Compas'2015 : Parallélisme/ Architecture / Système
Lille, France, du 30 juin au 3 juillet 2015

Détection automatique d'anomalies de performance

Mohamed Said Mosli Bouksiaa, François Trahay, Gaël Thomas

Télécom SudParis, Institut Mines-Télécom, CNRS UMR 5157 SAMOVAR, Université Paris-Saclay
{mohamed.mosli_bouksiaa,francois.trahay,gael.thomas}@telecom-sudparis.eu

---

Résumé
Le débogage des applications distribuées à large échelle ou encore des applications HPC est difficile. La tâche est encore plus compliquée quand il s'agit d'anomalies de performance. Les outils qui sont largement utilisés pour la détection de ces anomalies ne permettent pas d'en trouver les causes.
Dans cet article, nous présentons une approche basée sur l'analyse des traces d'exécution de programmes distribués. Notre approche permet de détecter des motifs récurrents dans les traces d'exécution et de les exploiter pour isoler les anomalies de performance. Les anomalies sont ensuite utilisées pour en trouver les causes.
Les résultats préliminaires montrent que nos algorithmes arrivent à détecter automatiquement de nombreuses anomalies et à les associer avec leurs causes.

Mots-clés : analyse de performance, traces d'exécution, anomalies de performance, détection de motifs

---

1. Introduction

Dans l'ère de l'informatique dans les nuages et du petascale, les systèmes de calcul subissent une évolution marquée par des architectures logicielles et matérielles de plus en plus complexes. Les systèmes distribués potentiellement hétérogènes avec des mémoires multi-niveaux soulèvent de nombreux défis : comment choisir le bon grain de distribution pour maximiser l'utilisation des machines ? Comment partitionner les données ? et bien d'autres questions émergent pendant la conception et le développement d'une application qui se doit de passer à l'échelle.
Cette situation concerne aussi bien les applications HPC que les applications distribuées à grande échelle. Dans ce contexte, une application devient hautement prédisposée aux problèmes de performance et son comportement devient difficile à déchiffrer.
Les outils de trace sont utilisés pour aider le développeur à comprendre le comportement de son application. Une trace d'exécution est un ensemble de fichiers générés à partir de l'exécution d'une application et qui contiennent, chacun, une séquence d'événements ayant lieu dans un même flux d'exécution (ou thread). Un événement représente généralement l'entrée ou la sortie d'une fonction. Chaque événement se compose d'un horodatage, d'un code unique décrivant le type de l'événement (nom de la fonction, etc) et d'une partie variable qui sert à conserver les paramètres de la fonction. Ces événements peuvent être analysés pendant ou après l'exécution de l'application pour tirer des conclusions sur le comportement du programme.

L'utilisation des traces pour la détection des anomalies de performance n'est pas nouvelle, mais la plupart des outils largement utilisés se contentent de localiser les anomalies de performance sans préciser leurs causes. Pourtant, dans les applications distribuées, chaque point de synchronisation est une opportunité pour que des anomalies de performance se propagent de processus en processus ou de thread en thread, ce qui les rend encore plus difficiles à comprendre et donc à corriger. Nous avons donc clairement besoin d'outils qui nous renseignent sur les sources des anomalies.

Dans cet article, nous proposons de détecter les motifs fréquents dans les traces d'exécution. Un motif est une séquence d'événements qui apparaît dans le même ordre plusieurs fois dans la trace. Les motifs trouvés sont filtrés pour ne garder que ceux qui peuvent indiquer la présence d'anomalies. Dans notre contexte, une anomalie de performance est une occurrence qui appartient à un motif et dont la durée est considérée trop longue. Les anomalies ainsi dévoilées sont analysées pour remonter jusqu'à leurs causes en cherchant leurs dépendances causales selon la relation *happened-before*.

Nous avons effectué une préévaluation de notre outil en analysant les traces de plusieurs applications issues des NAS Parallel Benchmarks (NPB). Ces traces ont été obtenues avec EZTrace et contiennent l'ensemble des appels aux fonctions MPI effectués par ces applications. La préévaluation est prometteuse et montre que :

- notre outil est capable d'identifier des centaines d'anomalies de performances.
- dans la plupart des traces étudiées, un certain nombre d'anomalies sources qui se propagent d'un thread à un autre sont trouvées par notre outil. Leur nombre varie selon l'application de quelques dizaines à quelques milliers.

Dans la Section 2, le processus de détection des anomalies est expliqué. Nous commençons par décrire l'algorithme permettant de trouver les motifs fréquents dans la trace. Ensuite, nous expliquons comment les anomalies sont séparées des occurrences normales. Enfin, nous présentons le processus de détection des sources d'anomalies. La Section 3 présente les résultats de l'évaluation de nos algorithmes. Tout d'abord, nous étudions les motifs, les anomalies et les sources d'anomalies détectées dans un ensemble de traces d'exécution. Ensuite, nous présentons un cas d'utilisation. Le Section 4 présente les travaux récents portant sur la même problématique. La Section 5 conclut l'article.

2. Détection des anomalies de performance

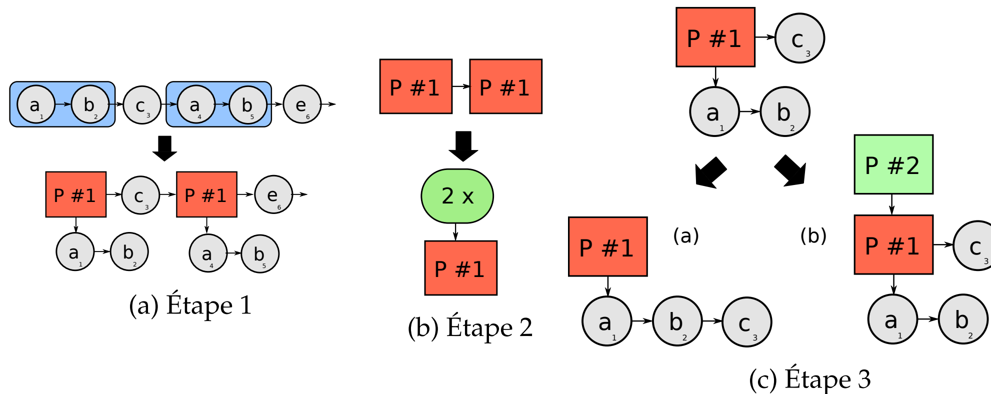
Cette Section présente le processus de détection des anomalies de performance dans l'exécution d'une application. Nous présentons les grandes lignes de l'algorithme utilisé pour la détection des motifs dans une trace d'exécution [11] dans la Section 2.1.

Une fois les motifs détectés, ces derniers sont classés selon les durées relatives de leurs occurrences en motifs réguliers et motifs irréguliers (Section 2.2). Les motifs réguliers permettent d'isoler les anomalies de performance.

Enfin, les séquences problématiques détectées sont utilisées pour remonter jusqu'aux sources de leurs anomalies. La détection des sources d'anomalies est présentée dans la Section 2.3. L'ensemble des tecnhiques décrites dans cette Section a été implémenté dans la plate-forme EZTrace [12].

2.1. Algorithme de détection des motifs

L'idée principale derrière l'algorithme permettant de détecter les motifs dans une trace est de commencer par les motifs de longueur 2, autrement dit les couples d'événements. Une fois un tel couple détecté, l'algorithme essaye de l'étendre en motifs plus longs. L'algorithme se



(a) Étape 1 (b) Étape 2 (c) Étape 3

FIGURE 1 – Les trois étapes de l'algorithme de détection de motifs

compose de trois étapes présentées dans la Figure 1 :

— La première étape (Figure 1a) consiste à trouver une séquence de deux événements (a,b) qui apparaît plusieurs fois dans la trace. Quand une telle paire est localisée, elle est remplacée par un motif p_1. Les spécificités de chaque occurrence du motif (les dates des événements, les paramètres des fonctions, etc.), sont gardées pour des étapes ultérieures de l'analyse.

— Une fois un motif p_1 repéré, la deuxième étape (Figure 1b) consiste à chercher les boucles composées de p_1, c'est à dire des occurrences consécutives de p_1, sans interruption.

— La troisième étape (Figure 1c a pour but d'étendre le motif p_1 en comparant entre eux les événements qui se trouvent après chaque occurrence de p_1. Si un même événement c se trouve après chaque occurrence de p_1 (cas (a)), c est attaché à la séquence d'événements qui définit p_1. Si plusieurs occurrences de p_1 (mais pas toutes) sont succédées par l'événement c (cas (b)), la définition de p_1 est gardée telle qu'elle est et un nouveau motif p_2 est défini qui correspond à (p_1,c). Enfin si c succède à p_1 dans une seule occurrence, le motif p_1 ne peut pas être étendu.

Les étapes 2 et 3 sont répétées tant qu'elles produisent des modifications sur la hiérarchie de la trace en trouvant de nouvelles boucles ou de nouvelles extensions de motifs. L'algorithme utilise ensuite l'étape 1 pour trouver un autre motif et répéter l'opération.

2.2. Détection des occurrences problématiques

L'algorithme décrit dans la Section 2.1 permet de détecter un grand nombre de motifs. L'étape suivante est de chercher des anomalies de performance en étudiant les variations dans les durées des occurrences de chaque motif. Certains motifs trouvés ne sont pas pertinents pour la détection des anomalies. Ces motifs ont soit peu d'occurrences, ce qui ne permet pas d'extraire des statistiques significatives, soit une distribution de durées trop irrégulière. Il est donc nécessaire de vérifier, pour chaque motif, si ses occurrences peuvent être considérées pour détecter des anomalies. Cette opération s'appelle la caractérisation des motifs. Tout d'abord, nous présentons notre méthode de classification des occurrences d'un motif en 3 groupes : occurrences de référence, occurrences normales et occurrences problématiques. Ensuite, nous expliquons comment séparer ces 3 groupes par le biais de 2 paramètres.

2.2.1. Les 3 groupes d'occurrences

Pour qu'un motif puisse être sélectionné, il faut être capable de classer ses occurrences distinctement en 3 groupes :

- Le groupe de référence : contient les occurrences les plus courtes. Ce groupe doit contenir la "majorité" des occurrences. Concrètement, il faut déterminer un pourcentage (voir Section 2.2.2)
- Le groupe normal : contient les occurrences "un peu" plus longues que les occurrences de référence. De même, un paramètre est utilisé (Section 2.2.2) pour fixer le degré de tolérance aux occurrences longues.
- Le groupe problématique : contient les occurrences "trop" longues. Ces séquences sont considérées problématiques et doivent être examinées en priorité.

2.2.2. Paramètres s et q : seuil d'anomalie et taille de l'ensemble de référence

La classification des occurrences d'un motif dépend de deux paramètres. Pour définir la taille du groupe de référence contenant la "majorité" des occurrences, nous utilisons un paramètre q. q est le quotient du nombre d'occurrences dans le groupe de référence par le nombre total des occurrences du motif. Si $q = 75\%$, cela signifie que pour qu'il contienne la "majorité" des occurrences, le groupe de référence doit contenir les trois-quarts des occurrences.

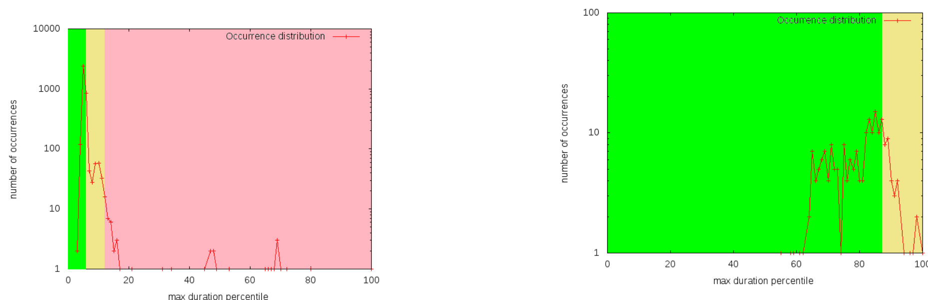
Pour définir les occurrences "trop" longues, on utilise un paramètre s qui désigne le seuil au delà duquel une occurrence est considérée comme anormale. Ce paramètre s est appelé le seuil d'anomalie. Une occurrence est normale si sa durée est inférieure à $s \times max_ref$ et problématique dans le cas contraire, max_ref étant la borne supérieure du groupe de référence. Autrement dit, si $s \times max_ref > 100\%$, on ne peut plus obtenir d'occurrences problématiques et le motif devient inutile. En conséquence, la deuxième contrainte sur le groupe de référence est qu'il s'étende sur un intervalle de durées $[0..max_ref]$ assez petit.

La Figure 2a présente un exemple de l'utilisation de ces 2 paramètres. La Figure montre la distribution des occurrences d'un motif en fonction de leurs durées. Le groupe de référence, représenté par l'espace vert, s'étend sur $[0..max_ref]$. Le groupe normal, représenté par l'espace jaune s'étend sur $[max_ref..s \times max_ref]$. Enfin, l'espace rouge représente le groupe problématique. Une fois une séquence appartenant à ce groupe détectée, elle est marquée d'un *drapeau* (*flag*) indiquant son statut problématique.

Les valeurs choisies pour ces deux paramètres affectent la sélection éventuelle d'un motif comme le montre la Figure 2b. Pour ce motif, les valeurs $q = 75\%$ et $s = 2$ ne permettent pas de détecter des anomalies. Des valeurs élevées de q et s permettent de détecter les anomalies flagrantes et ne retournent qu'une poignée d'occurrences problématiques. À l'inverse, des petites valeurs permettent de détecter des variations plus subtiles dans les durées des occurrences au risque de retourner plus de faux positifs. Dans sa forme actuelle, notre approche est plutôt semi-automatique, dans le sens où l'appréciation des résultats de l'analyse par l'utilisateur est requise afin de les valider. Par conséquent, nous pensons que le choix de ces deux paramètres est plus judicieux quand il est fait en fonction de l'application et du besoin de son développeur. Ainsi, on peut commencer par des valeurs relativement hautes pour corriger les problèmes qui affectent le plus la performance de l'application avant de passer aux problèmes qui ont moins d'impact.

2.3. Détection des sources d'anomalies

Une fois les séquences d'événements ayant une durée anormalement grande identifiées (voir Section 2.2), nous nous proposons dans cette Section d'identifier les sources de ces anomalies de façon automatique. En effet, une anomalie se produisant sur un thread peut se propager



(a) Séparation de l'ensemble des occurrences en trois intervalles : occurrences de référence (Vert), occurrences normales (Jaune) et occurrences anormales (Rouge)

(b) Exemple d'un motif ne permettant pas de trouver des anomalies : $q = 75\%$ incite à prendre tout l'intervalle [0..88%] comme groupe de référence. $s = 2$ ne permet de trouver aucune anomalie

FIGURE 2 – Le rôle des paramètres s et q dans la sélection des motifs intéressants

à d'autres threads à travers l'échange d'un message par exemple. Inversement, une anomalie initialement détectée sur un thread peut être due à un message qui met beaucoup de temps à arriver parce que le thread émetteur rencontre une anomalie à son tour.

Tout d'abord, il faut choisir un modèle causal, ou plus simplement, déterminer quels sont les modes de propagation d'anomalies à considérer. Nous avons choisi de chercher les relations causales entre problèmes de performance qui sont dues aux communications par messages. La propagation d'une anomalie peut aussi être due à d'autres moyens : sémaphore, condition, signaux, etc. Bien que ces moyens ne soient pas considérés dans cet article, cela n'affecte pas l'applicabilité de notre méthode à ces cas.

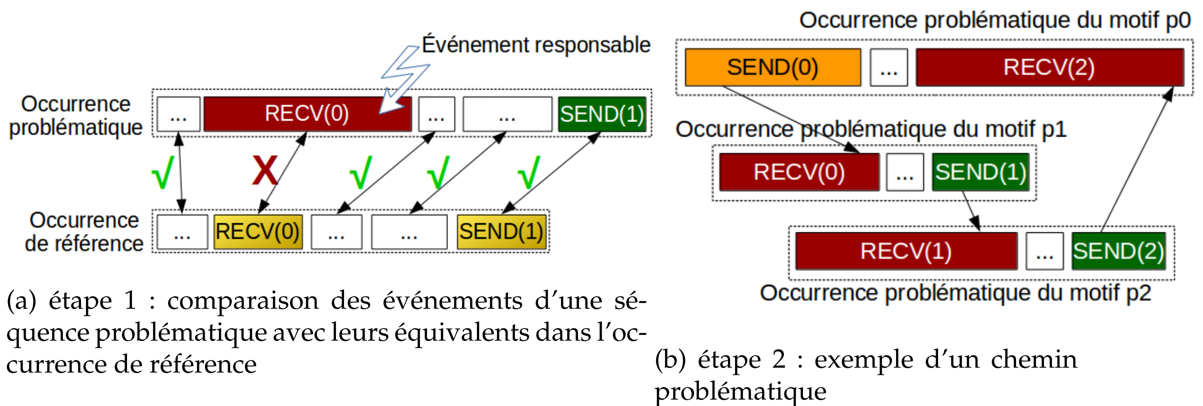
Le méchanisme de détection des sources d'anomalies se compose de deux étapes. La première étape permet, à partir d'une séquence problématique préalablement détectée, de trouver l'événement précis responsable de l'anomalie au sein de la séquence.

Si ce dernier est un événement de synchronisation entre threads (dans notre contexte, une réception de message), il se peut que l'anomalie provienne d'un autre thread. L'objet de la deuxième étape est de vérifier cette éventualité. Pour ce faire, il faut examiner les autres séquences d'événements ayant lieu sur d'autres threads mais qui sont liées par le biais de l'événement de réception à la séquence en cours d'examination.

Les deux étapes sont répétées jusqu'à ce que l'algorithme de détection des sources d'anomalies tombe sur une séquence normale ou trouve que l'événement responsable de l'anomalie en cours d'examination n'est pas un événement de réception.

Pour effectuer la première étape, *i.e.* pour trouver l'événement responsable de l'anomalie au sein d'une séquence, il est nécessaire de parcourir la séquence, événement par événement, en comparant la durée de chaque événement à celle de son homologue dans la séquence de référence appartenant au même motif (Figure 3a). Le même seuil d'anomalie s qui sert à comparer les durées de deux séquences d'événements entières dans 2.2 est utilisé. La séquence de référence est sélectionnée lors de la phase de caractérisation des motifs parmi les occurrences appartenant au groupe de référence.

La deuxième étape permet de construire un chemin problématique (Figure 4). Chaque noeud du chemin correspond à une anomalie. Dans un noeud, on enregistre l'identifiant du thread sur lequel l'anomalie a été découverte ainsi que l'événement précis qui en est responsable. Quand deux noeuds sont liés par une flèche, cela signifie que le noeud prédécesseur est une



(a) étape 1 : comparaison des événements d'une séquence problématique avec leurs équivalents dans l'occurrence de référence

(b) étape 2 : exemple d'un chemin problématique

FIGURE 3 – Processus de détection des anomalies

anomalie qui comporte un événement d'envoi (**SEND**) et que l'événement de réception (**RECV**) correspondant est responsable de l'anomalie dans le noeud successeur.

L'ensemble des chemins problématiques constitue le graphe problématique. A chaque fois qu'une relation de causalité entre deux anomalies est découverte (Figure 4), le noeud père (l'anomalie contenant le **SEND**) est d'abord recherché dans le graphe problématique. S'il y est déjà, cela signifie que l'anomalie correspondante a déjà été analysée, que l'événement responsable est connu et que les éventuelles sources ont déjà été déterminées. Dans ce cas, le chemin est attaché au bon nœud dans le graphe. Dans le cas contraire, la construction du chemin est poursuivie en appliquant la première étape (recherche de l'événement responsable) au noeud père.

La Figure 3b donne un exemple de chemin problématique : les rectangles en pointillés délimitent des occurrences problématiques. À l'intérieur de chacune d'elles sont représentés : en vert les événements de durée normale, en rouge les événements de durée anormale qui correspondent à une réception de message et en orange les événements dont la durée est trop longue mais qui ne comportent pas de réception de message.

Dans cet exemple, l'occurrence problématique du motif p0 contient deux événements problématiques. Le premier événement problématique, **SEND(0)**, est la source du chemin problématique. Son anomalie est propagée vers l'occurrence du motif p1 à travers le message d'id 0 qui arrive tardivement. Au sein de cette même séquence, l'événement **SEND(1)** est normal mais a été déclenché en retard à cause de **RECV(0)** ce qui prolonge la durée de **RECV(1)** dans l'occurrence du motif p2. Enfin, dans cette troisième séquence, l'événement **SEND(2)** qui a lieu en retard, provoque un long **RECV(2)** dans l'occurrence du motif p0. Ce dernier événement marque la fin du chemin problématique.

3. Évaluation

Dans cette Section, nous évaluons tout d'abord les performances de l'algorithme de détection de motifs et nous étudions les motifs, les anomalies et les sources d'anomalies détectées dans un ensemble de traces d'exécution. Nous présentons ensuite un cas d'utilisation. Ces travaux étant encore en cours de réalisation, les résultats présentés sont préliminaires.

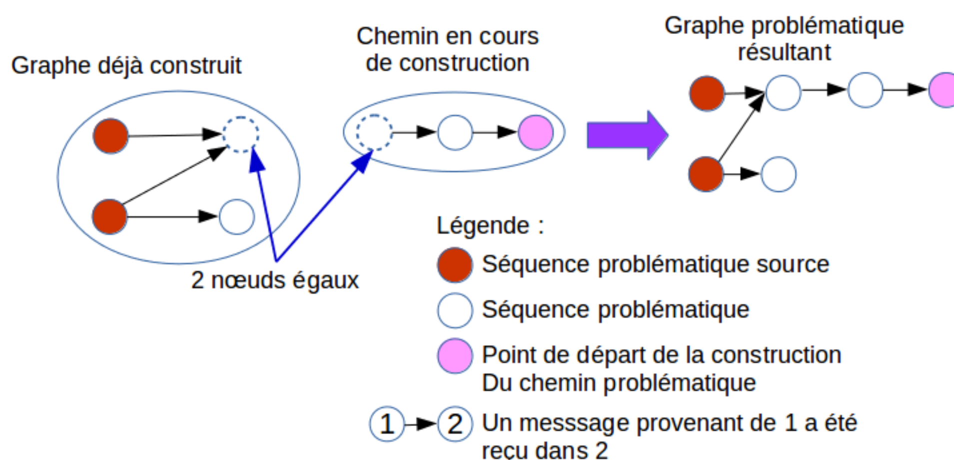


FIGURE 4 – Construction du graphe problématique

3.1. Évaluation des algorithmes proposés

Afin d'évaluler les algorithmes proposés, nous étudions tout d'abord le résultat de l'analyse des traces de plusieurs applications issues des NAS Parallel Benchmarks et s'exécutant sur 16 processus MPI. Ces traces ont été obtenues avec EZTrace et contiennent l'ensemble des appels aux fonctions MPI effectués par ces applications. Les résultats de ces analyses sont reportés dans la Table 1.

Les résultats montrent que la durée d'exécution de l'algorithme permettant de détecter des motifs dans les traces dépend du nombre d'événements contenus dans les traces. Le temps nécessaire à la détection de motifs reste toutefois inférieur à quelques secondes pour les plus grosses traces. Pour chacune des traces étudiées, plusieurs dizaines de motifs sont détectés. La phase de caractérisation des motifs (présentée dans la Section 2.2) permet d'éliminer les motifs trop irréguliers pour être analysés. Après caractérisation, seule une petite partie (entre 18 % et 59 % des motifs pour les traces de taille importante) est sélectionnée pour être analysée.

La détection d'anomalies, décrite dans la Section 2.2.2, est appliquée aux motifs sélectionnés avec un seuil d'anomalie de 2. Plusieurs centaines d'occurences anormales de motifs sont détectées à cette phase de l'analyse.

La colonne # *sources* présente le nombre de séquences présentant une anomalie et se propageant à d'autres séquences. La colonne l_{max} présente la longueur maximale des chemins problématiques pour chaque trace. Ces résultats montrent que dans la plupart des traces étudiées, un nombre restreint de séquences anormales se propagent d'un thread à un autre. De plus, une séquence anormale peut affecter un grand nombre de séquences ayant lieu sur d'autres threads. Par exemple, un chemin problématique comportant 15 messages MPI a été détecté dans la trace de l'application BT. L'existence de tels chemins est un argument en faveur de l'utilité de notre algorithme étant donné qu'ils sont extrêmement compliqués voire même impossibles à déceler manuellement

3.2. Cas d'utilisation : l'application CG

Afin d'évaluer les algorithmes proposés sur des traces d'applications réelles, nous présentons dans cette Section un cas d'usage s'appuyant sur une trace de l'application CG (issue des NAS Parallel Benchmark) s'exécutant sur 32 processus. Alors que la trace comporte 5 785 871 évé-

kernel	*#événements*	*détection (ms)*	*#motifs*	*#motifs sélectionnés*	*#problèmes*	*#sources*	l_{max}
EP	3 090	3.51	32	31 (97%)	32	0	1
FT	10 256	9.44	80	19 (24%)	19	0	1
IS	18 552	35.43	48	0	–	–	–
CG	284 754	178.23	160	65 (40%)	611	34	3
MG	118 688	186.17	2 728	494 (18%)	5 357	350	2
SP	557 318	596.84	174	48 (28%)	2 132	0	1
BT	399 944	951.51	112	66 (59%)	1 367	159	15
LU	4 568 002	4 564.80	210	101 (48%)	13 721	2372	6

TABLE 1 – Statistiques sur la recherche de problèmes de performance dans les traces NPB (Classe=A, NProcs=16) - seuil d'anomalie s=2

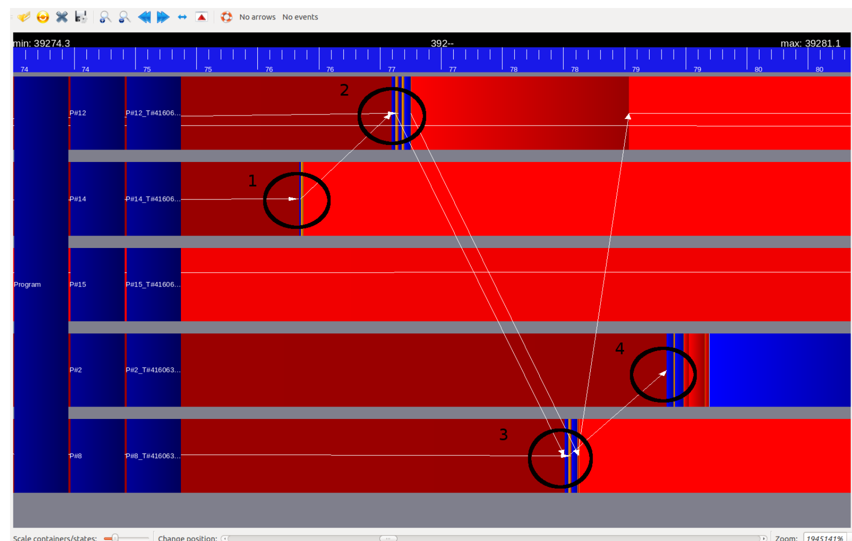


FIGURE 5 – Trace visualisée avec ViTE montrant un chemin problématique

nements répartis sur 106 secondes d'exécution, l'algorithme de détection d'anomalies sélectionne une fenêtre de 219 ms répartie sur 5 processus. Une représentation graphique de la fenêtre d'évènements sélectionnée est présentée à la Figure 5. La Figure représente les événements ayant lieu à partir du moment où un message problématique a été reçu par le processus P#14. Cette Figure montre que l'attente anormale du processus P#14 a des répercussions sur d'autres processus. Ainsi, le processus P#12 attend un message de P#14. Ce message n'est envoyé qu'après la réception par P#14 (cercle 1) et P#12 reçoit donc le message en retard (cerce 2). Le retard pris par P#12 est ensuite propagé au processus P#8 (cercle 3), puis P#2 (cercle 4).
Les algorithmes proposés permettent donc de repérer de manière automatique certaines anomalies dans des traces d'exécution ainsi que leurs répercussions. Cela permet d'indiquer à l'utilisateur quelles parties de la trace sont susceptibles de contenir des informations intéressantes. Pour le cas d'usage présenté ici, l'utilisateur peut se concentrer sur quelques dizaines d'événements au lieu de devoir parcourir manuellement une trace constituée de plusieurs millions d'événements.

4. État de l'art

Le diagnostic automatique de problèmes de performance suscite de plus en plus l'intérêt de différentes communautés que ce soit en parallélisme ou en systèmes, ce qui a conduit à la production de multiples outils qui offrent certaines solutions à cette problématique.
Par exemple, Scalasca [6] est un ensemble d'outils axé sur la scalabilité et qui produit des rapports détaillés aux développeurs d'applications distribuées les aidant dans l'analyse de performance surtout pour les applications comportant un grand nombre de processus. Les états d'attente sont identifiés, des motifs sont détectés et classés par catégories et enfin l'impact des différents problèmes identifiés est quantifié. La plate-forme appelée "The Bottleneck Discovery Framework" [4] permet de découvrir des régions du code qui comportent des goulets d'étranglement. Les goulets d'étranglement sont reconnus grâce à une base de données contenant des motifs spécifiques. Contrairement à ces deux outils, notre approche ne nécessite pas une connaissance spécifique des problèmes recherchés.
Plusieurs travaux [2, 5, 8, 10, 13] sont d'accord sur le fait que la simple localisation d'un problème n'est pas toujours d'une grande utilité pour le corriger vu que ce dernier n'est souvent pas indépendant et qu'il faudrait localiser les éléments fortement liés à ce problème et qui ont contribué à son apparition. Par exemple, les traces sont analysées selon un schéma de causalité qui s'appuie sur les états d'attente entre les composants [13]. Ceci permet notamment de distinguer le temps d'attente accumulé sur tout le chemin causal et qui est dû soit à des appels de fonctions soit à l'attente de libération d'un verrou. Par ailleurs, des techniques de statistiques et de machine learning sont parfois utilisées [10] pour déterminer quels sont les composants dont la présence s'accompagne le plus avec les mauvaises performances observées et rapportées par des utilisateurs. Le but est de localiser, dans le code, l'origine du problème observé. Pour ce faire, de nombreuses entrées différentes sont testées et leurs sorties sont classées selon leur bonne ou mauvaise performance ; ensuite sont déterminés les "indicateurs" (le fait de prendre une branche conditionnelle en particulier ou qu'une fonction retourne une certaine valeur) de bonnes ou mauvaises performances selon des modèles statistiques. Une autre méthode est d'identifier le chemin problématique grâce à la technique de *taint tracking* [9] en suivant le flux de données et le flux de contrôle à travers l'application [2]. Le framework Scalasca [8] a aussi intégré cette perspective en ajoutant l'analyse des causes racines des états d'attente à l'ensemble de ses fonctionnalités. La recherche des causes racines s'appuie entre autres sur la sommation des délais d'attente tout au long des chemins des communications. La plupart de ces outils cherchent une cause racine qui a une forme bien particulière (un indicateur spécifique [10]) ou qui se trouve dans un endroit spécifique (fichiers de configuration [2]). En revanche, notre technique n'a pas besoin de définir une forme particulière pour les sources d'anomalies vu qu'elles sont cherchées parmi les anomalies.

5. Conclusions

L'analyse automatique de grandes traces d'exécution est un problème important. Dans cet article, nous proposons un ensemble d'algorithmes permettant d'analyser des traces d'exécution pour y détecter certaines anomalies de performance. Un premier algorithme détecte les séquences d'événements se répétant et, en comparant les variations de la durée de ces séquences, repère les parties anormalement longues de la trace. Les séquences anormales sont ensuite étudiées afin de détecter la propagation d'anomalies d'un thread de l'application à un autre. Les évaluations des algorithmes proposés, implémentés dans EZTrace, montrent qu'ils permettent de détecter la propagation d'anomalies parmi des traces contenant plusieurs millions d'événé-

ments et facilite donc l'analyse de performance pour les utilisateurs.
Afin de compléter ces travaux préliminaires, nous comptons étudier des traces issues de domaines applicatifs variés (systèmes distribués, calcul hautes performances, etc.) Nous souhaitons également affiner les heuristiques utilisées pour caractériser les séquences présentant des anomalies. Par ailleurs, nous envisageons d'estimer les faux-positifs et les faux-négatifs détectés par notre algorithme afin de mieux évaluer sa précision.

Bibliographie

1. Aguilera (M. K.), Mogul (J. C.), Wiener (J. L.), Reynolds (P.) et Muthitacharoen (A.). – Performance debugging for distributed systems of black boxes. – In *ACM SIGOPS Operating Systems Review*, pp. 74–89. ACM, 2003.
2. Attariyan (M.), Chow (M.) et Flinn (J.). – X-ray : Automating root-cause diagnosis of performance anomalies in production software. – In *OSDI*, pp. 307–320, 2012.
3. Chow (M.), Meisner (D.), Flinn (J.), Peek (D.) et Wenisch (T. F.). – The mystery machine : End-to-end performance analysis of large-scale internet services. – In *Proceedings of the 11th symposium on Operating Systems Design and Implementation*, 2014.
4. Chung (I.-H.), Cong (G.), Klepacki (D.), Sbaraglia (S.), Seelam (S.) et Wen (H.-F.). – A framework for automated performance bottleneck detection. – In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–7. IEEE, 2008.
5. Diwan (A.), Hauswirth (M.), Mytkowicz (T.) et Sweeney (P. F.). – Traceanalyzer : a system for processing performance traces. *Software : Practice and Experience*, vol. 41, n3, 2011, pp. 267–282.
6. Geimer (M.), Wolf (F.), Wylie (B. J.), Ábrahám (E.), Becker (D.) et Mohr (B.). – The scalasca performance toolset architecture. *Concurrency and Computation : Practice and Experience*, vol. 22, n6, 2010, pp. 702–719.
7. Lagraa (S.), Termier (A.) et Pétrot (F.). – Scalability bottlenecks discovery in mpsoc platforms using data mining on simulation traces. – In *Proceedings of the conference on Design, Automation & Test in Europe*, p. 186. European Design and Automation Association, 2014.
8. Lorenz (D.), Böhme (D.), Mohr (B.), Strube (A.) et Szebenyi (Z.). – Extending scalasca's analysis features. *In : Tools for High Performance Computing 2012*, pp. 115–126. – Springer, 2013.
9. Newsome (J.) et Song (D.). – Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. – In *NDSS*, 2005.
10. Song (L.) et Lu (S.). – Statistical debugging for real-world performance problems. – In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pp. 561–578. ACM, 2014.
11. Trahay (F.), Brunet (E.), Mosli Bouksiaa (M.) et Jianwei (L.). – Selecting points of interest in traces using patterns of events. – In *23rd Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2015.
12. Trahay (F.), Ishikawa (Y.), Rue (F.), Namyst (R.), Faverge (M.) et Dongarra (J.). – Eztrace : a generic framework for performance analysis. – In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pp. 618–619. IEEE, 2011.
13. Yu (X.), Han (S.), Zhang (D.) et Xie (T.). – Comprehending performance from real-world execution traces : A device-driver case. – In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pp. 193–206. ACM, 2014.