PYTHIA : un oracle pour guider les décisions des runtimes

Alexis Colin, François Trahay, Denis Conan

Compas'2022 : Parallélisme / Architecture/ Système
MIS/UPJV - Amiens France, 5-8 juillet 2022

PYTHIA : un oracle pour guider les décisions des runtimes

Alexis Colin, François Trahay, Denis Conan

Télécom SudParis, Institut Polytechnique de Paris,
prénom.nom@telecom-sudparis.eu

Résumé
Les supports d'exécution, ou *runtimes*, sont des systèmes utilisés pour exploiter efficacement le matériel au sein des applications parallèles. Les *runtimes* dissimulent la complexité de la gestion du matériel derrière des interfaces haut-niveau. Pour garantir les meilleures performances possibles, ils doivent prendre des décisions en s'appuyant sur des heuristiques afin d'anticiper le comportement futur de l'application. Dans cet article, nous présentons PYTHIA, un oracle exploitant l'aspect déterministe de nombreuses applications parallèles. En enregistrant une trace d'exécution, PYTHIA capture le comportement du programme. Cette trace d'exécution est ensuite utilisée lors des exécutions ultérieures afin d'offrir aux *runtimes* des prévisions sur le comportement futur de l'application. Grâce à PYTHIA, nous avons implémenté une stratégie de parallélisme adaptatif dans GNU OpenMP qui permet de gagner jusqu'à 38% de temps d'exécution sur certains programmes.

Mots-clés : Runtime, Prévisions, Performances

1. Introduction

La complexité croissantes des serveurs et ordinateurs rend fastidieux le développement d'applications parallèles exploitant pleinement les ressources des machines. Afin d'écrire de manière efficiente de tels programmes, les développeurs utilisent des supports d'exécution, ou *runtimes*, tels que des bibliothèques de communication, d'ordonnancement de tâches ou de gestion mémoire. Ces *runtimes* dissimulent la complexité des machines derrières des interfaces communes à des matériels variés et doivent donc prendre des décisions quant à la façon d'utiliser ces matériels. Ces décisions qui peuvent avoir un impact significatif sur les performances des programmes reposent habituellement sur des heuristiques. Ces dernières utilisent des données collectées pendant l'exécution du programme pour estimer son comportement futur. Par exemple, la politique d'allocation *first touch* utilisée par GNU/Linux alloue chaque page mémoire sur le nœud NUMA associé au premier *thread* qui y accède. Cette heuristique suppose que ce *thread* va probablement continuer à utiliser cette page mémoire dans un futur proche ; mais l'heuristique peut se tromper et dégrader les performances.
Observant que le comportement de nombreuses applications parallèles varie peu, voire pas du tout, d'une exécution à l'autre, nous conjecturons que la connaissance du déroulement d'une exécution passée peut aider à prédire son comportement futur plus efficacement que ne le font les heuristiques actuelles. Dans cet article, nous présentons PYTHIA, un oracle offrant la possibilité de prévoir le comportement futur d'une application. Lors de la première exécution d'une application, le *runtime* notifie PYTHIA de la survenue d'événements notables. Ces événements sont collectés et analysés par PYTHIA, qui en sauvegarde une trace sous la forme

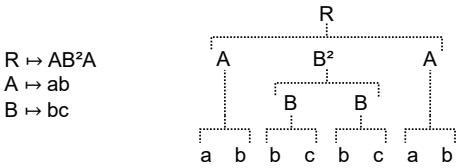


FIGURE 1 – Dépliage d'une grammaire représentant la séquence « abbcbcab ».

d'une grammaire à la fin de l'exécution. Lors de l'exécution suivante, l'oracle utilise les données sauvegardées afin de fournir des prévisions au *runtime*. Nous faisons la démonstration de l'utilisation de PYTHIA au sein de GNU OpenMP. PYTHIA prévoit la durée de chaque section parallèle OpenMP afin que le *runtime* puisse adapter le nombre de *threads* utilisés en fonction de la charge de travail à venir.
Nous présentons PYTHIA dans la section 2. Dans les sections 3 et 4, nous présentons respectivement comment PYTHIA peut être utilisé dans un *runtime* pour améliorer les performances d'applications OpenMP ainsi que les résultats obtenus. Nous discutons de l'état de l'art dans la section 5 et nous concluons cet article dans la section 6.

2. La bibliothèque PYTHIA

Nous proposons PYTHIA, une bibliothèque permettant aux *runtimes* d'enregistrer des événements pendant l'exécution d'un programme afin d'obtenir des prévisions lors des exécutions futures. Pendant la première exécution du programme, que nous nommons *exécution de référence*, PYTHIA-RECORD recueille les événements et les conserve dans une structure de données représentant le comportement du programme. Lors des exécutions ultérieures du programme, PYTHIA-PREDICT charge les données produites et compare les deux exécutions. Un *runtime* peut alors l'interroger pour connaître les événements qui devraient survenir à l'avenir et dans combien de temps. Le *runtime* peut alors prendre des décisions en conséquence.

Réduction de traces à la volée

Lors de l'exécution de référence, le runtime notifie PYTHIA-RECORD de différents événements qui peuvent être par exemple l'entrée ou la sortie de certaines fonctions comme `MPI_Send`, l'entrée dans une structure de contrôle comme une boucle ou une région parallèle, ou d'autres événements importants comme un envoi de message ou la soumission d'une tâche. PYTHIA-RECORD associe chacun de ces événements à un symbole terminal dans une grammaire représentant la structure du programme. Par convention, les symboles terminaux sont notés en minuscule et les symboles non terminaux en majuscule. Le nombre de répétitions successives d'un symbole est noté en exposant ; il est ignoré quand il vaut 1. La racine de la grammaire est notée R. La figure 1 montre comment une grammaire peut représenter une trace par remplacements successifs dans la racine de la grammaire de tous les symboles non terminaux par la séquence de symboles associée.
Comme dans d'autres travaux [1, 10], et afin d'inférer la structure du programme, PYTHIA-RECORD s'assure qu'à tout instant la grammaire respecte les règles suivantes :

— tous les symboles non terminaux représentent une séquence d'événements redondante et sont donc utilisés au moins deux fois dans la grammaire ;
— tout couple de symboles apparaît au plus deux fois côte à côte dans la grammaire : toutes les séquences redondantes de la trace sont représentées par un symbole non terminal ;

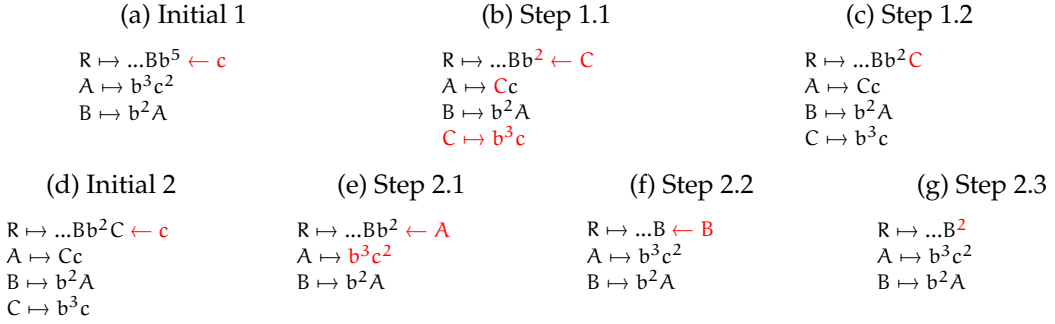


FIGURE 2 – Ajout d'un symbol dans la grammaire par PYTHIA-RECORD.

— aucun symbole n'apparaît deux fois côte à côte dans la grammaire : toutes les répétitions de la forme « $a^n a^m$ » sont fusionnées en « a^{n+m} ».

La figure 2 montre un exemple complet de mise à jour d'une grammaire pour y ajouter deux fois de suite le symbole c. La première étape consiste ajouter c à la fin de la séquence R finissant par « b^5 » (Fig. 2a). Comme la séquence « bc » existe déjà dans la grammaire sous la forme « b^3c », un nouveau symbole non terminal C est créé pour la représenter et « b^3 » est retiré de la fin de R (Fig. 2b). Ensuite, PYTHIA-RECORD ajoute le symbole C à R. La séquence « bC » n'est pas présente dans la grammaire, donc PYTHIA-RECORD ajoute C à la fin de R (Fig. 2c). PYTHIA-RECORD essaie ensuite d'ajouter une seconde fois le symbole c à la séquence R, laquelle se termine par C (Fig. 2d). La séquence « Cc » est déjà représentée par A et est donc réutilisée. La séquence « C » est retirée de la fin de R ; n'étant plus utilisée qu'une fois, elle est supprimée de la grammaire et PYTHIA-RECORD essaie désormais d'ajouter A à R (Fig. 2e). La séquence « b^2A » étant déjà représentée par B, « b^2 » est retiré de la fin de R et PYTHIA-RECORD essaie d'ajouter B à R (Fig. 2f). B étant déjà le dernier symbole de R, PYTHIA-RECORD incrémente son nombre de répétitions (Fig. 2g).

Recherche de correspondances entre les exécutions du programme

Lors des exécutions suivantes, au moment du lancement du programme PYTHIA-PREDICT charge la grammaire pour comparer l'exécution courante à l'exécution de référence. Pour cela, PYTHIA-PREDICT tient à jour une liste de *correspondances* possibles entre la grammaire et la séquence d'événements actuelle. Chaque correspondance consiste en un chemin dans la grammaire, du terminal représentant le dernier événement soumis par le *runtime* au symbole représentant la plus petite séquence englobante contenant la séquence en cours. La figure 3 montre comment PYTHIA-PREDICT peut repérer toutes les occurrences d'un événement survenues dans un contexte donné. En mémorisant l'utilisation du symbole b au sein de la séquence « ab » associée à A et l'utilisation de A au sein de la séquence « Ac » associée à C, PYTHIA-PREDICT peut identifier toutes les apparitions du symbole b précédées du symbole a et suivies du symbole c dans la séquence « abcabdababc ». Au moment de l'envoi par le *runtime* du premier événement de l'exécution, PYTHIA-PREDICT conserve uniquement le terminal associé. En effet, celui-ci constitue à lui seul la séquence complète d'événements reçus par PYTHIA-PREDICT. Lorsque PYTHIA-PREDICT reçoit un nouvel événement du *runtime*, il met à jour sa liste de chemins dans la grammaire :

— les chemins dont le terminal n'est pas suivi du terminal représentant le nouvel événement sont éliminés ;

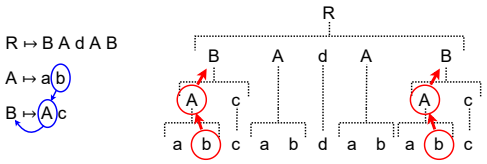


FIGURE 3 – Chemin dans la grammaire (en bleu) représentant un ensemble de correspondances possibles dans une trace (en rouge)

— dans le cas où la fin de la séquence est atteinte, PYTHIA-PREDICT cherche parmi les séquences englobantes celles contenant la nouvelle séquence.

Prévision des événements à venir

À partir de la grammaire et de la liste des correspondances possibles entre l'exécution courante et l'exécution de référence, PYTHIA-PREDICT peut simuler la survenue de nouveaux événements et ainsi simuler l'exécution future du programme. De plus, il est possible pour tout chemin dans la grammaire d'y associer différentes statistiques. En particulier, pour chaque chemin possible dans la grammaire, PYTHIA-RECORD sauvegarde le temps moyen séparant les événements qu'il représente de leur prédécesseur lors de l'exécution de référence. Grâce à ceci, PYTHIA-PREDICT est capable à tout instant de calculer la date des événements à venir à partir des chemins constituant la liste de correspondances potentielles entre l'exécution courante et l'exécution de référence.

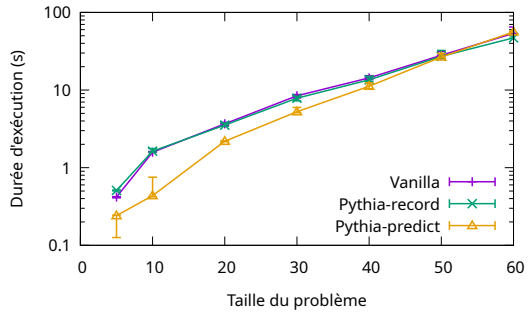
3. Utilisation de PYTHIA dans un runtime : une optimisation dans l'implémentation GNU d'OpenMP

Le *runtime* GNU OpenMP est l'implémentation d'OpenMP utilisée par le compilateur GCC. OpenMP permet d'annoter le code d'un programme afin d'indiquer quelles parties de celui-ci peuvent être parallélisées. Le *runtime* crée automatiquement des *threads*, les synchronise, et répartit la charge de travail entre eux. Toutefois, le nombre de *threads* à utiliser pour une région parallèle est un compromis entre le coût de synchronisation des *threads* et le gain dû à l'exécution parallèle du code. Si GNU OpenMP utilise généralement le maximum de *threads* possible, cela peut devenir coûteux si l'application consiste en de nombreuses petites régions parallèles. En effet les coûts de synchronisation peuvent alors devenir significatifs devant le gain de temps permis par la parallélisation.
Nous modifions l'implémentation GNU d'OpenMP afin qu'elle utilise PYTHIA pour décider du nombre de *threads* à utiliser pour chaque région parallèle OpenMP. Au début et à la fin de chaque région parallèle, GNU OpenMP soumet à PYTHIA-RECORD un événement permettant d'identifier la région parallèle. Au début de chaque région parallèle, le *runtime* interroge PYTHIA-PREDICT afin d'estimer le temps de calcul nécessaire pour l'exécuter. Cette durée est comparée à différentes valeurs seuils afin de déterminer le nombre de *threads* à utiliser. Nous modifions également OpenMP afin que les *threads* inutilisés soient mis en sommeil jusqu'à être réutilisés plutôt que détruits et recréés au besoin, ce qui est coûteux.

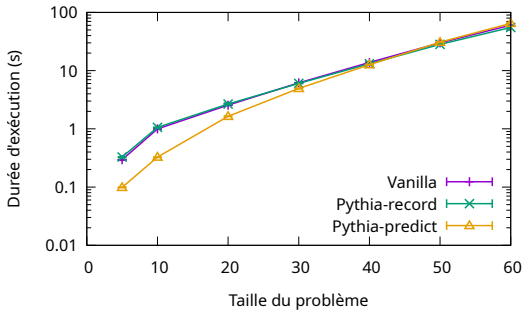
4. Évaluation

Nous évaluons PYTHIA et son utilité avec l'exemple décrit à la section 3. Nous présentons notre environnement expérimental dans la section 4.1, et dans la section 4.2, évaluons le surcoût de

FIGURE 4 – Durée d'exécution de Lulesh en fonction de la taille du problème.



(a) sur 3a401 avec 24 threads.



(b) sur b313 avec 16 threads.

PYTHIA-RECORD et les gains avec PYTHIA-PREDICT.

4.1. Environnement expérimental

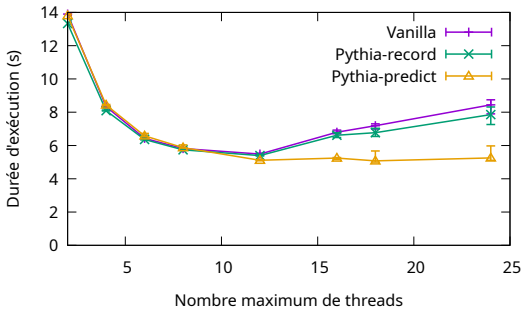
Nous effectuons nos expérimentations sur deux machines, « 3a401 »[1] et « b313 »[2], fonctionnant sous le noyaux Linux 5.4.0. Notre cas d'usage est le programme Lulesh en version 2.0, qui est une application OpenMP écrite en C++ qui résout un problème de Sedov et qui inclut trente régions parallèles de tailles variées [3]. Notre version de Lulesh est modifiée sur deux aspects. D'une part, nous ajoutons quelques appels à `omp_get_num_threads` afin que l'application tienne compte des variations du nombre de *threads* actifs. D'autre part, la stratégie d'allocation mémoire de Lulesh génère un grand nombre d'erreurs de pages mémoire. À notre surprise, enregistrer des événements avec PYTHIA-RECORD améliorait les performances de Lulesh de 15% en interférant avec ses allocations. Pour éliminer ce biais, nous modifions donc Lulesh afin qu'elle réutilise sa mémoire d'une étape de calcul à une autre.
Toutes nos expériences sont répétées dix fois, et nous indiquons les minimum, maximum et moyennes des temps d'exécution obtenus.

4.2. Évaluation des performances sur Lulesh

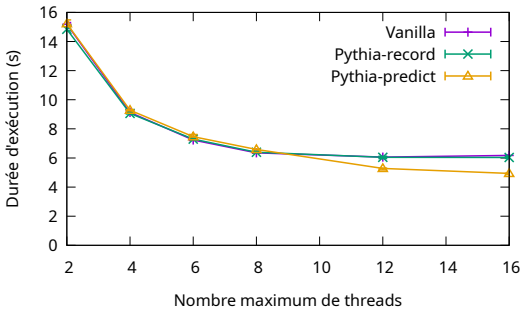
Les figures 4a et 4b montrent l'évolution de la durée d'exécution de Lulesh en fonction de la taille du problème à résoudre. Notez que l'axe Y est en échelle logarithmique. Les mesures montrent que PYTHIA-RECORD n'affecte pas significativement les performances de Lulesh. À l'inverse, PYTHIA-PREDICT améliore significativement le temps d'exécution, en particulier pour les problèmes de petites tailles. Le gain permis par PYTHIA réduit quand la taille du problème augmente. Ceci est dû au fait que les petites régions parallèles deviennent négligeables quand la quantité de données à traiter devient grande. Par conséquent, utiliser le nombre de *threads* maximal est optimal pour les grandes tailles de problèmes.
Afin de mieux comprendre l'impact de PYTHIA sur les performances de l'application, nous exécutons Lulesh avec un problème de taille 30 et nous faisons varier le nombre maximum de *threads* utilisables (Fig. 5). Lorsque le nombre de *threads* disponibles est faible, PYTHIA-PREDICT ne modifie pas la durée d'exécution de Lulesh. Quand le nombre de *threads* disponibles grandit, PYTHIA-PREDICT améliore les performances de Lulesh jusqu'à 38,8% et 20% sur 3a401 et b313, respectivement. En effet, PYTHIA-RECORD réduit le sur-coût lié aux coûts de synchronisation

1. Deux processeurs Intel Xeon Silver (12 coeurs et 24 *threads* chacun à 2,1GHz) avec 64Go de DRAM.
2. Deux processeurs Intel Xeon E5-2630 v3 (8 coeurs et 16 *threads* chacun à 2,4 GHz.) avec 32Go de DRAM

FIGURE 5 – Durée d'exécution de Lulesh en fonction du nombre maximal de threads.



(a) sur 3a401 pour un problème de taille 30.



(b) sur b313 pour un problème de taille 30.

croissants entre les *threads* sur les petites régions parallèles en maintenant le nombre de *threads* proche de son optimal sur ces régions.

5. État de l'art

Durant les dernières décennies, de nombreux travaux ont étudié la détection automatique de la structure d'une exécution. Différents travaux se concentrent sur la compression de données [15, 5] ou la compression de traces d'exécutions [8, 11]. Certains travaux appliqués à l'analyse et la recherche de bogues extraient la structure d'une trace à partir de ses redondances de manière exacte [13, 14, 12]. D'autres ajoutent une tolérance aux petites variations [6] ou exploitent conjointement d'autres sources d'information [9].
L'idée de représenter une trace par une grammaire a été étudiée dans différents travaux. Sequitur est un algorithme utilisé pour la compression de différents types de données sous forme de grammaire avec une complexité temporelle et spatiale linéaire [10]. Sequitur(1) est une adaptation de Sequitur utilisée pour détecter les chemins d'exécution des programmes [7]. Cyclitur étend Sequitur pour ajouter la notion de répétitions consécutives, et est appliqué à la détection d'anomalies dans des programmes embarqués [1].
Dans le domaine de la prédiction du comportement futur d'un programme, NLR infère les boucles imbriquées dans une trace d'exécution pour prédire les accès mémoire à venir [4]. Omnisc'IO utilise StarSequitur, un algorithme de réduction de grammaires similaire au nôtre pour prédire les prochaines entrées ou sorties d'un programme [2]. Cependant ces travaux sont spécialisés à la prédiction d'accès à des ressources spécifiques. À l'inverse, PYTHIA est un oracle générique qui peut être utilisé pour n'importe quel type d'optimisation.

6. Conclusion

Nous avons proposé PYTHIA, un oracle reposant sur la nature déterministe de nombreuses applications parallèles. Cet oracle compare l'exécution courante d'une application à une exécution précédente afin de prévoir le comportement futur du programme. Nous utilisons PYTHIA au sein de GNU OpenMP afin de prévoir la durée des régions OpenMP d'un programme. Grâce à ces prévisions, GNU OpenMP peut adapter dynamiquement le nombre de *threads* à utiliser en fonction de la charge de travail. Les expérimentations montrent que les prévisions de PYTHIA peuvent être utilisées pour améliorer significativement les performances d'une application. Cet exemple ouvre la voie à d'autres types d'optimisations au sein des *runtimes*.

Bibliographie

1. Amiar (A.), Delahaye (M.), Falcone (Y.) et Du Bousquet (L.). – Compressing microcontroller execution traces to assist system analysis. – In *International Embedded Systems Symposium*, pp. 139–150, 2013.
2. Dorier (M.), Ibrahim (S.), Antoniu (G.) et Ross (R.). – Omnisc'IO : a grammar-based approach to spatial and temporal I/O patterns prediction. – In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 623–634, 2014.
3. Karlin (I.), Keasler (J.) et Neely (J.). – *Lulesh 2.0 updates and changes*. – Rapport technique, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2013.
4. Ketterlin (A.) et Clauss (P.). – Prediction and trace compression of data access addresses through nested loop recognition. – In *Proc. of the IEEE/ACM international symposium on Code generation and optimization*, pp. 94–103, 2008.
5. Kieffer (J.-C.) et Yang (E.-h.). – Lossless data compression algorithms based on substitution tables. – In *Proc. of the IEEE Canadian Conference on Electrical and Computer Engineering*, pp. 629–632, 1998.
6. Knupfer (A.) et Nagel (W. E.). – Construction and compression of complete call graphs for post-mortem program trace analysis. – In *Proc. of the International Conference on Parallel Processing (ICPP)*, pp. 165–172, 2005.
7. Larus (J. R.). – Whole program paths. *ACM SIGPLAN Notices*, vol. 34, n5, 1999, pp. 259–269.
8. Milenkovic (A.) et Milenkovic (M.). – Stream-based trace compression. *IEEE Computer Architecture Letters*, vol. 2, n1, 2003, pp. 4–4.
9. Myers (D.), Storey (M.-A.) et Salois (M.). – Utilizing debug information to compact loops in large program traces. – In *Proc. of the European Conference on Software Maintenance and Reengineering*, pp. 41–50, 2010.
10. Nevill-Manning (C. G.) et Witten (I. H.). – Identifying hierarchical structure in sequences : A linear-time algorithm. *Journal of Artificial Intelligence Research*, vol. 7, 1997, pp. 67–82.
11. Noeth (M.), Ratn (P.), Mueller (F.), Schulz (M.) et De Supinski (B. R.). – ScalaTrace : Scalable compression and replay of communication traces for high-performance computing. *Journal of Parallel and Distributed Computing*, vol. 69, n8, 2009, pp. 696–710.
12. Taheri (S.), Briggs (I.), Burtscher (M.) et Gopalakrishnan (G.). – Difftrace : Efficient whole-program trace analysis and diffing for debugging. – In *Proc. of the IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 1–12, 2019.
13. Taniguchi (K.), Ishio (T.), Kamiya (T.), Kusumoto (S.) et Inoue (K.). – Extracting sequence diagram from execution trace of Java program. – In *Proc. of the International Workshop on Principles of Software Evolution (IWPSE)*, pp. 148–151, 2005.
14. Trahay (F.), Brunet (E.), Bouksiaa (M. M.) et Liao (J.). – Selecting points of interest in traces using patterns of events. – In *Proc. of the Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 70–77, 2015.
15. Yang (E.-H.) et Kieffer (J. C.). – Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform. I. Without context models. *IEEE Transactions on Information Theory*, vol. 46, n3, 2000, pp. 755–777.