 200` pour *medium*, et `-nx 300 -ny 300 -nz 300` pour *large*.

5.1.3 Supports d'exécution

Dans le cadre de nos expérimentations, nous avons développé deux *runtimes* afin d'évaluer les performances et l'utilisabilité de PYTHIA dans des cas proches d'un usage réel. Dans cette section, nous présentons ces deux *runtimes* que nous appelons Pythia-MPI+OpenMP et Pythia-OpenMP.

Pythia-MPI+OpenMP

Pythia-MPI+OpenMP est un *runtime* qui imite le comportement des *runtimes* MPI et OpenMP. En pratique, Pythia-MPI+OpenMP intercepte les appels du programme aux fonctions de ces deux *runtimes* à l'aide de la variable d'environnement `LD_PRELOAD` (Cf. section 3.1.5). Développer un *runtime* complet implémentant les interfaces MPI et OpenMP est au-delà du périmètre de cette thèse. Donc, Pythia-MPI+OpenMP repose sur des implémentations existantes de ces deux *runtimes* pour mimer leur comportement.

Pythia-MPI+OpenMP utilise PYTHIA pour capturer l'utilisation de MPI et de OpenMP par le programme. Chaque appel à l'un de ces *runtimes* est associé à un symbole terminal différent en fonction des arguments utilisés et est enregistré par PYTHIA-RECORD. Le détail des arguments considérés pour choisir si deux appels à MPI et OpenMP doivent être associés à un même symbole terminal se trouve à l'annexe B.

Pythia-MPI+OpenMP simule l'utilisation de PYTHIA que pourrait avoir un programmeur ou une programmeuse pour implémenter une optimisation au sein de MPI ou d'OpenMP. À chaque fois que l'application appelle une fonction de MPI ou d'OpenMP susceptible d'introduire une attente dans l'exécution, par exemple `MPI_Wait` ou une barrière OpenMP, Pythia-MPI+OpenMP demande à PYTHIA quels événements sont les plus susceptibles de survenir. Le nombre d'événements futurs demandés à PYTHIA est configurable afin de pouvoir mesurer le coût et la

correction des prévisions de PYTHIA en fonction de l'avance des prévisions. Pythia-MPI+OpenMP simule ainsi la volonté des concepteurs d'exploiter ces temps d'attente pour optimiser l'utilisation du processeur. L'annexe B indique pour chaque fonction instrumentée par Pythia-MPI+OpenMP lesquelles déclenchent une prévision.

Pythia-OpenMP

Le *runtime* Pythia-OpenMP a été développé afin d'évaluer les performances et l'utilisabilité de PYTHIA au sein d'un *runtime* avec l'implémentation d'une optimisation concrète. Pythia-OpenMP est un *runtime* OpenMP implémentant une stratégie de parallélisme dynamique. L'objectif est d'améliorer la vitesse des petites régions parallèles des programmes en évitant le coût de synchronisation inutile entre de nombreux *threads* si la quantité de travail à distribuer ne le justifie pas. Pour cela, nous avons modifié l'implémentation GNU du *runtime* OpenMP version 12. Alors que l'implémentation originale GNU OpenMP utilise un nombre de *threads* constant (par défaut, autant de *threads* que de cœurs disponibles), et ce quelque soit la région parallèle, Pythia-OpenMP peut changer ce nombre de *threads* en fonction de la durée prévue de la région parallèle. Ainsi, une région parallèle très courte est exécutée avec un faible nombre de *threads*, ce qui limite le surcoût de la synchronisation inter-*threads*, alors qu'une région parallèle plus longue est exécutée par un plus grand nombre de *threads* pour augmenter le parallélisme.

Pour cela, Pythia-OpenMP utilise PYTHIA pour enregistrer le début et la fin de chaque région parallèle. En plus de ces événements, Pythia-OpenMP enregistre la date à laquelle chacun de ces événements survient. À la fin de l'exécution de référence, Pythia-OpenMP re-parcourt la trace d'exécution et pré-calcule la durée moyenne de chaque région parallèle en fonction du contexte où elle s'est exécutée (Cf. section 4.3.3). Ces durées sont utilisées lors des exécutions suivantes pour estimer un ordre de grandeur de la durée des régions parallèles.

Si une trace de référence est disponible, Pythia-OpenMP interroge PYTHIA au début de chaque région parallèle pour obtenir une prévision sur la fin de l'exécution de la région parallèle. Pythia-OpenMP utilise le contexte de la prévision contenant la durée probable de la région parallèle à venir et choisit le nombre de *threads* à utiliser. Pythia-OpenMP utilise des valeurs seuils : par exemple, si la durée prévue est inférieure à $2\mu s$, le *runtime* n'exécute qu'un *thread* ; si la durée prévue est inférieure à $8\mu s$, il exécute 4 *threads* ; sinon, le nombre maximum de *threads* est utilisé.

5.2 Évaluation du surcoût de PYTHIA-RECORD

Afin d'évaluer le surcoût de PYTHIA-RECORD, nous exécutons les différentes applications présentées dans la section 5.1.2 en utilisant le *runtime* Pythia-MPI+OpenMP présenté dans la section 5.1.3. Dans la mesure où notre objectif est de mettre PYTHIA-RECORD à l'épreuve, chaque application est exécutée avec la taille de problème *large*. Toutes les applications sont exécutées en utilisant quatre nœuds du cluster Paravance. Les applications issues des NAS Parallel Benchmarks n'utilisent pas OpenMP et sont donc exécutées avec 64 rangs MPI (16 processus par machine, soit un par cœur physique de processeur). Les autres applications (AMG, Lulesh, Kripke, MiniFE et Quicksilver) utilisent à la fois MPI et OpenMP, et sont exécutées en utilisant huit rangs MPI (deux rangs par machine) et huit *threads* OpenMP par rang (soit un *thread* par cœur physique).

La table 5.1 liste pour chaque programme la durée moyenne sur dix exécutions dans deux configurations différentes : sans PYTHIA-RECORD (Vanilla) d'une part, et avec PYTHIA-RECORD utilisé pour enregistrer tous les appels aux *runtimes* MPI et OpenMP comme décrit dans la section 5.1.3 d'autre part. La table indique également le nombre total d'événements enregistrés et le nombre moyen de règles grammaticales utilisées par PYTHIA-RECORD pour représenter la trace d'exécution correspondant à chacun des *threads* de l'application.

Le surcoût en durée d'exécution varie de -5,8% pour MiniFE à +4,9% pour Quicksilver. Pour la majorité des applications, le surcoût varie entre -1,1% et +1,4%. Par conséquent, les résultats montrent que PYTHIA-RECORD

Application	Vanilla (s)	PYTHIA-RECORD (s)	surcoût (%)	# événements	# règles
BT	24.2	24.2	0.7	2,329,920	3
CG	9.9	9.9	-0.3	3,837,890	15
EP	4.2	4.1	-3.8	384	1
FT	17.4	17.4	0.2	3,072	2
IS	3.2	3.2	0.1	2,493	2
LU	23.0	23.3	1.4	18,164,200	11
MG	4.2	4.1	-0.5	609,888	14
SP	24.3	24.4	0.2	356,870	9
AMG	38.7	38.4	-0.9	118,438	150
Lulesh	125.6	124.2	-1.1	28,150,300	12
Kripke	59.8	61.0	2.0	9,881	46
miniFE	25.8	24.3	-5.8	39,272	8
Quicksilver	35.9	37.6	4.9	26,786,800	409

TABLE 5.1 – Mesure des performances de PYTHIA-RECORD.

n'a pas d'impact significatif sur les performances de la plupart des programmes.

Certaines applications ont des exécutions irrégulières, ce qui résulte en un plus grand nombre de règles grammaticales. Par exemple, Quicksilver utilise MPI pour « transférer » une particule du domaine correspondant à un rang MPI à un autre. Les échanges de messages MPI sont alors irréguliers car ils dépendent de la position initiale des particules dans les données du problème. Ces communications MPI irrégulières engendrent un grand nombre de règles grammaticales, et les modifications fréquentes de la grammaire entraînent un surcoût non négligeable de 4,9% sur la durée d'exécution.

Le nombre d'événements enregistrés par chaque programme varie significativement. Certains programmes comme EP ou FT ne font que quelques appels à MPI au cours de leur exécution tandis que d'autres font un usage intensif de MPI, par exemple LU, ou de MPI et OpenMP, par exemple Lulesh et Quicksilver, avec plusieurs millions d'appels enregistrés. Nous remarquons également que même lorsque de nombreux événements sont enregistrés, les grammaires générées ne contiennent pour la plupart qu'une quinzaine de règles grammaticales.

Si nous observons les grammaires générées, nous remarquons que leur structure reproduit le programme instrumenté. Cette observation confirme le choix des contraintes appliquées au grammaire. En particulier, les boucles des programmes sont correctement représentées par des répétitions. Ces répétitions permettent de représenter plusieurs milliers d'événements en quelques symboles. À titre d'exemple, la figure 5.1 montre la grammaire générée pour le programme BT et pour un rang MPI. La grammaire consiste essentiellement en 200 itérations d'une boucle comprenant une poignée d'appels au *runtime* MPI.

$$
\begin{aligned}
R &\mapsto \text{Bcast}^6\ B\ \text{Barrier}\ A^{200}\ \text{Allreduce}\ B\ \text{Reduce Barrier} \\
A &\mapsto B\ \text{Isend Irecv [...] Wait}^2 \\
B &\mapsto \text{Irecv [...] WaitAll}
\end{aligned}
$$

FIGURE 5.1 – Grammaire générée à partir des appels à MPI observés au sein d'un rang MPI de l'application BT avec la taille de problème *large*. (Le préfix `MPI_` des fonctions a été omis à des fins de concision.)

5.3 Évaluation des performances de PYTHIA-PREDICT

Afin d'évaluer les performances de PYTHIA-PREDICT, nous réalisons différentes expérimentations sur les différentes applications présentées dans la section 5.1.2. Les applications sont exécutées en utilisant le *runtime* Pythia-MPI+OpenMP présenté dans la section 5.1.3 sur la machine Paravance présentée dans la section 5.1.1. Toutes les

mesures présentées ici sont des moyennes réalisées en répétant dix fois chaque expérience.

Nous présentons ici les résultats de nos mesures de la correction des prédictions de PYTHIA-PREDICT (section 5.3.2) et de leur coût (section 5.3.1).

5.3.1 Correction des prédictions de PYTHIA-PREDICT

Dans cette section, nous évaluons la correction des prédictions de PYTHIA-PREDICT en fonction du nombre d'événements en avance à prédire. Comme dans l'expérience précédente, chaque programme présenté dans la section 5.1.2 est exécuté avec le *runtime* Pythia-MPI+OpenMP. De même, les NAS Parallel Benchmarks sont exécutés avec 64 rangs MPI tandis que les autes programmes sont exécutés avec huit rangs MPI et huit *threads* OpenMP par rang. La taille de problème utilisée pour l'exécution de référence de chaque programme est la plus petite disponible (*small*), donc avec un jeu de données restreint. Cette exécution de référence est ensuite fournie à PYTHIA-PREDICT lorsque l'application est ré-exécutée avec les tailles de problème *small*, *medium*, ou *large*. Ainsi, nous évaluons la capacité de PYTHIA à prévoir le comportement d'une application dans le cas où l'exécution de référence correspond à un jeu de données restreint.

Comme vu dans la section 5.1.3, Pythia-MPI+OpenMP enregistre tous les appels aux *runtimes* MPI et OpenMP. À chaque fois qu'une fonction bloquante est atteinte, Pythia-MPI+OpenMP demande à PYTHIA-PREDICT de prédire dans l'ordre les n prochains événements les plus susceptibles de survenir. Pythia-MPI+OpenMP vérifie ensuite quels sont les événements soumis à PYTHIA par la suite. Une prévision est considérée correcte si le $n^{\text{ème}}$ événement qui suit la prédiction est bien l'événement prédit.

Nous rapportons dans la figure 5.2 les mesures effectuées. La correction de PYTHIA-PREDICT décroît lentement lorsque n augmente, mais reste élevée pour la plupart des applications. Sur huit des treize applications testées, les prédictions de PYTHIA-PREDICT sont correctes à plus de 90% jusqu'à 128 événements en avance et quel que soit le jeu de données utilisé. Lorsque le jeu de données *small* est utilisé, c'est-à-dire lorsque le jeu de données utilisé par l'application est le même que celui utilisé pendant l'exécution de référence, le taux de prédictions valides avoisine les 95% pour 11 des 13 applications et jusqu'à 64 événements en avance.

Comme nous l'avons vu dans la section 5.2, AMG et Quicksilver exhibent des comportements irréguliers. Malgré cela, les prédictions de PYTHIA-PREDICT à 16 événements en avance restent correctes à 78,9% pour AMG et à 78,6% pour Quicksilver avec le jeu de données *large*. Cela s'explique par le fait que, malgré le caractère dans l'ensemble irrégulier de ces applications, certaines parties de leurs exécutions restent régulières.

Cependant, plusieurs applications montrent une dégradation de la correction des prédictions de PYTHIA-PREDICT quand n croît. En particulier, les nombres d'itérations des boucles au sein des programmes LU et MG varient en fonction de la taille des jeux de données, ce qui perturbe les prédictions. En effet, PYTHIA-PREDICT basant ses prédictions sur une trace d'exécution enregistrée avec la taille de problème *small*, il fait très souvent des erreurs de prédiction sur la fin des boucles. Par exemple, si le taux de prédictions correctes pour le programme LU avec le jeu de données *small* reste supérieur ou égal à 98,9% jusqu'à 128 événements en avance, avec les jeu de données *medium* et *large*, il n'est que de 74,7% à deux événements en avance, et décroît jusqu'à 5,5% à 16 événements en avance.

Les résultats montre toutefois que le comportement de la plupart des applications testées varie peu d'une exécution à l'autre, même quand la taille du problème varie. PYTHIA-PREDICT est ainsi capable de prévoir le comportement futur de la plupart des applications avec justesse, y compris lorsque le nombre d'événements en avance n est élevé. De plus, pour 8 des 13 applications étudiées, changer la taille du jeu de données n'a pas d'incidence significative sur la qualité des prédictions, y compris pour plus d'une centaine d'événements en avance.

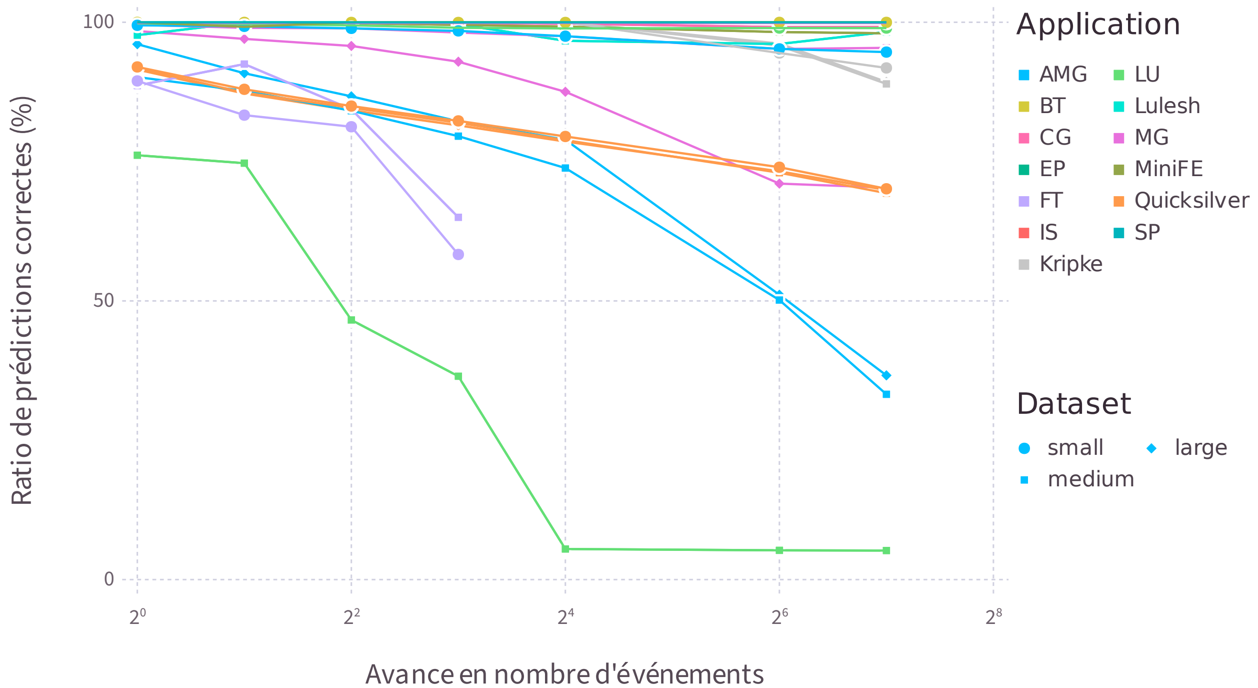


FIGURE 5.2 – Justesse de prédictions de PYTHIA sur Paravance.

5.3.2 Coût des prévisions de PYTHIA-PREDICT

Lorsqu'un *runtime* a recours à PYTHIA pour mettre en œuvre une optimisation, le coût de la prédiction fournie par PYTHIA-PREDICT doit rester inférieur au bénéfice de l'optimisation qu'elle permet. Dans cette section, nous évaluons le coût des prévisions produites par PYTHIA-PREDICT. Pour cela, nous exécutons chacune des applications avec le *runtime* Pythia-MPI+OpenMP et avec une taille de problème *large*. Pour chaque application, l'exécution de référence utilisée correspond à une taille de problème *small* afin de placer PYTHIA dans la situation la plus défavorable. Nous faisons varier l'avance de la prédiction, notée n pour n événements en avance, et nous mesurons par instrumentation le temps passé dans PYTHIA-PREDICT pour effectuer ces prédictions.

La figure 5.3 montre la durée moyenne nécessaire à PYTHIA-PREDICT pour établir une prédiction en fonction du nombre d'événements à prédire en avance. Pour toutes les applications, le coût des prédictions croît linéairement avec n. Pour les applications autres que Quicksilver, le coût des prédictions à quatre événements en avance est compris entre 1μs pour Lulesh et 7,34μs pour Kripke. Pour BT, CG, SP et Lulesh, la prédiction du prochain événement coûte moins de 500ns. Une prédiction à 128 événements en avance coûte entre 18μs pour Lulesh et 100μs pour Kripke.

Cependant, nous remarquons que le coût d'une prédiction varie d'une application à l'autre. En particulier, il est significativement plus élevé pour l'application Quicksilver, de 419μs pour prédire le prochain événement jusqu'à 825μs pour une prédiction à 128 événements en avance. Ceci s'explique par le fait que le nombre de règles de la grammaire utilisées pour établir les prévisions a un impact direct sur son coût. En effet, PYTHIA-PREDICT doit parcourir la grammaire à l'aide d'un ou plusieurs points de progression. Si la grammaire est « complexe », PYTHIA-PREDICT a potentiellement recours à plus de points de progression. C'est typiquement le cas de Quicksilver dont les grammaires comportent en moyenne 409 règles, contre 12 pour Lulesh et 46 pour Kripke (Cf. section 5.2).

Ces résultats montrent que les performances de PYTHIA permettent à un *runtime* de mettre en œuvre des

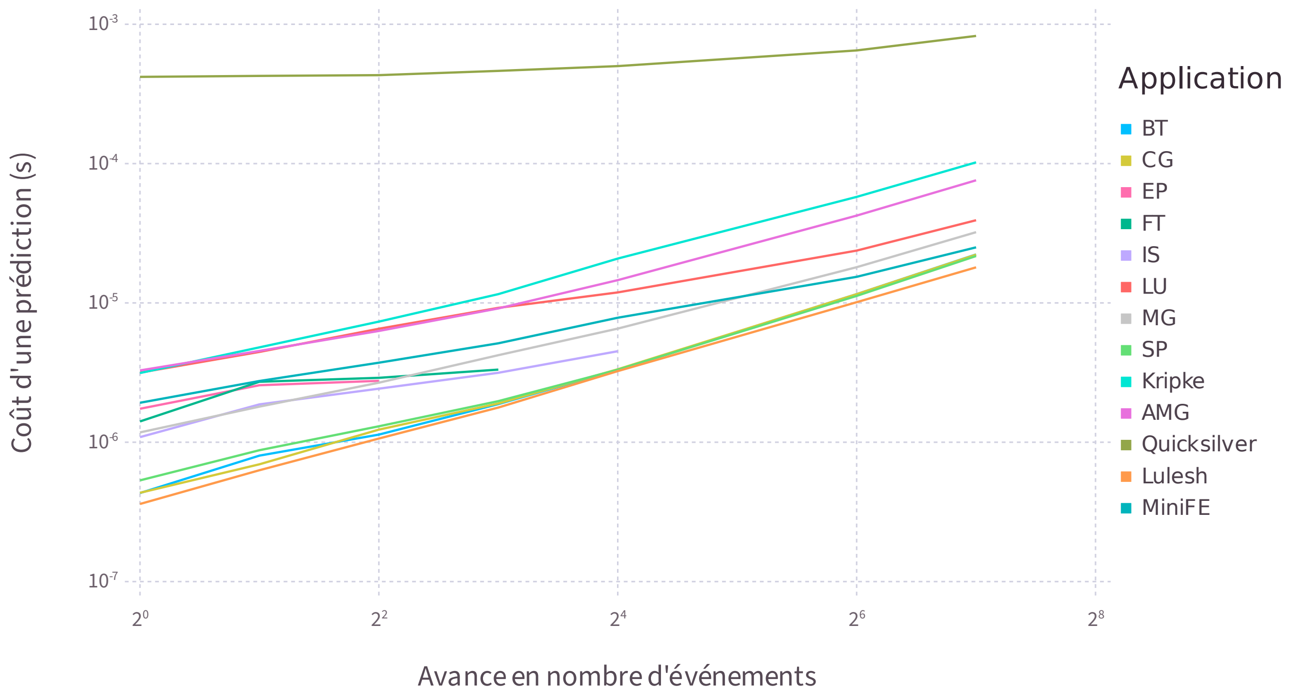


FIGURE 5.3 – Coût des prédictions de PYTHIA-PREDICT sur Paravance avec le jeu de données *large*.

optimisations fines, de l'ordre de quelques microsecondes en prédisant le prochain événement, ou de l'ordre de quelques dizaines de microsecondes avec des prédictions à une quinzaine d'événements en avance. Les prévisions à plus long terme sont plus coûteuses à cause du fait du parcours de la grammaire qu'elles nécessitent. Pour la plupart des applications, le coût d'une prédiction reste inférieur à 20 microsecondes pour 64 événements en avance et inférieur à 100 microsecondes pour 128 événements en avance. Ces résultats permettent à un *runtime* de mettre en œuvre des optimisations à gros grain comme le chargement anticipé de données.

5.4 Évaluation de l'utilisabilité de PYTHIA

Dans cette section, nous évaluons l'utilisabilité de PYTHIA au sein d'un *runtime*. Pour cela, nous utilisons le *runtime* Pythia-OpenMP présenté dans la section 5.1.3 pour exécuter l'application Lulesh. Nous commençons par présenter les modifications apportées à Lulesh ainsi qu'à l'implémentation GNU d'OpenMP pour obtenir Pythia-OpenMP (section 5.4.1 et 5.4.2). Ensuite, nous présentons les résultats obtenus (section 5.4.3).

5.4.1 Modifications apportées à GNU OpenMP

Pythia-OpenMP est une version modifiée du *runtime* GNU OpenMP mettant en œuvre une stratégie de modulation du nombre de *threads* à utiliser pour chaque région parallèle du programme. L'objectif de cette stratégie est d'optimiser l'exécution des petites régions parallèles. Sur des machines disposant d'un grand nombre de cœurs, l'implémentation GNU d'OpenMP utilise toujours le maximum de *threads* disponibles. Cette heuristique est optimale pour les régions capables de tirer parti des ressources de la machine. Cependant, dans le cas de petites régions parallèles, si le nombre de *threads* est trop grand, le temps passé à distribuer le travail entre eux et à les synchroniser

devient non négligeable, voire dépasse le temps effectivement passé à exécuter le code de la région parallèle. Pour ces petites régions parallèles, réduire le nombre de *threads* peut donc améliorer les performances du programme.

Dans un premier temps, nous modifions la façon dont les *threads* sont gérés au sein du *runtime*. La norme OpenMP spécifie que le programme peut modifier le nombre de *threads* utilisés pour exécuter les régions parallèles en utilisant la fonction `omp_set_num_threads` ou *via* la directive `num_thread` en début de région parallèle. Malheureusement, lorsque le nombre de *threads* est modifié pendant l'exécution du programme par l'une ou l'autre de ces méthodes, le *runtime* GNU OpenMP (dans sa version 12.0) supprime les *threads* surnuméraires ou en crée de nouveaux. La création et la suppression de *threads* sont des actions coûteuses, qui, si elles sont utilisées trop fréquemment, peuvent significativement ralentir l'exécution. Nous avons par conséquent modifié l'implémentation de la gestion des *threads* pour que, lors d'un changement du nombre de *threads* utilisés par le programme, les *threads* surnuméraires soient mis en attente pour être réutilisés plus tard au besoin. Ainsi, la modification du nombre de *threads* au cours de l'exécution ne dégrade plus les performances du programme.

Dans un second temps, nous avons implémenté l'enregistrement d'événements au début et à la fin de chaque région parallèle du programme. Au moment de la compilation d'un programme utilisant OpenMP, le compilateur découpe le code de l'application. Dans le but de distribuer le code des différentes régions parallèles, le compilateur extrait les régions parallèles et les transforme automatiquement en fonctions autonomes (Cf. section 2.2.1). Pythia-OpenMP utilise l'adresse de cette fonction pour identifier la région parallèle. Par ailleurs, Pythia-OpenMP associe à chaque région parallèle deux symboles terminaux au sein de la grammaire, un pour l'entrée dans la région parallèle et un pour la fin. Ces événements sont transmis à PYTHIA au début et à la fin de chaque région parallèle.

Pythia-OpenMP enregistre également une estampille pour chaque événement soumis à PYTHIA. À la fin de l'exécution de référence, Pythia-OpenMP utilise la méthode décrite dans la section 4.3.3 afin de construire une table des durées moyennes d'exécution de chaque région parallèle en fonction du contexte dans lequel elle a été exécutée. Cette table est sauvegardée dans un fichier en parallèle de la grammaire représentant la trace de référence.

Lorsque l'utilisateur fournit la trace de référence d'une application, Pythia-OpenMP utilise PYTHIA-PREDICT pour prédire la durée de chaque région parallèle et pour ajuster le nombre de *threads* à utiliser. Au début de chaque exécution, la table des durées d'exécution des régions parallèles est chargée en mémoire. Au début de chaque région parallèle, Pythia-OpenMP demande une prédiction à PYTHIA-PREDICT sur le prochain événement à venir. Le point de progression fourni par PYTHIA-PREDICT est utilisé par Pythia-OpenMP afin de récupérer la durée d'exécution probable de la région parallèle. En comparant cette durée d'exécution à différentes valeurs seuils déterminées empiriquement, Pythia-OpenMP sélectionne le nombre de *threads* à utiliser.

L'enregistrement d'événements avec PYTHIA-RECORD, et l'appel et l'exploitation des prédictions de PYTHIA-PREDICT nécessite la modification d'une centaine de lignes de codes dans l'implémentation GNU OpenMP.

5.4.2 Modifications apportées à Lulesh

Dans le but de tester Pythia-OpenMP, nous modifions le programme Lulesh de deux manières. D'une part, Lulesh n'a pas été conçu ni testé pour un *runtime* OpenMP susceptible de modifier le nombre de *threads* utilisés en cours d'exécution. La première modification apportée à Lulesh consiste donc en différents appels à la fonction `omp_get_num_threads` afin de récupérer le nombre de *threads* au lieu de conserver celui-ci dans des variables.

D'autre part, nos premières mesures ont montré une forte amélioration des performances de Lulesh pendant l'exécution de référence. Malgré l'utilisation de PYTHIA-RECORD, l'exécution de Lulesh était parfois jusqu'à 30% plus rapide que dans sa configuration non instrumentée. En inspectant l'utilisation mémoire de Lulesh, nous avons établi que ce résultat surprenant était dû aux allocations effectuées par Pythia-OpenMP et par PYTHIA-RECORD qui venaient perturber le schéma d'allocation et de désallocation mémoire de Lulesh. Cette perturbation menait

à une amélioration significative de l'utilisation des caches mémoire du processeur et expliquait les performances observées. Afin que cet effet ne vienne pas fausser nos mesures, nous modifions Lulesh pour remplacer la plupart des allocations et désallocations mémoire du programme par l'utilisation de blocs mémoire pré-alloués au début son exécution. Ces modifications permettent de rendre négligeable l'impact des allocations mémoire de Pythia-OpenMP et de PYTHIA sur les performances de Lulesh.

5.4.3 Impact de Pythia-OpenMP sur la durée d'exécution de Lulesh

Afin de mesurer l'impact de la stratégie de parallélisme adaptatif, nous mesurons la durée d'exécution de Lulesh sur les machines Pudding et Pixel dans différentes configurations. Tous les résultats présentés dans cette section sont des moyennes calculées à partir de 10 mesures. Chaque mesure est effectuée dans trois configurations différentes de Pythia-OpenMP. Dans la configuration « Vanilla », la stratégie de parallélisme adaptative est désactivée : la totalité des *threads* disponibles sont utilisés pour chaque région parallèle du programme, conformément au comportement par défaut de l'implémentation GNU d'OpenMP. La configuration « *Pythia-record* » correspond à l'exécution de référence de Lulesh : la stratégie de parallélisme automatique est désactivée et Pythia-OpenMP utilise PYTHIA pour construire une trace d'exécution de Lulesh avant de précalculer la table contextuelle des durées d'exécution de chaque région parallèle. Enfin, dans la configuration « *Pythia-predict* », Pythia-OpenMP utilise PYTHIA-PREDICT et les données collectées pendant l'exécution de référence pour choisir le nombre de *threads* le plus adapté à chaque région parallèle.

La première expérience réalisée consiste à mesurer la variation de la durée d'exécution de Lulesh exécutée avec Pythia-OpenMP en fonction de la taille du problème. Les figures 5.4 et 5.5 montrent les résultats obtenus respectivement sur les machines Pudding et Pixel. L'axe des abscisses représente la taille du problème tandis que l'axe des ordonnées représente (en échelle logarithmique) la durée d'exécution de Lulesh. En comparant les courbes *Vanilla* (en violet) et *Pythia-record* (en vert), nous remarquons que l'enregistrement des événements avec PYTHIA-RECORD n'affecte pas significativement la durée d'exécution de Lulesh. À l'inverse, l'utilisation de la stratégie de parallélisme adaptatif utilisant PYTHIA-PREDICT (en orange) permet d'améliorer significativement les performances de Lulesh pour les problèmes de petites tailles. Par exemple, pour un problème de taille 30 sur Pudding, Lulesh s'exécute en 8,44s dans la configuration *Vanilla* et 5,25s dans la configuration *Pythia-predict*, soit un gain de performance de 38%.

La durée d'exécution gagnée avec PYTHIA-PREDICT décroît au fur et à mesure que la taille du problème augmente. Cela s'explique par le fait que les petites régions parallèles deviennent de plus en plus marginales au sein de l'exécution de Lulesh quand la taille du problème augmente.

Afin de mieux comprendre ces gains de performance, nous mesurons maintenant la durée d'exécution de Lulesh pour une taille de problème de 30 en faisant varier le nombre de *threads* maximal mis à disposition de Pythia-OpenMP. Les figures 5.6 et 5.7 montrent les résultats obtenus respectivement sur les machines Pudding et Pixel. Sur les deux figures, l'axe des abscisses représente le nombre maximal de *threads* utilisables par Pythia-OpenMP tandis que l'axe des ordonnées représente la durée d'exécution de Lulesh.

Sur les deux machines, nous constatons que les durées d'exécution de Lulesh pour les trois configurations *Vanilla*, *Pythia-record* et *Pythia-predict* sont similaires tant que le nombre de *threads* est inférieur ou égal à 8. Étudions maintenant les résultats avec plus de 8 *threads* pour chacune des deux machines.

Commençons par la machine Pudding. Au délà de 8 *threads*, nous voyons que la durée d'exécution de Lulesh augmente quand la stratégie de parallèlisme adaptatif n'est pas utilisée. Cette dégradation des performances de Lulesh est due à l'augmentation du coût de synchronisation des *threads* dans les petites régions parallèles. Nous remarquons aussi que la durée d'exécution de Lulesh croît légèrement plus vite dans la configuration *Vanilla* que dans la configuration *Pythia-record*. Cela semble être un relicat de perturbations des schémas d'accès mémoire

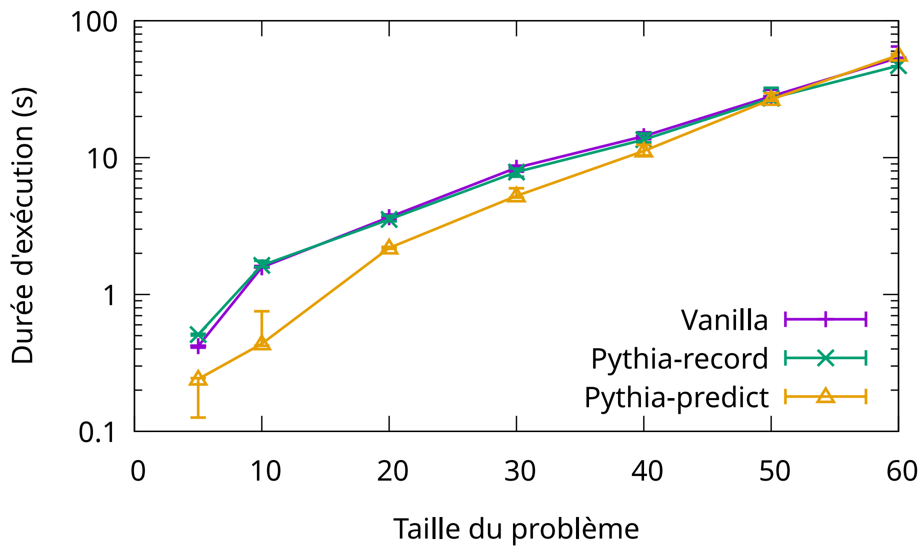


FIGURE 5.4 – Temps d'exécution de Lulesh en fonction de la taille du problème avec 24 *threads* sur Pudding.

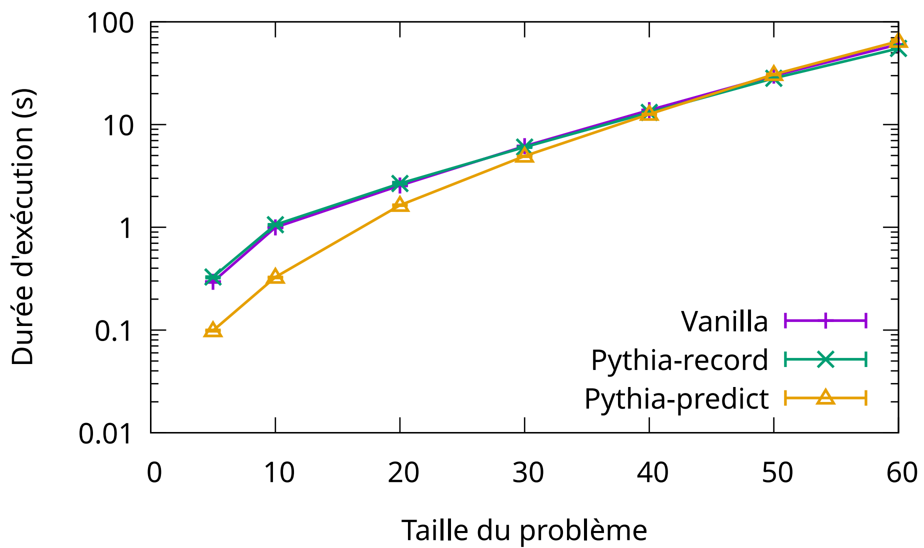


FIGURE 5.5 – Temps d'exécution de Lulesh en fonction de la taille du problème avec 16 *threads* sur Pixel.

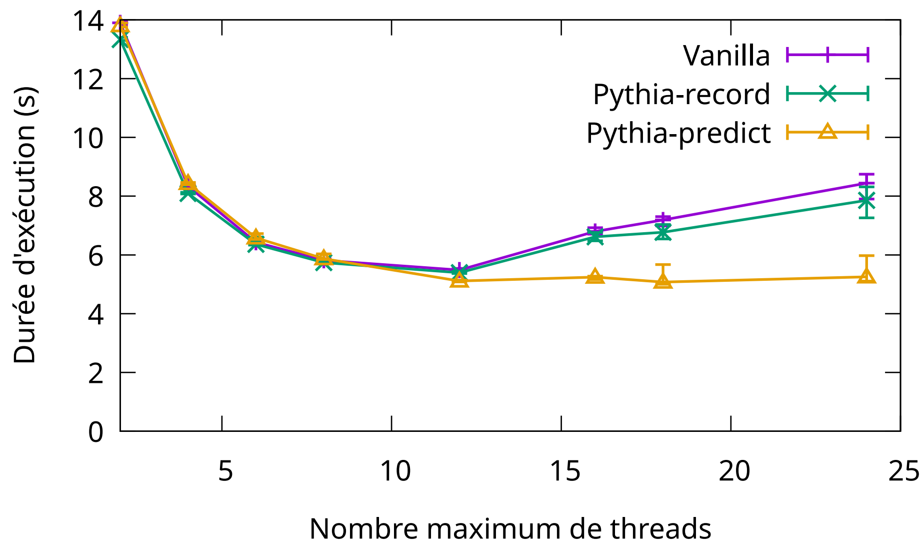


FIGURE 5.6 – Temps d'exécution de Lulesh en fonction du nombre de *threads* disponibles pour une taille de problème de 30 sur Pudding.

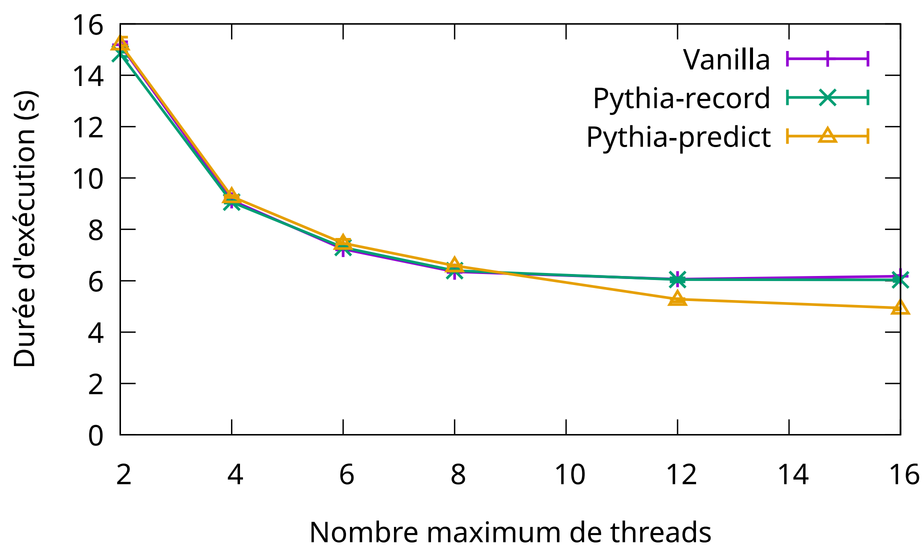


FIGURE 5.7 – Temps d'exécution de Lulesh en fonction du nombre de *threads* disponibles pour une taille de problème de 30 sur Pixel.

de Lulesh par les allocations mémoire de Pythia-OpenMP et PYTHIA-RECORD. Dans la configuration *Pythia-predict*, l'augmentation de la durée d'exécution de Lulesh avec plus de 8 *threads* est annulée par la stratégie de parallélisme adaptatif. En effet, une fois le nombre optimal de *threads* atteint pour les petites régions parallèles, Pythia-OpenMP n'utilise plus les *threads* supplémentaires disponibles. Les *threads* inutiles ne sont pas utilisés et le coût de synchronisation disparaît, en conséquence de quoi la durée d'exécution de Lulesh se stabilise au delà de 8 *threads*.

Sur la machine Pixel, la dégradation des performances de Lulesh au delà de 8 *threads* est moindre. Nous voyons malgré tout que la stratégie de parallélisme adaptatif semble tout de même permettre à Lulesh de profiter des *threads* supplémentaires.

Ces résultats montrent que PYTHIA peut être intégré à un *runtime* et que ses prédictions peuvent être utilisées pour mettre en œuvre des optimisations. Enfin, le faible coût des prédictions confirme l'utilisabilité de l'oracle pour des optimisations à grain fin, c'est-à-dire de l'ordre de la microsecondes.

5.5 Conclusion

Dans ce chapitre, nous avons présenté l'évaluation des différents mécanismes que nous avons proposés et implémentés. Dans la première partie du chapitre, nous avons évalué la collecte d'événements par PYTHIA-RECORD sur un ensemble d'applications MPI et MPI+OpenMP. Cette évaluation a montré que le surcoût engendré par PYTHIA-RECORD est faible, et négligeable pour la plupart des applications testées. Par ailleurs, la plupart des applications testées possèdent un comportement très régulier, et donc les grammaires générées comportent un faible nombre de règles.

Dans la suite du chapitre, nous avons évalué les prédictions de PYTHIA-PREDICT sur le même ensemble d'applications MPI et MPI+OpenMPI. Cette évaluation a montré que les prédictions de PYTHIA-PREDICT se révèlent correctes dans plus de 90% des cas pour la majorité des applications, y compris en changeant la taille du jeu de données utilisé et avec une avance pouvant aller jusqu'à 128 événements. De plus, le coût de ces prédictions à quelques événements d'avances est suffisamment faible pour qu'elles soient utilisées afin d'implémenter des optimisations fines (de l'ordre de la microseconde). Le coût des prédictions à long terme est plus élevé mais reste suffisamment faible pour permettre la mise en œuvre d'optimisation de l'ordre de la centaine de microsecondes.

Enfin, dans la troisième partie du chapitre, nous avons évalué l'utilisabilité de PYTHIA au sein d'un *runtime*. Cette évaluation a montré qu'il est possible d'utiliser PYTHIA pour mettre en œuvre une stratégie de parallélisme adaptatif optimisant la durée d'exécution des petites régions parallèles au sein d'un *runtime* OpenMP. L'implémentation de cette optimisation consiste en une centaine de lignes de code pour modifier la gestion des *threads* au sein du *runtime* et pour utiliser PYTHIA. Les mesures effectuées montrent que l'optimisation est fonctionnelle et permet un gain de performance significatif sur les petites régions parallèles d'un programme OpenMP.

Chapitre 6

Conclusion et perspectives

Sommaire

6.1 Contexte et problématique de la thèse . . . 69
6.2 Contributions de la thèse . . . 70
6.2.1 Réduction à la volée d'une trace d'exécution sous la forme d'une grammaire . . . 70
6.2.2 Utilisation d'une grammaire pour prédire le comportement futur d'un programme . . . 71
6.3 Perspectives et travaux futurs . . . 71
6.3.1 Pistes d'amélioration de PYTHIA . . . 71
6.3.2 Généralisation d'un langage de requête pour l'oracle, annotation dynamique de la grammaire et programmation événementielle . . . 72
6.3.3 Analyse de fichiers de journalisation . . . 72
6.3.4 Recherche de bogues au sein d'applications . . . 73

Au cours de cette thèse, nous nous sommes intéressés à la prédiction du comportement d'un programme à partir de la trace d'une de ses exécutions. Ce chapitre clôt ce manuscrit. Nous y rappelons le contexte de cette thèse ainsi que la problématique dégagée (section 6.1), nos contributions (section 6.2), et enfin les perspectives et travaux envisagés pour continuer nos recherches (section 6.3).

6.1 Contexte et problématique de la thèse

Les applications parallèles sont plus que jamais au cœur de nombreux domaines scientifiques et industriels. Cependant, la conception et la production d'applications parallèles est une tâche ardue. La complexité des machines, et surtout la diversité des architectures matérielles des ordinateurs parallèles, rendent nécessaire l'utilisation de modèles de programmation dédiés à la conception de programmes parallèles. Afin de permettre la portabilité des performances d'une machine à une autre, ces modèles de programmation introduisent des abstractions mises en œuvre par les *runtimes*. Pour ce faire, les *runtimes* doivent prendre des décisions qui impactent directement les performances des programmes.

Afin de prendre des décisions éclairées, les *runtimes* ont besoin d'anticiper le comportement futur des programmes. Actuellement, les *runtimes* s'appuient pour cela sur différentes méthodes telles que les heuristiques, les annotations du code, ou les modèles de performances. Les heuristiques, du fait de leur simplicité de mise en œuvre, restent de loin la méthode privilégiée par les *runtimes* pour estimer le comportement futur des applications. Pourtant, les heuristiques peuvent se tromper : comme elles exploitent uniquement l'état actuel de la machine, voire des

données recueillies depuis le début de l'exécution du programme, leur connaissance du comportement de l'application qui s'exécute est limitée. L'annotation de code par les développeurs et les développeuses peut renseigner un *runtime* sur le comportement futur d'un programme. Les modèles de performance permettent à un *runtime* d'expérimenter différentes configurations et optimisations afin de mesurer leur impact sur le comportement du programme. Malheureusement, le recours aux annotations de code nécessite de modifier le code source du programme, et les modèles de performances ne permettent pas d'explorer son avenir. Une alternative est d'avoir recours à un oracle. Les oracles permettent de prédire le comportement futur d'un programme en comparant l'exécution du programme à un enregistrement d'une de ses exécutions passées. Les oracles s'appuient sur l'observation que le comportement de nombreux programmes parallèles varie peu, voire pas du tout, d'une exécution à l'autre. Malgré leurs avantages, comme la possibilité d'explorer le futur du programme, ou le fait de ne pas nécessiter de modification du code source des programmes, ils sont actuellement très peu utilisés au sein des *runtime*.

Nous nous sommes intéressés à la prise de décision des *runtimes* utilisés par des applications parallèles. Ces prises de décisions imposent plusieurs contraintes. Le temps dont dispose un *runtime* pour prendre une décision est très faible, de l'ordre de la microseconde. Passé ce délai, le gain apporté par la décision risque de devenir nul, ou même négatif. Par ailleurs, la méthode de prévision doit être suffisamment précise pour permettre la prise de décision.

Nous avons donc cherché à répondre à la question : **en complétant un *runtime* avec un oracle, comment prédire le comportement futur d'une application parallèle de manière précise pour prendre de meilleures décisions ?**

6.2 Contributions de la thèse

Nous avons développé PYTHIA, un oracle générique pouvant être utilisé pour capturer le comportement d'un programme et d'en prédire le comportement futur au sein d'un *runtime*. Ces deux fonctionnalités sont implémentées respectivement au sein de PYTHIA-RECORD, qui capture le comportement du programme, et de PYTHIA-PREDICT, qui en prédit le comportement futur.

6.2.1 Réduction à la volée d'une trace d'exécution sous la forme d'une grammaire

Après avoir identifié les besoins de PYTHIA-RECORD (performance, faible encombrement mémoire, généricité, utilisation d'événements abstraits), nous avons choisi de représenter l'exécution d'un programme sous la forme d'une grammaire. Nous avons décrit précisément comment utiliser une grammaire afin de représenter la trace d'une exécution d'un programme.

Nous avons alors conçu un algorithme capable de construire à la volée une grammaire pour représenter une séquence d'événements soumis par un *runtime* à PYTHIA-RECORD pendant l'exécution d'un programme. Un *runtime* peut ainsi, grâce à PYTHIA-RECORD, enregistrer une trace d'exécution d'un programme et en capturer la structure. PYTHIA-RECORD est notamment capable de repérer les principales structures algorithmiques du programme telles que les boucles ou les séquences d'événements récurrentes. La réduction de la trace d'exécution pendant l'exécution permet en outre de réduire l'utilisation mémoire de PYTHIA-RECORD.

L'évaluation de PYTHIA-RECORD a montré que son usage pour enregistrer tous les appels aux *runtimes* MPI et OpenMP d'un panel de programmes n'avait pas d'impact significatif sur la durée d'exécution de ces programmes. L'évaluation a également montré que la plupart des applications parallèles testées possèdent un comportement très régulier, ce qui s'exprime par un faible de nombre de règles des grammaires construites.

6.2.2 Utilisation d'une grammaire pour prédire le comportement futur d'un programme

Notre deuxième contribution a été de décrire comment une grammaire représentant une trace d'exécution d'un programme peut être utilisée pour suivre l'exécution de celui-ci et pour établir des prédictions sur son comportement futur.

Dans un premier temps, nous avons développé un algorithme permettant à PYTHIA-PREDICT de comparer l'exécution d'un programme à son exécution de référence grâce à la grammaire de l'exécution de référence. Cet algorithme permet de suivre l'exécution du programme quand son exécution est identique à l'exécution de référence. Par recherche de similitudes entre les exécutions, cet algorithme permet aussi de retrouver l'état d'avancement d'un programme lorsque son exécution courante diffère de son exécution de référence. Pour cela, l'algorithme représente l'avancement d'un programme sous la forme d'une collection de ce que nous appelons des points de progression.

Dans un second temps, nous avons adapté cet algorithme afin d'explorer les différents comportements futurs possibles du programme à partir de son état d'avancement courant. En utilisant cet algorithme, PYTHIA-PREDICT est capable de renseigner un *runtime* sur les prochains événements à venir, et de probabiliser ceux-ci. Grâce au points de progression, PYTHIA-PREDICT est capable non seulement de prédire les prochaines actions du programme, mais aussi de prédire le contexte de ces actions. Nous avons montré comment un *runtime* peut mettre à profit cette information afin d'enrichir les prédictions de PYTHIA-PREDICT avec des informations contextuelles correspondant aux besoins d'une optimisation.

L'évaluation de PYTHIA-PREDICT a montré que notre oracle peut être utilisé pour implémenter une optimisation au sein d'un *runtime* OpenMP. Des mesures sur un panel de treize programmes parallèles ont montré que les prédictions de PYTHIA-PREDICT sont correctes jusqu'à plus d'une centaine d'événements en avance pour huit de ces programmes, y compris en changeant les tailles des jeux de données utilisés, et pour onze de ces programmes, sans changer la taille des jeux de données. L'évaluation a aussi montré que le coût d'une prédiction de PYTHIA-PREDICT est assez faible pour servir à des optimisations de l'ordre de la microseconde pour des prédictions à quelques événements en avance, et à des optimisations de l'ordre de quelques dizaines de microsecondes pour des prédictions à une centaine d'événements en avance.

6.3 Perspectives et travaux futurs

Dans cette section, nous présentons différentes idées et pistes de recherches pour continuer nos travaux.

6.3.1 Pistes d'amélioration de PYTHIA

Bien que PYTHIA soit actuellement utilisable au sein d'un *runtime*, nous envisageons différentes améliorations et recherches qui permettraient peut-être d'en améliorer encore les performances. Nous en faisons ici une liste :

- l'implémentation de PYTHIA-PREDICT, bien que performante, n'a pas bénéficié d'autant d'efforts d'optimisation que nous l'aurions souhaités. Nous pensons que les performances de la prédiction peuvent encore être améliorée, par exemple en modifiant la façon dont les points de progression sont stockés et manipulés au sein de PYTHIA-PREDICT ;
- nous avons vu dans l'évaluation de PYTHIA-PREDICT que ses performances étaient moins bonnes pour certaines applications dont la grammaire comportent de nombreuses règles, p.ex. Quicksilver. Ce problème de performance devrait être investigué ;
- nous avons vu que l'utilisation d'une trace d'exécution n'est pas adaptée pour capturer le comportement d'un programme utilisant la programmation par tâches. Un axe de recherche futur est donc de doter PYTHIA d'un

modèle de représentation hybride utilisant des grammaires pour représenter le déroulement de chaque tâche, et un graphe pour les dépendances entre les tâches ;

— actuellement, PYTHIA-RECORD enregistre le comportement de chaque *thread* du programme dans une grammaire différente. Ceci a deux conséquences : d'une part, il n'est pas possible de repérer une séquence qui est exécutée sur plusieurs *threads* différents ; d'autre part, il revient au *runtime* d'établir une correspondance entre les différents *threads* des différentes exécutions. En enregistrant le déroulement de l'exécution de chaque *thread* au sein d'une même grammaire, par exemple en autorisant une grammaire à avoir plusieurs règles racines, PYTHIA-PREDICT serait en mesure de ré-associer lui-même les différents *threads* des différentes exécutions, et peut-être même de s'adapter de manière transparente à un changement du nombre de *threads*. Cette fonctionnalité soulève cependant différents problèmes de recherche, par exemple pour synchroniser les modifications de la grammaire par plusieurs *threads*, et serait limitée aux seules machines à mémoire partagée.

6.3.2 Généralisation d'un langage de requête pour l'oracle, annotation dynamique de la grammaire et programmation événementielle

Nous avons vu qu'il était possible d'utiliser les points de progression des prédictions de PYTHIA-PREDICT pour prédire des valeurs numériques, et nous avons évalué cet usage avec la prédiction des durées des régions parallèles OpenMP. Une autre utilisation possible des points de progression pour prédire le comportement d'un programme serait d'utiliser les points de progression pour « étiqueter » une grammaire. Ce procédé permettrait d'associer, à l'avance, des informations à différents états d'avancement d'un programme : par exemple, à un état d'avancement donné du programme, indiquer que le programme est susceptible d'ouvrir un fichier, ou ne devrait plus envoyer de message *via* le réseau avant un certain temps. Cette technique constituerait une méthode alternative à l'exploration de la grammaire pour prédire le comportement futur d'un programme.

Afin d'étiqueter une grammaire, nous pouvons envisager de concevoir un langage de requêtes pour PYTHIA. Au moment de l'exécution d'un programme, un *runtime* pourrait demander à PYTHIA de l'informer de la survenues de certaines situations. Par exemple, un *runtime* comme NewMadeleine [6] pourrait demander à être prévenu par PYTHIA lorsque le programme s'apprête à envoyer une série de messages, ceci afin de les agréger. Autre exemple : un *runtime* pourrait demander à être informé lorsqu'une ressource est sur le point d'être inutilisée pendant un certain temps, ceci afin de l'utiliser pour l'optimisation d'un autre programme et sans gêner l'exécution du programme courant.

6.3.3 Analyse de fichiers de journalisation

La création à partir d'une séquence d'événements d'une grammaire décrivant les principales structures algorithmiques pourraient être utilisée afin d'analyser un fichier de journalisation d'une application. À titre d'exemple, tout au long de nos travaux, nous avons utilisé PYTHIA-RECORD au sein d'un autre programme que nous avons appelé PYTHIA-REDUCE. Ce programme lit ligne par ligne son entrée standard et utilise une table de hachage pour associer chaque ligne à un symbole terminal au sein d'une grammaire. En réduisant la séquence de symboles correspondants aux lignes lues sur l'entrée standard à l'aide de PYTHIA-RECORD, PYTHIA-RECORD peut alors afficher une représentation de la structure de son entrée. Nous avons implémenté différents modules d'affichage pour PYTHIA-REDUCE, qui est ainsi capable d'afficher son résultat sous une forme textuelle, sous une forme arborescente, sous la forme d'une grammaire ou encore au format `dot`, ce qui permet de générer des graphes au format `svg`. En inspectant la forme de la grammaire, PYTHIA-REDUCE est également capable de fournir visuellement des informations complémentaires sur la trace, comme le fait qu'une ligne soit unique au sein de la trace. Cet utilitaire

profite des performances de PYTHIA-RECORD et s'est avéré à la fois pratique et agréable à utiliser tout au long de la seconde moitié de cette thèse. Il conviendrait d'étendre les capacités d'analyse de PYTHIA-REDUCE afin d'explorer ses capacités comme outil pour assister à la compréhension de programmes plus complexes, pour déboguer ou, par exemple, aider à la recherche de chemins de code menant à des interblocages.

6.3.4 Recherche de bogues au sein d'applications

Au cours de cette thèse, nous avons développé un algorithme performant capable de découvrir à la volée la structure d'une trace d'exécution. Cet algorithme est en particulier capable de découvrir les structures de boucles imbriquées. Nous avons de même développé une technique permettant de trouver, toujours à la volée, des similitudes entre une séquence d'événements et une trace d'exécution entièrement constituée et réduite. Développer un mécanisme de comparaison des grammaires en cours de construction pourrait permettre de déceler des bogues pendant l'exécution d'un programme. De même, comparer l'exécution courante d'un programme à un ensemble d'exécutions de référence pourrait permettre de détecter des comportements déviants, par exemple à cause de bogues ou de failles de sécurité.

Bibliographie

[1] Azzeddine Amiar, Mickaël Delahaye, Ylies Falcone, and Lydie Du Bousquet. Compressing microcontroller execution traces to assist system analysis. In *International Embedded Systems Symposium*, pages 139–150, 2013.

[2] Joseph Antony, Pete P Janes, and Alistair P Rendell. Exploring thread and memory placement on numa architectures : Solaris and linux, ultrasparc/fireplane and opteron/hypertransport. In *International Conference on High-Performance Computing*, pages 338–352, 2006.

[3] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. Automatic calibration of performance models on heterogeneous multicore architectures. In Hai-Xiang Lin, Michael Alexander, Martti Forsell, Andreas Knüpfer, Radu Prodan, Leonel Sousa, and Achim Streit, editors, *Euro-Par 2009 – Parallel Processing Workshops*, pages 56–65, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU : a unified platform for task scheduling on heterogeneous multicore architectures. In *European Conference on Parallel Processing*, pages 863–874. Springer, 2009.

[5] Charles Aulagnon, Damien Martin-Guillerez, François Rue, and François Trahay. Runtime function instrumentation with eztrace. In *European Conference on Parallel Processing*, pages 395–403, 2012.

[6] Olivier Aumage, Elisabeth Brunet, Nathalie Furmento, and Raymond Namyst. New madeleine : A fast communication scheduling engine for high performance networks. In *Workshop on Communication Architecture for Clusters*, pages 1–8, 2007.

[7] Ayguade, Eduard and Copty, Nawal and Duran, Alejandro and Hoeflinger, Jay and Lin, Yuan and Massaioli, Federico and Teruel, Xavier and Unnikrishnan, Priya and Zhang, Guansong. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3) :404–418, 2009.

[8] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center, 1995.

[9] Babak Behzad, Surendra Byna, Stefan M. Wild, Mr. Prabhat, and Marc Snir. Improving parallel i/o autotuning with performance modeling. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '14, page 253–256, New York, NY, USA, 2014. Association for Computing Machinery.

[10] Dean Michael Berris, Alistair Veitch, Nevin Heintze, Eric Anderson, and Ning Wang. Xray : A function call tracing system. Technical report, Google Inc., 2016. A white paper on XRay, a function call tracing system developed at Google.

[11] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk : An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1) :55–69, 1996.

[12] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, et al. Grid'5000 : A large scale and highly reconfigurable experimental grid testbed. *The International Journal of High Performance Computing Applications*, 20(4) :481–494, 2006.

[13] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. Parsec : Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6) :36–45, 2013.

[14] Mohamed Said Mosli Bouksiaa, François Trahay, Alexis Lescouet, Gauthier Voron, Remi Dulong, Amina Guermouche, Elisabeth Brunet, and Gaël Thomas. Using differential execution analysis to identify thread interference. *IEEE Transactions on Parallel and Distributed Systems*, 30(12) :2866–2878, 2019.

[15] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 265–275. IEEE, 2003.

[16] Élisabeth Brunet, Francois Trahay, Alexandre Denis, and Raymond Namyst. A Sampling-Based Approach for Communication Libraries Auto-Tuning. In *IEEE International Conference on Cluster Computing*, pages 299–307, 2011.

[17] Bryan Buck and Jeffrey K Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4) :317–329, 2000.

[18] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[19] Surendra Byna, Yong Chen, Xian-He Sun, Rajeev Thakur, and William Gropp. Parallel i/o prefetching using mpi file caching and i/o signatures. In *SC'08 : Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2008.

[20] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[21] William E Cohen. Tuning programs with OProfile. *Wide Open Magazine*, 1 :53–62, 2004.

[22] Vincent Danjean, Raymond Namyst, and Pierre-André Wacrenier. An efficient multi-level trace toolkit for multi-threaded applications. In *European Conference on Parallel Processing*, pages 166–175. Springer, 2005.

[23] Benhur de Oliveira Stein, J Chassin de Kergommeaux, and G Mounié. Pajé trace file format. Technical report, Technical report, ID-IMAG, Grenoble, France, 2002. http ://www-id. imag. fr . . ., 2010.

[24] Peter Deutsch and Jean-Lou Gailly. Zlib compressed data format specification version 3.3. RFC 1950, RFC Editor, 1996.

[25] Lamia Djoudi, Denis Barthou, Patrick Carribault, Christophe Lemuet, Jean-Thomas Acquaviva, William Jalby, et al. Maqao : Modular assembler quality analyzer and optimizer for itanium 2. In *Proc. of the 4th Workshop on EPIC architectures and compiler technology, San Jose*, volume 200, 2005.

[26] Matthieu Dorier, Shadi Ibrahim, Gabriel Antoniu, and Rob Ross. Omnisc'IO : a grammar-based approach to spatial and temporal I/O patterns prediction. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 623–634, 2014.

[27] Alexandre E Eichenberger, John Mellor-Crummey, Martin Schulz, Michael Wong, Nawal Copty, Robert Dietrich, Xu Liu, Eugene Loh, and Daniel Lorenz. OMPT : An OpenMP tools application programming interface for performance analysis. In *International Workshop on OpenMP*, pages 171–185, 2013.

[28] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang Nagel, and Felix Wolf. Open Trace Format 2 : The Next Generation of Scalable Trace Formats and Support Libraries. In *Applications, Tools and Techniques on the Road to Exascale Computing*, volume 22, pages 481 – 490, 01 2012.

[29] Giulio Eulisse and Lassi Tuura. IgProf profiling tool. In *Proc. in Computing in High Energy Physics and Nuclear Physics*, Interlaken, Switzerland, October 2004.

[30] MPI Forum. A Message Passing Interface Standard : Version 2.2, 2009.

[31] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. Challenges of memory management on modern NUMA systems. *Communications of the ACM*, 58(12) :59–66, Nov 2015.

[32] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The scalasca performance toolset architecture. *Concurrency and computation : Practice and experience*, 22(6) :702–719, 2010.

[33] Brice Goglin and Nathalie Furmento. Memory migration on next-touch. In *Linux Symposium*, 2009.

[34] Susan L Graham, Peter B Kessler, and Marshall K McKusick. Gprof : A call graph execution profiler. *ACM Sigplan Notices*, 17(6) :120–126, 1982.

[35] Mike Heroux and Simon Hammond. MiniFE : finite element solver, 2019.

[36] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. Taskflow : A lightweight parallel and heterogeneous task graph computing system. *IEEE Transactions on Parallel and Distributed Systems*, 33(6) :1303–1320, 2021.

[37] Roman Iakymchuk and François Trahay. LiTL : Lightweight Trace Library. https://github.com/trahay/LiTL.

[38] Mohammed Islam Naas, François Trahay, Alexis Colin, Pierre Olivier, Stéphane Rubini, Frank Singhoff, and Jalil Boukhobza. EZIOTracer : Unifying Kernel and User Space I/O Tracing for Data-Intensive Applications. *SIGOPS Oper. Syst. Rev.*, 55(1) :88–98, jun 2021.

[39] Ian Karlin, Jeff Keasler, and JR Neely. Lulesh 2.0 updates and changes. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2013.

[40] Ronan Keryell, Ruyman Reyes, and Lee Howes. Khronos SYCL for OpenCL : a tutorial. In *Proceedings of the 3rd International Workshop on OpenCL*, pages 1–1, 2015.

[41] Alain Ketterlin and Philippe Clauss. Prediction and trace compression of data access addresses through nested loop recognition. In *IEEE/ACM international symposium on Code generation and optimization*, pages 94–103, 2008.

[42] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E Nagel. Introducing the open trace format (OTF). In *International Conference on Computational Science*, pages 526–533, 2006.

[43] Andreas Knupfer and Wolfgang E Nagel. Construction and compression of complete call graphs for post-mortem program trace analysis. In *International Conference on Parallel Processing (ICPP)*, pages 165–172, 2005.

[44] Alexey Kukanov and Michael J Voss. The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks. *Intel Technology Journal*, 11(4), 2007.

[45] Adam J Kunen, Teresa S Bailey, and Peter N Brown. Kripke-a massively parallel transport mini-app. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2015.

[46] Christoph Lameter. NUMA (Non-Uniform Memory Access) : An Overview : NUMA becomes more common because memory controllers get close to execution units on microprocessors. *Queue*, 11(7) :40–51, 2013.

[47] James R Larus. Whole program paths. *ACM SIGPLAN Notices*, 34(5) :259–269, 1999.

[48] Chris Lattner and Vikram Adve. LLVM : A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[49] Hui Lei and Dan Duchamp. An Analytical Approach to File Prefetching. *USENIX ATC*, page 12, 1997.

[50] Edgar A León, Brice Goglin, and Andres Rubio Proaño. M&mms : navigating complex memory spaces with hwloc. In *Proceedings of the International Symposium on Memory Systems*, pages 149–155, 2019.

[51] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin : building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6) :190–200, 2005.

[52] Steven McCanne and Van Jacobson. The BSD Packet Filter : A New Architecture for User-level Packet Capture. In *USENIX winter*, volume 46, 1993.

[53] Matthias S Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E Nagel. Developing scalable applications with vampir, vampirserver and vampirtrace. In *PARCO*, volume 15, pages 637–644, 2007.

[54] Aaftab Munshi. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, 2009.

[55] Mohammed Islam Naas, François Trahay, Alexis Colin, Pierre Olivier, Stéphane Rubini, Frank Singhoff, and Jalil Boukhobza. EZIOTracer : unifying kernel and user space I/O tracing for data-intensive applications. In *Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, pages 1–11, 2021.

[56] Craig G Nevill-Manning and Ian H Witten. Identifying hierarchical structure in sequences : A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7 :67–82, 1997.

[57] Xing Pan and Frank Mueller. Controller-aware memory coloring for multicore real-time systems. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 584–592, 2018.

[58] Simon J Pennycook, Jason D Sewall, and Victor W Lee. A metric for performance portability. *arXiv preprint arXiv :1611.07409*, 2016.

[59] Eric S Raymond. *The art of Unix programming*. Addison-Wesley Professional, 2003.

[60] James Reinders. *VTune performance analyzer essentials*, volume 9. Intel Press Santa Clara, 2005.

[61] David F Richards, Ryan C Bleile, Patrick S Brantley, Shawn A Dawson, Michael Scott McKinley, and Matthew J O'Brien. Quicksilver : a proxy app for the Monte Carlo transport code mercury. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 866–873. IEEE, 2017.

[62] Barbara G Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3) :216–226, 1979.

[63] Sameer S Shende and Allen D Malony. The TAU parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2) :287–311, 2006.

[64] Alan Jay Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems (TODS)*, 3(3) :223–247, 1978.

[65] Herb Sutter et al. The free lunch is over : A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3) :202–210, 2005.

[66] Saeed Taheri, Ian Briggs, Martin Burtscher, and Ganesh Gopalakrishnan. Difftrace : Efficient whole-program trace analysis and diffing for debugging. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12, 2019.

[67] Toshiyuki Takahashi, Shinji Sumimoto, Atsushi Hori, Hiroshi Harada, and Yutaka Ishikawa. Pm2 : High performance communication middleware for heterogeneous network environments. In *SC'00 : Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, pages 16–16, 2000.

[68] Koji Taniguchi, Takashi Ishio, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Extracting sequence diagram from execution trace of Java program. In *International Workshop on Principles of Software Evolution (IWPSE)*, pages 148–151, 2005.

[69] Bartosz Taudul. Tracy Profiler. https://github.com/wolfpld/tracy.

[70] François Trahay, Elisabeth Brunet, Mohamed Mosli Bouksiaa, and Jianwei Liao. Selecting points of interest in traces using patterns of events. In *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 70–77, 2015.

[71] François Trahay, François Rue, Mathieu Faverge, Yutaka Ishikawa, Raymond Namyst, and Jack Dongarra. EZTrace : a generic framework for performance analysis. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 618–619. IEEE, 2011.

[72] Ulrike Yang, Robert Falgout, and Jongsoo Park. Algebraic Multigrid Benchmark. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2017.

Annexe A

API de Pythia

Dans cette annexe, nous présentons l'interface C++ mise à disposition des *runtimes* par PYTHIA-RECORD et PYTHIA-PREDICT.

La figure A.1 détaille l'interface de PYTHIA-RECORD. PYTHIA-RECORD expose différents types, parmi lesquels trois classes :

- `Terminal` représente un symbole terminal ;
- `NonTerminal` représente un symbole non terminal et est utilisé pour désigner la racine de la grammaire ;
- `Grammar` représente une grammaire.

PYTHIA-RECORD met à disposition de l'utilisateur quatre fonctions :

- `new_terminal` permet de déclarer un nouveau symbole terminal et d'y attacher des données utilisateur ;
- `get_payload` permet de récupérer les données utilisateur associées à un terminal lors de la création de ce dernier ;
- `append_terminal` permet d'ajouter un nouveau symbole terminal à la fin de la règle racine d'une grammaire. Si aucun symbole non terminal n'est fourni, la fonction crée une nouvelle règle racine et retourne le symbole correspondant ;
- `print_bin_file` permet d'enregistrer une grammaire dans un fichier. La fonction prend en argument la grammaire à sauvegarder, le flux de sortie et un *callback* permettant d'associer à chaque symbole terminal une chaîne de caractères pour l'identifier au rechargement de la grammaire.

```
struct Terminal;
struct NonTerminal;
struct Grammar { ... };

auto new_terminal(Grammar & g, void * payload) -> Terminal *;
auto get_payload(Terminal *) -> void *;

auto append_terminal(Grammar & g, NonTerminal * nt, Terminal * t) -> NonTerminal *;

auto print_bin_file(Grammar const &, std::ostream &, terminal_printer const &) -> void;
```

FIGURE A.1 – Détail de l'API de PYTHIA-RECORD.

La figure A.2 détaille l'interface de PYTHIA-PREDICT. PYTHIA-PREDICT expose différents types, parmi lesquels :

- `GrammarNode` permet de désigner l'utilisation d'un symbole au sein de la grammaire ;

— `StateOfProgressNode` permet de désigner un élément d'un point de progression ;
— `StateOfProgress` représente un point de progression ;
— `Prediction` est une collection de points de progression permettant de parcourir l'arbre des possibles suite à une demande de prédiction. En plus des différents points de progression correspondant à la prédiction, la classe `Prediction` contient des données permettant de trier les futurs alternatifs par probabilité décroissante.

PYTHIA-RECORD met à disposition de l'utilisateur quatre fonctions :

— `load_bin_file` recharge une grammaire depuis un fichier à l'aide d'une fonction passée en argument permettant de ré-associer les événements entre eux d'une exécution sur l'autre ;
— `get_prediction_from_state_of_progress` demande à PYTHIA-PREDICT d'initier une prédiction sur le comportement futur du programme ;
— `get_first_alternative` permet d'explorer le prochain événement futur alternatif de la prédiction ;
— `get_first_next` permet d'obtenir le prochain événement prédit ;
— `get_prevalence` permet de connaître le nombre d'événements de la trace de référence désignés par un point de progression.

```
class GrammarNode;
class StateOfProgressNode {
    GrammarNode const * node;
    unsigned int repeats;
};
using StateOfProgress = std::vector<StateOfProgressNode>;
class Prediction {
    std::vector<StateOfProgress> states_of_progress;
    ...
};

auto load_bin_file(Grammar &, std::istream &, terminal_reader const &) -> void;
auto get_prediction_from_state_of_progress(StateOfProgress const &) -> Prediction;
auto get_first_next(Prediction&) -> bool;
auto get_first_alternative(Prediction&) -> bool;
auto get_prevalence(Prediction const &) -> size_t;
```

FIGURE A.2 – Détail de l'API de PYTHIA-PREDICT.

Annexe B

Liste des fonctions interceptées par le *runtime* MPI-OpenMP modifié

Nous listons ici les différentes fonctions des *runtimes* MPI et OpenMP interceptées par notre *runtime* Pythia-MPI+OpenMP. Les fonctions du *runtime* OpenMP sont listées dans la table B.1 et celles du *runtime* MPI dans la table B.1. Pour chaque fonction, nous renseignons, s'il y a lieux, les arguments pris en considération pour choisir si deux appels à la fonction correspondent à un symbole terminal. Nous indiquons aussi si l'appel à la fonction donne lieu à une demande de prédiction de la part de Pythia-MPI+OpenMP auprès de PYTHIA.

Function interceptée	Arguments considérés	Déclenche une prédiction
GOMP_atomic_enter		non
GOMP_atomic_exit		non
GOMP_barrier		oui
GOMP_parallel_start		oui
GOMP_parallel_end		non
GOMP_critical_start		non
GOMP_critical_end		non
GOMP_parallel_loop_static		non
GOMP_parallel_loop_dynamic		non
GOMP_parallel_loop_guided		non
GOMP_parallel_loop_runtime		non
GOMP_loop_runtime_start		non
GOMP_loop_runtime_next		non
GOMP_loop_end		non
GOMP_ordered_start		non
GOMP_ordered_end		non
GOMP_sections_start		non
GOMP_sections_next		non
GOMP_single_start		non
GOMP_single_copy_start		non
GOMP_single_copy_end		non
GOMP_parallel		non

TABLE B.1 – Fonctions du *runtime* GNU OpenMP interceptées par Pythia-MPI+OpenMP.

Function interceptée	Arguments considérés	Déclenche une prédiction
MPI_Allgather		oui
MPI_Allgatherv		oui
MPI_Allreduce		oui
MPI_Alltoall		oui
MPI_Alltoallv		oui
MPI_Barrier		oui
MPI_Bcast	$root$	oui
MPI_Bsend	$dest, count, data_type$	oui
MPI_Gather	$root$	oui
MPI_Gatherv	$root$	oui
MPI_Get	$target_rank, target_count, target_data_type$	non
MPI_Iallgather		non
MPI_Iallgatherv		non
MPI_Iallreduce		non
MPI_Ialltoall		non
MPI_Ialltoallv		non
MPI_Ibarrier		non
MPI_Ibcast	$root$	non
MPI_Ibsend	$dest, count, data_type$	non
MPI_Igather	$root$	non
MPI_Igatherv	$root$	non
MPI_Irecv	$src, count, data_type$	non
MPI_Ireduce	$root, count, data_type$	non
MPI_Ireduce_scatter	$count, data_type$	non
MPI_Irsend	$dest, count, data_type$	non
MPI_Iscan	$op, count, data_type$	non
MPI_Iscatter	$root, recv_count, recv_type$	non
MPI_Iscatterv	$root, recv_count, recv_type$	non
MPI_Isend	$dest, count, data_type$	non
MPI_Issend	$dest, count, data_type$	non
MPI_Probe		non
MPI_Put	$target_rank, target_count, target_data_type$	non
MPI_Recv	$source, count, data_type$	non
MPI_Reduce	$root, op, count, data_type$	oui
MPI_Reduce_scatter	$root, op, count, data_type$	oui
MPI_Rsend	$dest, count, data_type$	non
MPI_Scan	$op, count, data_type$	non
MPI_Scatter	$root, recv_count, recv_type$	oui
MPI_Scatterv	$root, recv_count, recv_type$	oui
MPI_Send	$dest, count, data_type$	non
MPI_Sendrecv	$dest, count, data_type$	non
MPI_Sendrecv_replace	$dest, count, data_type$	non
MPI_Ssend	$dest, count, data_type$	non
MPI_Start		non
MPI_Startall	$count$	non
MPI_Wait		oui
MPI_Waitall	$count$	oui
MPI_Waitany	$count$	oui
MPI_Waitsome	$count$	oui

TABLE B.2 – Fonctions du *runtime* MPI interceptées par Pythia-MPI+OpenMP.

ECOLE DOCTORALE

Titre : De la collecte de trace à la prédiction du comportement d'applications parallèles

Mots clés : HPC ; supports d'exécution ; analyse de performance ; prédiction de performance.

Résumé : Afin d'exploiter les ressources des serveurs et des supercalculateurs, les développeurs ont recours à des modèles de programmations spécifiques qui sont mis en œuvre par des *runtimes* dont le rôle est de permettre à chaque programme d'exploiter pleinement les capacités de la machine qui l'exécute. Pour cela, les *runtimes* doivent prendre des décisions qui ont un impact direct sur les performances. Pour prendre de bonnes décisions, les *runtimes* essaient d'anticiper le comportement futur des programmes, mais les moyens à leur disposition sont limités.
Nous présentons PYTHIA, un oracle générique permettant aux *runtimes* de prédire le comportement futur d'un programme. Nous décrivons comment enregistrer une trace d'exécution d'un programme pour en capturer la structure sous la forme d'une grammaire. Nous développons un algorithme performant capable de construire une telle grammaire à la volée pendant l'exécution d'un programme sans dégrader ses performances. Nous montrons ensuite comment utiliser une grammaire représentant la structure d'une exécution d'un programme pour prédire son comportement futur lors de ses exécutions ultérieures. PYTHIA permet en particulier d'explorer un arbre probabilisé des prochaines actions potentielles d'un programme.
L'évaluation de notre travail montre que les prédictions de PYTHIA peuvent être utilisées pour implémenter des optimisations au sein d'un *runtime*. Nous faisons aussi la démonstration de l'utilisabilité de PYTHIA en l'utilisant pour mettre en œuvre une stratégie de parallélisme adaptatif au sein d'un *runtime* OpenMP existant.

Title : From trace collection to the prediction of the behaviour of parallel applications

Keywords : HPC ; Runtime systems ; Performance analysis ; Performance prediction.

Abstract : In order to exploit the resources of servers and supercomputers, developers use specific programming models that are implemented by runtimes. Runtimes allow each program to fully exploit the capacities of the machine that executes it. To do this, runtimes take decisions that have a direct impact on the performance of the programs. In order to take good decisions, runtimes try to anticipate the future behavior of the programs, but the means at their disposal are limited.
We present PYTHIA, a generic oracle allowing runtimes to predict the future behavior of a program. We describe how to record an execution trace and to capture its structure in the form of a grammar. We develop an algorithm capable of building such a grammar on the fly during the execution of a program without degrading its performance. We then show how to use a grammar representing the structure of a program execution to predict its future behavior during its subsequent executions. In particular, PYTHIA allows to explore a probabilized tree of potential next actions of a program.
The evaluation of our work shows that the predictions of PYTHIA can be used to implement optimizations within a runtime. We have also demonstrated the usability of PYTHIA by using it to implement an adaptive parallelism strategy within an existing OpenMP runtime.

Institut Polytechnique de Paris
91120 Palaiseau, France