ANNEE **2022/2023**

RAPPORT de "MISSION en ENTREPRISE"

présenté par :

Catherine GUELQUE

Mission effectuée de 01/04/2023 au 01/09/2023 chez

TELECOM SUDPARIS

Sujet de la mission :

Hierarchical Trace Format: Analyse de performance pour l'exascale

Directeur de stage : **Francois TRAHAY**

Conseiller de Stage : **Elisabeth BRUNET**

Travail effectué pour la société : Télécom SudParis, 9 Place Marguerite Perey

TELECOM SudParis

ANNEE 2022/2023

Nom de l'étudiant : Catherine GUELQUE

Option : ASR

Entreprise : TELECOM SUDPARIS

Date : 01/04/2023 - 01/09/2023

Sujet : Hierarchical Trace Format: Analyse de performance pour l'exascale

Résumé :

Le domaine du Calcul Haute Performance (HPC) est un domaine en pleine évolution technique. L'arrivée de nouveaux paradigmes, tels que l'utilisation d'accélérateurs (GPU), ou le calcul par tâche (StarPU) demandent des adaptions des outils déjà existant. De plus, l'arrivée de centres de calculs exaflopiques, c'est à dire de *clusters* capables de réaliser plus d'un milliard de milliard de calculs flottants par seconde, risque de rendre certains outils obsolètes.
Parmi ces outils, les traces, des enregistrements des exécutions des programmes, permettant des analyses post-mortem de ces derniers, sont utiles pour améliorer le passage à l'échelle des applications. Cependant, la plupart des outils de traces existant ne sont pas adaptés à ces nouveaux paradigmes, et au calcul exaflopique.
C'est dans ce contexte que nous proposons un nouveau format de trace : `HTF` (Hierarchichal Trace Format). Ce format propose une forte scalabilité, avec des systèmes de compression avancés. Il offre des performances proches d'autres outils récents, tels que Pilgrim, et propose des analyses avancées, permettant un passage à l'échelle exaflopique, et simples à calculer.

Abstract :

The field of High Performance Computing (HPC) is a rapidly evolving domain. The emergence of new paradigms, such as the use of accelerators (GPUs), or task-based computing (StarPU), requires adjustments to existing tools. Furthermore, the advent of exaflop-capable computing facilities, which are *clusters* capable of performing more than a billion billion floating-point calculations per second, may render certain tools obsolete.
One kind of such tools are traces, records of program executions, which allow for post-mortem analysis of these executions. They are quite valuable for enhancing the scalability of applications. However, most of the existing trace tools are not suited for these new paradigms and exascale computing.
It is within this context that we propose a new trace format : `HTF`, which stands for Hierarchical Trace Format. This format offers strong scalability, incorporating advanced encoding and compression systems. It delivers performance comparable to other recent tools, such as Pilgrim, and provides sophisticated analyses that facilitate exascale scalability and straightforward calculations.

Table des matières

Remerciements

J'aimerai tout d'abord remercier François Trahay, pour m'avoir offert l'opportunité de faire ce stage, ainsi que pour son encadrement et sa bienveillance tout au long de ce stage. Je remercie également Valentin Honoré, qui m'a tout autant encadré pendant ces cinq mois.

Je remercie toute l'équipe du département INF de Télécom SudParis, et en particulier Pierre Pollet, Jean-François Dummolard, Jana Toljaga, Suba Tanigassalame, Adam Chader, Mickaël Boichot, Yohan Piperau et Marie Reinbigler, qui m'ont accueilli à bras ouvert dans leur équipe, et avec qui j'ai pu partager de grands moments de réflexion autour de plusieurs tasses de café et quelques tasses de thé.

Merci à toutes les personnes que j'ai pu rencontrer pendant les conférences et séminaires auxquels j'ai assisté pendant ce stage, qui ont pu me fournir des informations précieuses et utiles au cours de discussions souvent trop courtes.

Merci à Chloé Hennequin et Prométhée Toneatti pour leur aide lors de la relecture de ce rapport.

1 Introduction

De nombreux domaines scientifiques ont des besoins en calcul importants. Que ce soit en météorologie, en astrophysique, en biologie, en chimie, ou en ingénierie, la demande pour des capacités de calculs de plus en plus avancées ne cesse de grandir. Face à la complexité de ces calculs, et pour répondre à cette demande, un nouveau domaine de l'informatique s'est développé : le Calcul Haute Performance, ou HPC (*High Performance Computing*).

Le HPC est une discipline informatique qui vise à résoudre des problèmes complexes et gourmands en ressources de calcul en utilisant des ordinateurs extrêmement puissants. Ces systèmes, appelés supercalculateurs, sont conçus pour exécuter des calculs intensifs à des vitesses bien supérieures à celles des ordinateurs conventionnels. Ces machines possèdent également des capacités de stockage de données conséquentes. La plupart des supercalculateurs modernes sont en réalité des clusters, c'est-à-dire des groupes de machines. Pour les exploiter, il est nécessaire de répartir les calculs sur ces multiples ordinateurs. De plus, les paradigmes de calcul des supercalculateurs ne cessent d'évoluer, et les avancées les plus récentes utilisent de nouvelles technologies telles que le calcul sur processeur graphique (GPU). Ces machines ont des besoins spécifiques, que ce soit des besoins matériels (machines à multiples cœurs, clusters interconnectés, accélérateurs...) ou bien logiciels (OpenMP pour le calcul sur plusieurs threads, MPI pour le calcul sur plusieurs machines, CUDA pour le calcul sur GPU...).

La France a répondu à un appel d'offre à l'échelle européenne, et a donc été sélectionnée pour héberger l'un des deux centres de calculs exaflopiques européens d'ici 2025. C'est dans ce cadre qu'a été lancé le PEPR [1] NumPEx [2], piloté par le CEA, le CNRS et l'INRIA, dont le but est de développer la pile logicielle qui sera utilisée sur ce supercalculateur d'un nouveau type.

Le développement de logiciels HPC performants présente de nombreux défis, tels que la répartition des calculs sur les différentes machines et threads, ou encore le partage des ressources sur ces différentes machines. Par exemple, de multiples machines accédant à la même ressource vont causer des problèmes de contention, ralentissant grandement l'exécution du programme.

Il existe plusieurs outils conçus pour aider le développement et l'optimisation de programmes HPC. Dans le cadre de ce travail, nous nous intéressons surtout aux traces, qui sont des enregistrements détaillés des événements et activités qui se produisent pendant l'exécution de ces programmes. Ces traces peuvent ensuite être analysées ou visualisées, afin de permettre une meilleure optimisation de ces programmes. Cependant, l'utilisation d'outils générant ces traces ralentissent l'exécution du programme : c'est ce que l'on appelle le surcoût, qui est généralement limité à un ordre de grandeur de 5% du temps d'exécution normal du programme. Nonobstant, les outils de traçage et formats de trace existant actuellement ne sont pas adaptés au calcul exaflopique. En effet, la plupart des formats utilisés produiraient des traces beaucoup trop grosses, et le surcoût associé au traçage de ces programmes pourrait être trop élevé.

Le but de ce stage est donc de proposer un outil de traçage destiné au passage à l'échelle exaflopique, ainsi qu'un format de trace qui permette une analyse efficace des traces d'applications générées.

1. Projets et Équipements Prioriaires pour la Recherche
2. Numérique Pour l'Exascale

2 Contexte

Cette section du rapport se consacre à l'examen approfondi de l'état de l'art dans le domaine de l'enregistrement et de l'analyse des traces HPC, en mettant l'accent sur les limitations actuelles des formats de trace existants et en explorant les besoins spécifiques au calcul exaflopique. Nous présentons également les avancées récentes dans le domaine, telles que les initiatives visant à développer des formats de trace plus flexibles, complets et adaptés aux architectures modernes. En comprenant les lacunes et les opportunités actuelles, nous posons les bases nécessaires pour la proposition et le développement d'un nouveau format de trace novateur, capable de capturer efficacement les nuances complexes des interactions au sein des systèmes HPC de prochaine génération.

Il existe deux types d'outils d'analyse de performance pour les applications HPC : les profilers et les traces. Ces traces sont générées par des outils de traçage, qui utilisent une grande variété de formats. Elles sont ensuite analysées par des outils d'analyse de traces.

2.1 Profilers

Le principe d'un profiler est de récupérer des données de performance d'un programme. Pour cela, les profilers collectent des informations statistiques sur le programme, tels que des appels à des fonctions, le temps passé dans certaines fonctions, etc., sans enregistrer de traces détaillées de l'exécution. Il existe pour cela deux méthodes : la première consiste à intercepter des appels à des fonctions spécifiques, ce qui permet un faible surcoût mais limite l'analyse à certaines fonctions prédéfinies ; la deuxième consiste à régulièrement prendre des échantillons du programme, en l'interrompant à des intervalles de temps constants. L'avantage de cette seconde méthode est une possible meilleure précision pour les analyses statistiques, dépendant du nombre d'appels de fonctions et du temps passé dans ces fonctions, et surtout une précision quantifiable par l'utilisateur, puisque sa précision est gouvernée par sa fréquence d'échantillonage.

Parmi les multiples outils de profiling, le plus répandu est `perf` [1], le profiler par défaut de Linux, qui utilise la seconde méthode détaillée plus haut. Il s'agit d'un profiler mono-processus, c'est-à-dire qu'il ne peut pas analyser correctement des programmes utilisant MPI par exemple. C'est également le cas de `gProf` [2], également la méthode d'échantillonage statistique, ou de `oProfile` [3], qui propose de nombreuses analyses sur le déroulement de l'exécution.

Il existe également des profilers spécifiquement conçus pour le HPC, tels que `mpiP` [4], qui utilise la première méthode en interceptant des appels aux fonctions de la bibliothèque standard MPI, ou `Caliper` [5], un profiler conçu pour les applications utilisant OpenMP.

Ainsi, les profilers permettent d'obtenir des informations sur le fonctionnement interne d'un programme. En résumant une exécution complète à des analyses statistiques, les profilers ont cet avantage de fournir ces informations à faible coût, autant en terme de surcoût temporel qu'en terme de coût de stockage. Cependant, ce résumé limite les analyses plus détaillées d'un programme. Il est en effet compliqué d'identifier des problèmes d'interactions entre les threads sans avoir une vision plus détaillée de l'exécution (tel qu'une attente dans un MPI_Recv à cause du retard d'envoi d'un autre processus, comme dans la Figure 1).

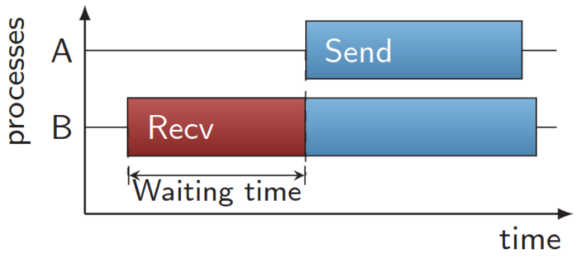


FIGURE 1 – Exemple de ralentissement causé par une mauvaise interaction des processus. Comme le processus A a commencé son appel à MPI_Send trop tardivement, le processus B passe du temps à attendre dans le MPI_Recv

2.2 Outils de traçage

En parallèle des profilers, il existe des outils capable de générer un enregistrement complet d'une exécution. Un tel enregistrement est appelé une **trace**, et nous nous focalisons par la suite sur l'étude des différents outils et formats de trace existant pour le HPC. La Figure 2 montre la visualisation d'une trace d'exécution d'un programme sur plusieurs threads.

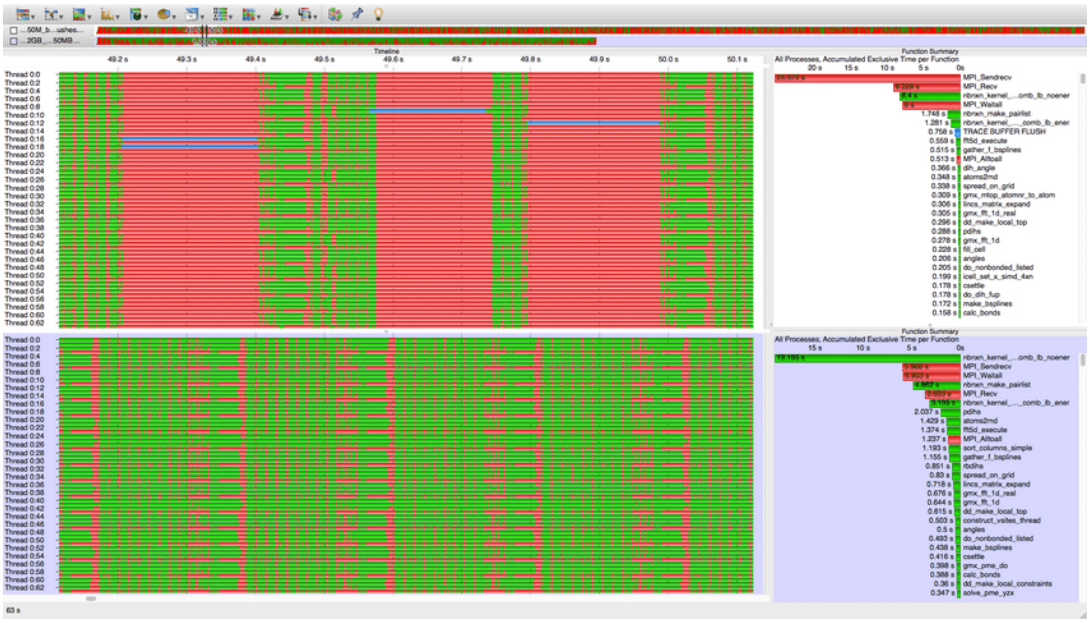


FIGURE 2 – Trace au format OTF2 visualisée par `Vampir`. On peut voir les différents threads et leur état au cours de l'exécution à gauche, et le temps passé dans différentes fonctions à droite.

Les outils de traçage opèrent d'une manière similaire aux profilers, dans la mesure où ils interceptent les appels à des fonctions ou des callbacks pré-définies (habituellement les fonctions issues des bibliothèques standard HPC telles qu'OpenMP, MPI, CUDA ou StarPU), ainsi que des appels système. Mais, contrairement à un profiler, les outils de traçage vont également récolter d'autres informations, et sauvegarder ces interceptions sous la forme d'un ou plusieurs évènements horodatés. Ces évènements contiennent de multiples informations utiles à l'analyse, en plus de l'horodatage, tel que des paramètres pour certaines fonctions. Cette interception vise à élaborer une séquence chronologique

de l'évolution du programme, ce qui permet une analyse détaillée à a fin de l'exécution. Les traces générées sont ensuite enregistrées sous la forme d'un ou plusieurs fichiers, contenant différentes parties de la trace, selon le format de trace utilisé.

Il existe plusieurs outils de traçage, utilisant des formats différents, et offrant des fonctionnalités variées. Par exemple, `EZTrace` [6] génère des traces au format OTF2 [7] et permet de tracer des programmes utilisant un nombre exhaustif de bibliothèques de calcul parallèle. D'autre outils fonctionnent en écosystème, tel que Scalasca [8], qui fournit un outil pour enregistrer une trace en utilisant différents formats (OTF2, TAU, CUBE4), ainsi que plusieurs outils d'analyse. Scalasca rentre dans l'écosystème d'outil de traçages et d'analyse de trace nommé Score-P [9], dont le but est de regrouper différents formats et outils de traçage pour simplifier les analyses.

Scalatrace [10] est un outil de tracage dont le but est de détecter la structure d'un programme, plutôt que de récolter des informations sur les évènements, et ne stocke donc pas les timestamps. Il utilise ensuite ces informations pour générer des rapports rapides et concis sur l'exécution. Il se situe donc à mi-chemin entre un profiler et un outil de traçage. Cypress [11] est un outil de traçage utilisant une analyse à la compilation pour prévoir le comportement du programme, et ainsi diminuer le surcoût de traçage à l'exécution. Pilgrim [12][13] est un autre outil ayant pour but d'offrir une compression efficace et quasi-sans perte des traces.

Les outils de traçages permettent donc de récolter beaucoup plus d'informations que les profilers, mais la collecte de données exhaustives entraîne divers problèmes : surcoût plus élevé, traces pouvant être lourdes, synchronisation entre les différents threads... Tout cela varie également en fonction du format de trace utilisé.

2.3 Formats de traces

Il existe une multitude de formats de trace, mais tous peuvent être classés dans deux grandes familles : les format générant des traces séquentielles, et ceux générant des traces structurelles.

2.3.1 Format de traces séquentiels

Le principe d'une trace séquentielle est d'enregistrer ou de lire tous les évènements les uns à la suite des autres. Un évènement correspond à une interception de fonction, un callback, ou n'importe quel point d'intérêt détecté pendant l'exécution du programme. Il est associé à un timestamp, ainsi qu'à une identification, et parfois, à plusieurs attributs.

C'est le cas de formats tels qu'OTF2 [7], qui enregistre chaque évènement avec son timestamp et ses arguments, sans se soucier de si cet évènement était déjà arrivé auparavant. La structure d'un évènement au format OTF2 est présentée dans la Figure 3. D'autres formats utilisent des systèmes similaires, tels que FxT [14] (qui possède encore plus d'informations par évènement, notamment s'il s'agit d'un évènement du noyau du système ou d'un évènement en espace utilisateur), ou le Trace Event Format [15], un format basé sur le JSON et utilisé par le visualisateur de trace de Chromium, mais qui n'est pas adapté aux traces de HPC, car trop volumineuses. Pajé [16] est également un exemple de trace lisible par un humain, utilisant un principe hiérarchique pour faciliter la lecture. LiTL [17] est un format de trace isolant chacune des sous-traces correspondant à chaque thread d'exécution. Il est alors possible d'analyser un seul fil d'exécution, ou de combiner plusieurs traces pour les analyser en parallèle.

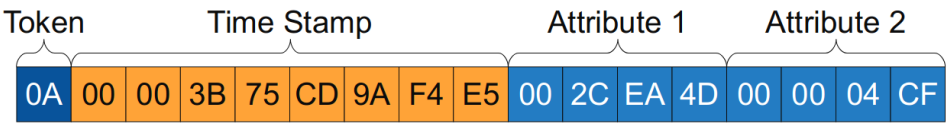


FIGURE 3 – Un évènement en OTF2

Certains formats de trace séquentiel, comme FxT, utilisent un seul buffer pour tous les threads d'exécution, ce qui nécessite une synchronisation. Cette méthode a pour avantage une lecture de la trace immédiate, puisque tous les évènements à travers tous les threads sont enregistrés de manière linéaire ; cependant, une écriture dans un buffer synchronisée sur un grand nombre de threads provoque des problèmes de contention, qui ralentissent naturellement l'exécution.

Les formats de traces séquentiels offrent donc plusieurs avantages tels que leur facilité d'implémentation, et leur surcoût à l'écriture faible. Il suffit d'intercepter un évènement et de le rajouter dans un buffer, puis d'écrire ce buffer dans un fichier, et cela est suffisant. Ces traces sont également simples à lire, puisqu'il suffit essentiellement de parcourir un tableau pour obtenir la frise chronologique de l'exécution. Les formats de ce type sont également simples à développer et à comprendre, expliquant leur popularité. Néanmoins, les tailles de ce type de traces croissent linéairement avec le nombre d'évènements enregistrés (si le format ne prend pas en compte de compression). Il est cependant possible de modifier ces formats de trace, pour permettre l'utilisation de techniques de compressions efficaces [18], mais cette implémentation nécessite de rajouter des étapes de décompression et de réorganisation des données à la lecture.

2.3.2 Détection de la structure d'un programme

Plusieurs travaux se concentrent sur l'extraction de la structure des traces, que ce soit lors de l'exécution du programme, ou de manière post-mortem. En effet, les applications utilisées en HPC sont généralement très régulières [19], avec des structures simples, contenant des boucles se répétant un grand nombre de fois. La détection de cette structure permet de faire des analyses statistiques plus rapidement, simplifie la lecture de la trace par un être humain, et permet une meilleure compression de la trace.

Il est possible de détecter la structure d'un programme post-mortem, en analysant une trace. C'est ce qui est présenté dans un travail par Trahay et al. [20], qui propose un algorithme en deux étapes pour définir la structure d'une trace. La première étape consiste à trouver des motifs récurrents dans la trace, et la deuxième à regrouper ces motifs récurrents adjacents en des structures de boucle, comme illustré dans la Figure 4. Cette détection est cependant unique à chaque thread d'exécution, et doit donc être répétée pour chaque thread, alors même qu'ils possèdent probablement une structure similaire.

La plupart des implémentions utilisent un système de détection de motifs récurrents à l'exécution, qui est appliqué à chaque fois qu'un nouvel évènement est rajouté à la trace [10][13]. La Figure 5 montre un exemple de cet algorithme en action. L'algorithme est généralement tronqué, pour éviter les complexités quasi-polynomiales quand la taille de la trace devient trop grande. Cela empêche donc de détecter des motifs trop longs. L'inconvénient principal de ce genre de trace est le surcoût associé à la détection de la structure, qu'il se situe au moment de l'exécution ou bien post-mortem.

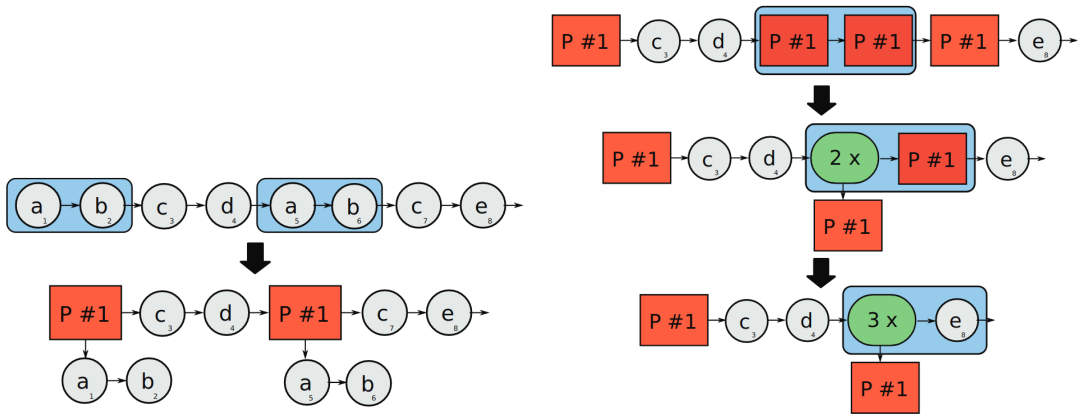


FIGURE 4 – Détection de la structure d'une trace en deux étapes, suivant l'algorithme proposé par Trahay et al. [20].

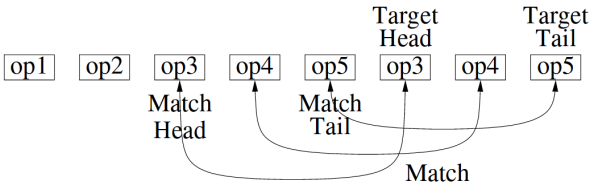


FIGURE 5 – Détection d'une répétition de la séquence 3-4-5.

Pilgrim [12] utilise un algorithme de détection de motifs un peu différent, puisqu'il se base sur algorithme nommé *non-sequitur*, qui utilise un système de grammaire appelé *Context Free Grammar* (CFG), qui permet de factoriser une suite d'évènements en une multitude de motifs présents dans la CFG, de manière récursive. Ainsi, en utilisant cet algorithme, la séquence `xyzxyz` serait successivement simplifiée en `xAxA` avec `A = yz`, puis en `BB`, avec `B = xA`, puis simplement en `2*B`. Cela permet une compression très efficace de la trace, mais ne permet pas de récupérer la structure de la trace à proprement parler. Les grammaires ainsi générées sont uniques à chaque thread d'exécution, mais sont fusionnées à la fin de l'exécution pour ne plus former qu'une seule grammaire unique. La Figure 6 présente les différentes étapes de cette fusion, qui commence par plusieurs passages de fusion des grammaires, et qui se fini par un passage dit "sequitur", permettant d'éliminer les doublons de la grammaire.

L'outil CYPRESS [11](qui fait office de format de trace et d'outil de traçage) propose une détection de la structure du programme à la compilation, en analysant son code source. Cette analyse sera ensuite utilisée au moment de l'exécution par le système pour permettre une détection de la structure, avec des performances similaires à des traces séquentielles, tout en ne laissant qu'un surcoût minimum à la compilation et à l'exécution.

Pythia [19] est un outil utilisant cette détection de structure pour prédire les ressources demandées par les applications. Ces prédictions sont ensuite utilisées pour guider le runtime, notamment en implémentant une stratégie de parallélisation adaptative pour OpenMP. Les améliorations de performance en utilisant Pythia peuvent atteindre jusqu'à 38% de réduction du temps d'exécution.

Les traces implémentant une détection de la structure présentent plusieurs avantages, notamment une grande versatilité. En effet, comme les timestamps sont stockés à la fin des évènements, il est très simple de les supprimer (pour ne garder que la structure de

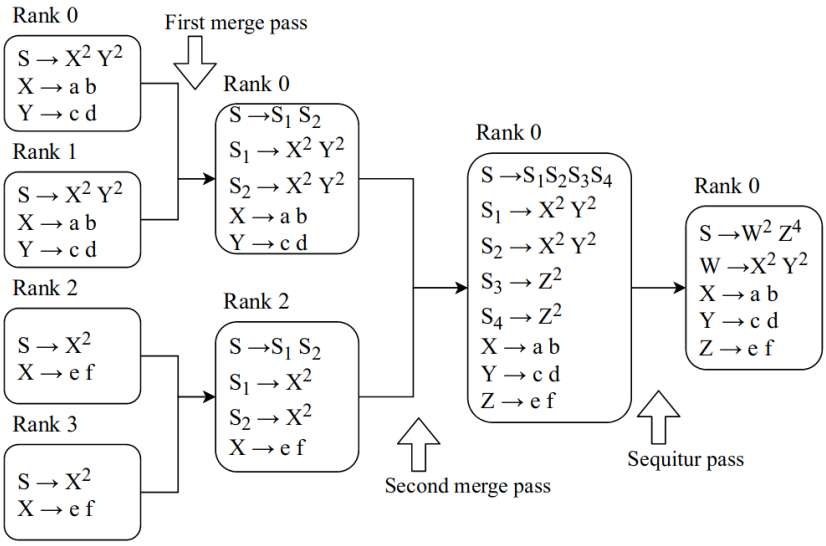


FIGURE 6 – Algorithme de fusion des CFG de Pilgrim.

la trace, qui ne pèse alors que quelques dizaines de Ko), ou bien de les compresser, en utilisant différentes méthodes. C'est par exemple le cas de Pilgrim, qui propose diverses méthodes de compression, avec ou sans pertes.

Cependant, les travaux utilisant cette méthode sont centrés voir spécifiques à MPI, et ne permettent donc pas d'extraire toutes les informations intéressantes du programme. La lecture de la trace par la machine est également plus complexe, puisqu'il ne s'agit plus seulement de lire un simple tableau. Enfin, l'espace gagné par la détection de la structure peut en fait être perdu si le programme est très irrégulier [3]. Les traces ainsi générées sont très linéaires, et le rajout du détail de la structure, en plus du surcoût de son calcul rendent le traçage moins performant qu'une simple trace séquentielle. C'est par exemple le cas sur les applications Quicksilver ou AMG, que nous utilisons par la suite en tant que benchmarks.

2.4 Gestion des données et compression des timestamps

Lors de l'écriture de la trace, diverses données vont être collectées sur les évènements, tels que leur timestamp, leur durée, les arguments dans le cas d'appels de fonction, et même dans certains cas des méta-données sur l'évènement. Ces données forment la plus grosse partie du poids de la trace, et c'est pourquoi il est important de les gérer de manière efficace. Comme nous l'avons déjà vu précédemment, la plupart des traces ont un système de gestion des données par évènement, c'est-à-dire que chaque évènement est associé à des données (un timestamp, un temps d'exécution, des arguments pour les appels à fonctions, des méta-données sur l'évènement) récoltées pendant l'exécution.

Cependant, certains formats de trace à détection de structure utilisent une méthode différente en stockant ces données par type d'évènement. Cela permet de compresser ces données de manière très efficace, notamment grâce à leur similarité. Dans le cas de Pilgrim [12], une méthode de compression à perte bâtie sur cette propriété de similarité

3. Un programme dit régulier ne présente que peu ou pas d'aléatoire, ainsi qu'une répartition très homogène des calculs sur les différents threads et processus, alors qu'un programme irrégulier est plutôt imprévisible

a d'ailleurs été proposée, avec de très bons résultats. En revanche, il est à noter que cette propriété n'est pas vraie pour toutes les données. En effet, s'il semble raisonnable de supposer que des évènements similaires aient la même durée, cela devient plus compliqué pour les arguments et méta-données, et complètement faux pour les timestamps. Cependant, si la structure de la trace est connue, les timestamps deviennent beaucoup moins utiles que les durées, puisqu'ils peuvent être déduits à partir de ces derniers. Dans le cadre de ce travail, nous ne nous intéresserons qu'à la gestion des timestamps et des durées, puisque la compression des autres types de données n'est que peu abordée dans les papiers du domaine.

L'outil de traçage de Pilgrim implémente de multiples méthodes de compressions différentes, parmi lesquelles ZSTD, un algorithme de compression sans perte [21] , SZ [22] et ZFP [23], deux algorithmes de compression à perte conçus spécifiquement pour la compression de données issues de traces de programmes HPC, ainsi que deux nouvelles méthodes proposées par les auteurs. La première se base sur un regroupement des durées des évènements par intervalles exponentiels, tandis que la seconde utilise des intervalles correspondant à une courbe gaussienne (respectivement notées CFG et HIST dans la Figure 7).

Afin de comparer l'efficacité de ces différents algorithmes de compression, nous avons mesuré la taille des traces générées par Pilgrim sur différents programmes classiques de HPC avec trois niveaux de complexité de calcul croissants pour chaque programme, ainsi que la taille de la trace si l'on n'effectue aucune compression (notée LOSSLESS dans la Figure 7). Les résultats ainsi obtenus sont présentés dans la Figure 7, et montrent qu'une bonne gestion de ces données peut entraîner une réduction de la taille de la trace générée par un facteur allant jusqu'à plusieurs ordres de grandeur en utilisant des méthodes à perte. La perte de précision sur les données récoltées n'est cependant pas forcément un problème dans le cas des durées, notamment si cette perte est inférieure à une certaine limite que fixée. C'est ce que propose d'ailleurs SZ, ainsi que, dans une moindre mesure, les algorithmes proposés par Pilgrim.

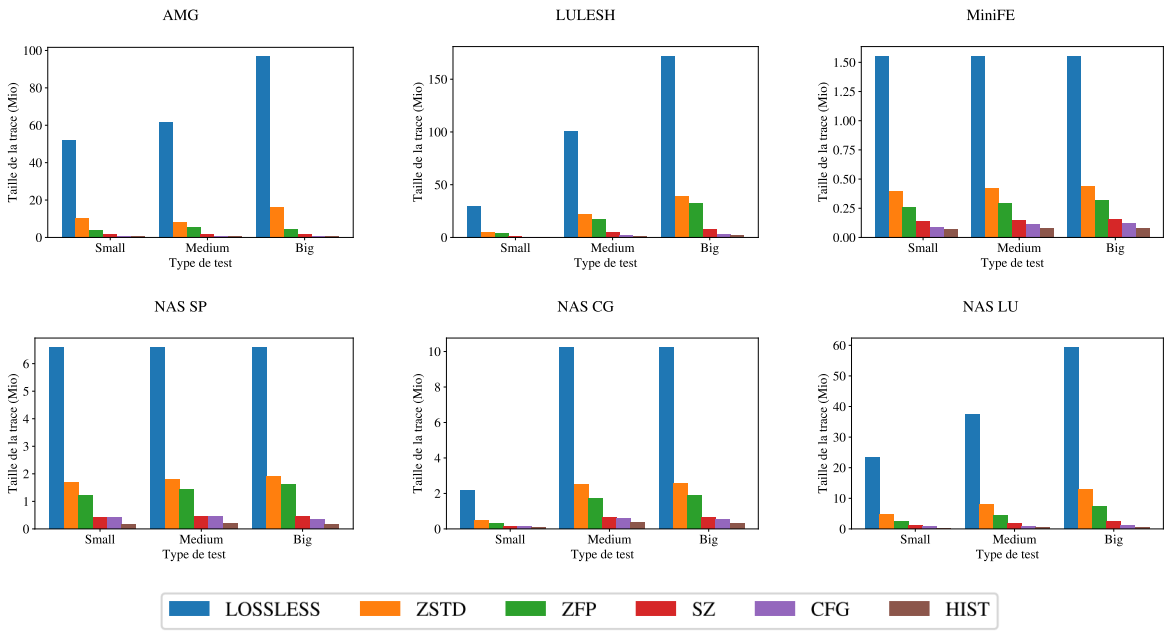


FIGURE 7 – Tailles des traces obtenues pour différentes méthodes de compression sur différents benchmarks, en utilisant Pilgrim (en Mébioctets).

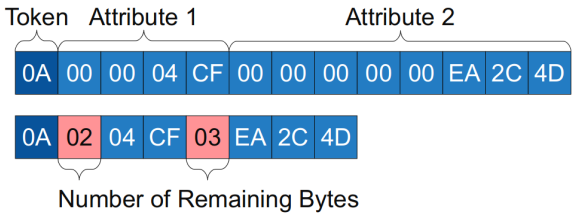


FIGURE 8 – Proposition de Wagner et al. [18] d'optimisation de l'écriture de différents attributs.

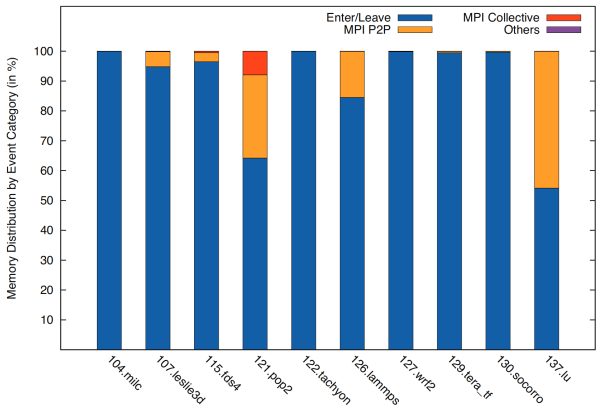


FIGURE 9 – Répartition des différents types d'évènements dans une trace OTF2 selon différentes applications.

Un travail d'amélioration du format OTF2 [18] propose également un encodage relativement efficace et peu coûteux pour tous les types de données. Cette méthode se base sur l'observation que plusieurs données sont stockées sur plus d'octets que nécessaire. Les tableaux les regroupant se retrouvent ainsi avec plusieurs octets nuls. La Figure 8 illustre comment ces octets peuvent être remplacés par un seul octet, indiquant le nombre d'octets stockant les données. D'après cet article, les timestamps étaient ainsi réduits d'une taille de 8 à 4 octets seulement, à la condition de ne pas les stocker en tant que timestamps, mais plutôt en tant que différentiel de timestamps, ce qui rajoute donc des calculs à l'écriture et à la lecture de la trace, dont le surcoût n'a pas été mesuré.

Une dernière optimisation proposée par Wagner et al. a été de faire une étude statistique afin de trouver les types d'évènements les plus communs, dont la réduction de taille aurait un bien plus grand impact sur la taille de la trace [18]. Les résultats de cette analyse statistique, présentés dans la Figure 9, montrent que les évènements *Enter/Leave* constituent une écrasante majorité des évènements présents dans une trace, suivi des évènements MPI P2P, tels que les MPI_Send ou MPI_Receive. L'optimisation proposée quant à cette propriété des traces n'est cependant spécifique qu'au format OTF2, et sera donc moins utile que l'optimisation proposée précédemment.

La gestion des timestamps est donc un élément à considérer dans le développement d'une trace prévue pour l'Exascale, et la méthode de compression de ceux-ci également. L'utilisation d'une trace structurelle, permettant de faire abstraction de ces timestamps à l'écriture ou à la lecture, améliorerait la scalabilité de notre format de trace. De même, une bonne compression des timestamps permettraient de réduire la taille de notre trace de plusieurs ordres de grandeur. Il faut cependant noter que cela diminue la simplicité

d'une trace, et que la compression rajoute un temps de calcul à l'écriture et à la lecture de la trace. De plus, toutes les compressions ne se font pas sans pertes, et il faudra quantifier la perte maximale acceptable par notre format de trace.

2.5 Analyses des traces

De multiples outils offrent la possibilité d'exploiter les traces collectées pour réaliser des analyses approfondies, ainsi que pour créer des visualisations qui servent à l'optimisation du programme. La représentation d'une trace sous la forme d'un diagramme de Gantt permet par exemple de visualiser les différentes dépendances dans le code, et permet donc d'optimiser le fonctionnement du programme. De même, des solutions telles que Scalasca [8] intègrent des capacités d'analyse de traces, ce qui permet d'identifier des problèmes liés aux conflits de ressources, aux ralentissements inhabituels, et plus encore. Le logiciel StarVZ [24] utilise des visualisations innovantes (notamment des animations) pour afficher des informations concernant les ressources de calcul et les tâches lors de l'exécution d'un programme utilisant StarPU.

Tous les formats de trace présentés dans ce rapport offrent une possibilité, d'une manière ou d'une autre, de visualiser une frise chronologique de l'exécution du programme tracé, en utilisant différents supports. Par exemple, le format Trace Event Format peut être visualisé à l'aide de Catapult [25], présent par défaut dans Chromium (dont un aperçu est présenté Figure 10). Il existe plusieurs outils capable de générer des visualisations pour les traces au format OTF2, tels que Vampir [26] ou ViTE [27]. Le format Pajé peut être visualisé par StarVZ [24] ou PajéNJ [16]. Il existe une multitude de visualiseurs de traces pour tous les formats utilisés.

FIGURE 10 – Une trace au format TFE, visualisé avec Catapult dans Chromium. Des noms explicites sont donnés à chaque thread (à gauche), et une frise chronologique détaillée des évènements interceptés est visible à droite.

Ces différents outils de visualisations possèdent leurs fonctionnalités propres, qui permettent de faire des analyses plus ou moins poussées. Par exemple, Catapult permet de faire des statistiques concernant des évènements similaires à la volée, ainsi que de nommer différents threads et processus, créer des groupes d'évènements, et réaliser des mesures de durée. La visualisation de traces de Vampir ne se contente pas d'être une simple frise chronologique, elle fournit également des informations sur les communications point à

point, indique des points d'intérêts (rafales de messages, utilisation de fonctions d'entrée/sortie), et permet l'utilisation d'outils d'analyses poussées tels que des statistiques sur les mesures récoltées, des calculs de métriques définies par l'utilisateur, des mesures des FLOP au cours de l'exécution, ou des informations sur l'utilisation de la mémoire selon les différents threads.

Il est également possible de détecter des zones de contention, c'est-à-dire les moments de l'exécution où plusieurs processus demandent l'accès à des mêmes ressources, et plus généralement, les interférences entre les différents processus [28].

Ce genre d'outil possède néanmoins un problème de performance inhérent. En effet, pour afficher toute une trace et faire des analyses sur cette dernière, il est nécessaire de la charger entièrement en mémoire. De plus, plus une trace se fait grosse, et plus cela prendra du temps pour les outils de visualisation et d'analyses pour s'exécuter. Cela crée ainsi un deuxième problème : le temps de traitement de l'analyse de la trace. Enfin, sur des traces longues et générées sur de plus gros clusters, c'est-à-dire possédant une grande quantité de processus, la visualisation de trace peut ne pas être lisible. Il devient alors compliqué de comprendre la structure du programme rapidement.

Il existe cependant certaines analyses qui peuvent être réalisées uniquement avec la structure de la trace, telles que les matrices de communications proposées par Pilgrim, présentées Figure 11. De telles matrices indiquent la fréquence de communication entre les divers nœuds du cluster MPI, ce qui peut permettre de détecter une mauvaise répartition des communications entre les différents nœuds. Une case clair indique un axe de communication beaucoup plus sollicité que les autres, alors qu'une case sombre indique au contraire un axe plutôt négligé.

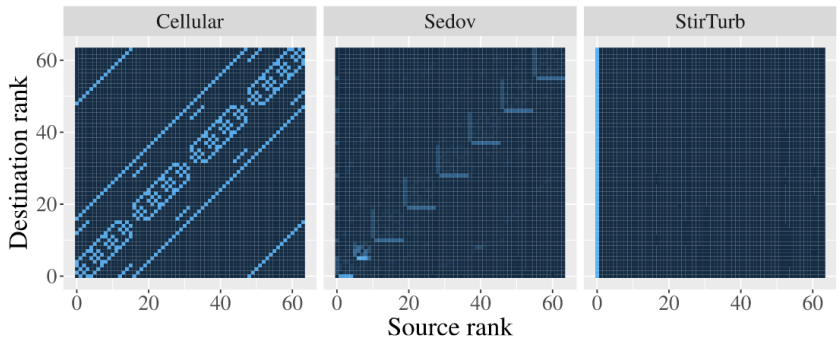


FIGURE 11 – Matrices de communication générées par Pilgrim [13] pour différentes version de FLASH sur 64 processus.

Les outils d'analyse de traces ont ainsi plusieurs avantages : en présentant une reconstruction de l'exécution post-mortem, ils permettent à l'utilisateur de vérifier l'optimisation de son programme et de détecter différentes zones de contention. Des outils d'analyses plus complexes permettent d'obtenir des informations encore plus fines. Cependant, ces analyses peuvent être coûteuses, et ne sont pas toutes adaptées à des traces de grande taille. En effet, un diagramme de Gantt sur plusieurs milliers de threads serait par exemple très peu lisible. Les analyses reposant sur l'utilisation des timestamps peuvent être également très coûteuse en mémoire et en temps sur les traces de trop grande taille, si elles ne sont pas implémentées de manière à ne sélectionner que les informations intéressantes. Il est donc important de penser à cela en développant notre format de trace, ainsi que les outils d'analyse dépendant de ce format de trace.

2.6 Conclusion

Il existe une grande variété d'outils permettant d'analyser les comportements des applications de calcul réparti, chacun ayant ses propres caractéristiques, avantages et inconvénients : les profilers permettent d'obtenir des informations très globales à faible coût, mais échouent à fournir des informations détaillées sur le déroulement de l'exécution. Les traces, quant à elles, sont très utiles pour pouvoir précisément analyser le déroulé d'une application en post-mortem, mais peuvent s'avérer de plus en plus complexes à utiliser lorsque le nombre de processus et la durée d'exécution augmentent. D'une part, des exécutions de longue durée génèrent des traces exceptionnellement lourdes à cause du poids des timestamps. D'autre part, des exécutions sur plusieurs milliers de threads généreraient des traces lourdes à cause d'une forte redondance d'informations pour chaque thread.

De plus, la plupart de ces outils se limitent à l'analyse d'applications MPI, alors même que le calcul sur GPU prend une place de plus en plus importante dans le domaine du HPC. De même, le calcul réparti par tâche, comme le propose StarPU, semble devenir une partie importante de la pile logicielle du prochain calculateur exaflopique européen. Il est donc important d'incorporer ces deux paradigmes dans notre format de trace.

Dans ce contexte, il devient nécessaire de développer un nouveau format de trace, conçu en pensant avant tout à l'Exascale. Ce format devra proposer une compression efficace, un surcoût minimal, des possibilités d'analyses rapides, l'interception d'une grande quantité d'évènements, ainsi qu'une détection de la structure du programme afin de faciliter son analyse.

3 Un nouveau format de trace pour l'Exascale

Dans cette section, nous présentons `HTF` (Hierarchical Trace Format), un format de trace conçu et pensé pour le calcul exaflopique. `HTF` est un format générique, contenant la structure du programme, avec des données récoltées pendant l'exécution facilement compressibles, et accessibles indépendemment les unes des autres.

La structure de la trace restera toujours suffisamment légère pour pouvoir la charger et ensuite l'utiliser pour charger dynamiquement les données d'exécution. Cela permettra de présenter diverses analyses réalisées sur la trace à faible coût. Afin de rendre le format de trace indépendant de l'outil de traçage, le format de trace que nous proposons permet de stocker des événements quelconques. De plus, nous avons utilisé des techniques de compressions inspirées des différents travaux déjà présentés pour permettre de réduire fortement la taille de nos traces. Enfin, nous avons développé deux outils d'analyse pour cet format de trace, tirant tous les deux partis des spécificités de notre proposition.

L'idée principale de notre proposition est de représenter une trace sous la forme d'une structure de données telle que présentée dans la Figure 12. Les **Évènements** (c'est-à-dire les interceptions de fonctions) sont identifiés par des **tokens** (dans la Figure 12, l'évènement correspondant à la création du thread principal est ainsi représenté par le token "E0"), et les **Séquences** regroupent plusieurs tokens. Une Séquence se répétant sera représentée par une **Boucle**. Les Séquences et les Boucles sont également représentées par des tokens, ce qui permet une imbrication de Séquences et de Boucles.

À chaque interception d'un Évènement, un token correspondant à cet Évènement est ajouté à un buffer, et un timestamp est généré. Ce dernier est utilisé pour calculer la durée de l'Évènement intercepté précédemment. Cela permet de calculer les durées des évènements au cours de l'exécution, et non pas à sa fin. Ces timestamps sont ensuite stockés dans un tableau pour chaque type d'Évènement, ce qui facilitera leur compression. À chaque ajout d'un token dans le buffer, un algorithme de détection de motifs est exécuté, et une Boucle (composée de deux Séquences) peut être générée. Dans ce cas, les durées de ces Séquence sont calculées, et sont également stockées dans un tableau, de manière similaire aux Évènements.

Chaque thread d'exécution possède sa propre structure, contenant des informations qui lui sont propre, et donc son propre buffer. De même, chaque processus possède une structure propre, avec des informations le concernant. C'est à partir de ce fonctionnement, similaire à la détection de structure présentée dans la Figure 4, que nous rajoutons ensuite un algorithme de détection de la structure, et que nous mettons en place une compression des timestamps.

3.1 Détection de la structure

Notre algorithme de détection de la structure du programme est similaire à celui proposé par ScalaTrace [10], Pilgrim [13] ou le travail de Trahay et al. sur la détection de structure de trace en post-mortem [20], mais en rajoutant une étape de hiérarchisation, ce qui simplifie les calculs de reconnaissance de motifs.

L'algorithme distingue 3 types d'évènements : les débuts de blocs (tels que l'entrée dans une fonction, la création d'un thread, etc.), les fins de blocs, et les évènements ponctuels (tels que la réception d'un message MPI). Nous utilisons ensuite ces blocs pour créer différents niveaux hiérarchiques, représentés par des Séquences. Chaque niveau possède son propre buffer dans lequel les Évènements interceptés sont rajoutés, ce qui nous

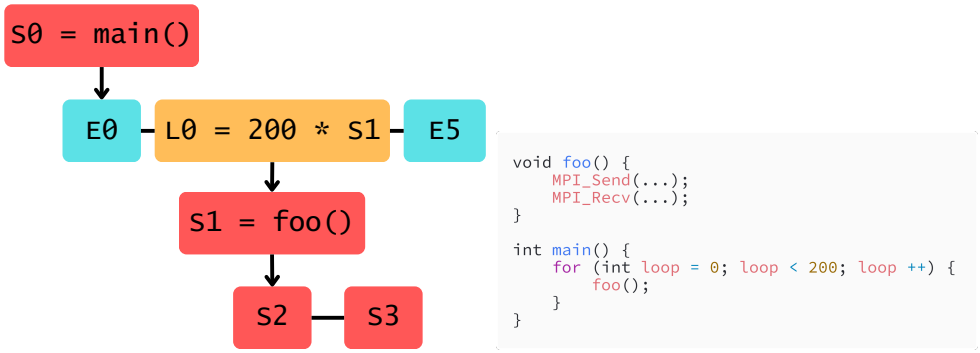


FIGURE 12 – Structure d'un programme, telle que générée par `HTF`, ave c à droite un code la générant. Les Séquences S2 et S3 correspondent respectivement à MPI_Send et MPI_Recv, se sont composées d'un Enter, d'un évènement ponctuel, et d'un Leave, qui n'ont pas été représentés par un soucis de lisibilité.

permet de limiter leur taille. Lorsqu'une nouvelle Séquence est créée, on vérifie parmi les Séquences déjà enregistrées si cette Séquence n'existe pas déjà. Pour accélérer cela, nous utilisons une fonction de hashage sur le tableau de tokens.

À l'ajout d'un token dans l'un de ces buffers, on cherche à détecter un éventuel motif récurrent. Pour cela, une comparaison est faite entre les n derniers tokens de notre buffer et les n tokens précédents, et ce pour des longueurs de plus en plus grandes. Pour éviter une complexité quadratique, nous limitons la taille maximale d'une séquence détectable de cette manière. Lorsqu'une boucle est détectée, les suites de tokens se répétant sont remplacés par un token désignant la boucle nouvellement créée. Un exemple d'une structure ainsi formée, ainsi que le code la générant, est disponible dans la Figure 12. Dans cette Figure, nous représentons la structure d'un programme faisant 200 appels à une fonction `foo`, contenant un MPI_Send et un MPI_Recv. Ainsi, la structure de notre trace contient une Boucle, faisant 200 itérations de la Séquence 1 (correspondant à la fonction `foo`). Ce Séquence 1 contient deux sous-Séquences, correspondants chacune à un appel d'une fonction MPI.

3.2 Stockage des timestamps

Notre gestion des timestamps s'inspire des méthodes d'optimisations présentées dans le travail d'amélioration de la compression d'OTF2 par Wagner et al. [18], ainsi que des la gestion des timestamps de Pilgrim [12]. Lors de l'écriture de la trace, les timestamps des évènements sont convertis en leur durée à la volée, ce qui nous permet ainsi une meilleure compression (grâce à la plus grande similarité des durées entre elles que des timestamps entre eux). Les durées ne sont pas stockées uniquement pour les évènements, elles le sont également pour les séquences. Cela génère de la redondance puisqu'il s'agit d'une donnée entièrement calculable grâce aux durées des évènements uniquement. Néanmoins, cela permet de limiter l'exploration de la trace, ce qui rentre dans notre logique de scalabilité.

Lors du stockage de ces durées, nous laissons à l'utilisateur le choix d'utiliser différentes compressions. En plus d'avoir implémenté un mode pour ne stocker aucun timestamp, nous avons implémenté trois méthodes de stockage des timestamps, ainsi qu'une méthode hybride :

— NONE : Pas de compression
— ZSTD : Compression des timestamps avec ZSTD

— MASKING : Compression des timestamps avec un encodage adaptatif
— MASKING_ZSTD : Encodage adaptatif puis compression avec ZSTD
— DYNAMIC_ZSTD : Compression adaptative avec ZSTD

Ce nouveau système d'encodage se base sur la méthode présentée dans la Figure 8, mais en partant du postulat que le nombre d'octets sur lequel les durées doivent être stockées sera le même, ou quasiment le même, pour toutes les durées. Pour calculer ce nombre d'octets, un `ET` binaire est réalisé entre toutes les durées, dont le résultat est appelé un masque. Ce masque est ensuite utilisé pour calculer le nombre d'octets nécessaires, qui sera indiqué au début du tableau.

L'option de compression adaptative avec ZSTD vient de la constatation que, pour certains programmes, il existe parfois un grand nombre d'évènements ou de séquences ne se répétant qu'une ou deux fois. Dans ces cas, la compression avec ZSTD devient désavantageuse. Cette option permet ainsi de n'utiliser ZSTD que dans les cas où cela permet de réduire la taille des tableaux de timestamps.

3.3 Lecture et analyse de la trace

Le format de trace `HTF` a été pensé pour pouvoir être lu de façon incrémentale, afin de ne pas charger toute la trace en mémoire en une seule fois. Cela permet à un utilisateur de charger ce qu'il juge intéressant. Pendant la durée du stage, nous avons développé deux outils en ligne de commande permettant de lire les traces au format `HTF` : un premier permettant d'afficher la structure de la trace, ainsi que d'afficher les différents évènements dans leur ordre chronologique, et un deuxième jouant un rôle de profiler sur la trace.

3.3.1 Lecture de la trace

Comme expliqué précédemment, le format `HTF` a été conçu pour être lisible de manière incrémentale. La séquence principale, correspondant au programme principal, est toujours notée comme étant la séquence 0. À partir de cette séquence, il devient possible d'explorer toute la trace. En répétant ce processus de manière récursive, on obtient alors une lecture de la structure de la trace, ce qui nous permet de l'afficher dans le terminal. Ce parcours "niveau par niveau" nécessite de lire la définition des Boucles et des Séquences, mais ne nécessite pas de connaître les timestamps, ni même de charger les données des évènements. La Figure 13 présente un exemple de représentation de la structure d'un programme obtenue en traçant un programme de Ping-Pong.

Une deuxième manière d'afficher une trace est de charger les timestamps, et d'afficher sa structure avec les timestamps et les durées correspondantes. On peut également afficher les évènements sans la structure, mais plutôt les uns à la suite des autres, avec leur timestamps correspondant. Dans ces deux cas, il est nécessaire de lire l'intégralité de la trace, avec les timestamps.

Cette lecture permet d'évaluer rapidement les zones intéressantes, par exemples les séquences les plus longues, ou bien des séquences anormalement longues lors d'une boucle.

Notre gestion des durées et des hiérarchies présente néanmoins un problème à la lecture. En effet, pour pouvoir connaître la durée d'un évènement à un endroit donné, il est nécessaire de connaître le nombre d'évènements similaires ayant eu lieu avant lui. La solution a été d'utiliser ce que nous avons appelé des **savestates**, qui, pour chaque séquence, sauvegarde l'état de notre curseur de lecture. Après une première lecture de la trace, il n'y a donc plus besoin de la parcourir pour pouvoir la lire efficacement. Cela

```
Reading events for thread 0 (P#0T#0):
Tag     Event
S6      E0_S L1 Eb_S
├─E0_S  THREAD_BEGIN()
├─L1    2 * S5 = L0 S4
│ ├─L0   (100, 10_000) * S3 = S1 S2
│ │ ├─S1  E1_E E2_S E3_L
│ │ │ ├─E1_E       Enter 0 (MPI_Send)
│ │ │ ├─E2_S       MPI_SEND(dest=1, comm=0, tag=0, len=16)
│ │ │ └─E3_L       Leave 0 (MPI_Send)
│ │ └─S2  E4_E E5_S E6_L
│ │   ├─E4_E       Enter 1 (MPI_Recv)
│ │   ├─E5_S       MPI_RECV(src=1, comm=0, tag=0, len=16)
│ │   └─E6_L       Leave 1 (MPI_Recv)
│ └─S4   E7_E E8_S E9_S Ea_L
│   ├─E7_E        Enter 2 (MPI_Barrier)
│   ├─E8_S        MPI_COLLECTIVE_BEGIN()
│   ├─E9_S        MPI_COLLECTIVE_END(op=0, comm=0, root=-1, sent=0, recved=0)
│   └─Ea_L        Leave 2 (MPI_Barrier)
└─Eb_S  THREAD_END()
```

FIGURE 13 – Structure d'un programme de Ping-Pong, réalisant une première boucle de 100 Send/Receive, puis de 10 000.

pourrait être encore amélioré en utilisant un système de comptage des tokens apparaissant dans les séquences, pour permettre d'éviter de parcourir de nouveau des séquences ayant déjà été visitées auparavant.

Un autre problème est de calculer le timestamp d'un évènement, car il est nécessaire d'avoir parcouru toute la trace auparavant afin de faire la somme des durées. Cela est cependant mitigé par la présence des durées des séquences, qui permettent d'éviter d'avoir à explorer toutes les séquences, comme illustré dans la Figure 14.

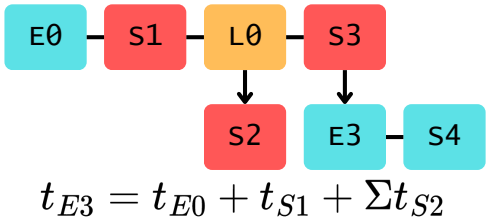


FIGURE 14 – Exemple de lecture partielle d'une trace. Les Séquences S1 et S2 ne sont pas explorées. Le timestamp de E3 peut être calculé en faisant la somme des durées des évènements et séquences le précédant.

3.3.2 Profiler

Le profiler que nous avons développé permet d'afficher différentes informations concernant la trace. Il propose notamment d'afficher toutes les différentes séquences, boucles et évènements, ainsi que des statistiques les concernant tels que le nombre d'occurrences, ou la durée moyenne des fonctions. Ces données sont calculées sans avoir à parcourir la trace ni à charger les durées.

Il est également possible de charger les durées afin de réaliser des calculs statistiques sur ces dernières. Comme il n'est pas nécessaire de charger tous les évènements et séquences en même temps, on pourrait imaginer une parallélisation de ces calculs, afin d'améliorer les performances de l'outil sur de plus grosses traces. On pourrait également imaginer ne faire ces calculs que sur les séquences intéressant l'utilisateur.

Une version simplifiée des données fournies par notre profiler est fournie Figure 15. Le programme tracé est le même que celui de la Figure 13, un Ping-Pong MPI, mais ne réalisant que 1 000 itérations, et les deux threads présentent donc quasiment exactement les mêmes évènements, d'où l'omission du deuxième thread dans ce résumé de trace. On constate qu'on retrouve tous les évènements attendus dans le programme du Ping-Pong, et en bon nombre.

```
Archive fffffffe:
    trace_name: ping_pong.htf
    Strings {.nb_strings: 26} :
        // Définitions de 26 Strings

    Regions {.nb_regions: 6} :
        // Définitions de 6 Régions (Sections de code)

    Location_groups {.nb_lg: 2}:
        // Définitions des 2 Location Groups (Processus)

    Locations {.nb_loc: 2}:
        // Définitions des 2 Locations (Fils d'exécutions)

    Threads {.nb_threads: 2}:
        0:    {.archive=0,    .nb_events=12, .nb_sequences=7, .nb_loops=2}
        // 2nd threads, identique au premier

    Archives {.nb_archives: 2}

Thread 0 {.archive: 0}
    Events {.nb_events: 12}
        E0      THREAD_BEGIN()                          {.nb_events: 1}
        E1      Enter 0 (MPI_Send)                      {.nb_events: 1100}
        E2      MPI_SEND(dest=1, comm=0, tag=0, len=16) {.nb_events: 1100}
        E3      Leave 0 (MPI_Send)                      {.nb_events: 1100}
        E4      Enter 1 (MPI_Recv)                      {.nb_events: 1100}
        E5      MPI_RECV(src=1, comm=0, tag=0, len=16)  {.nb_events: 1100}
        E6      Leave 1 (MPI_Recv)                      {.nb_events: 1100}
        E7      Enter 2 (MPI_Barrier)                   {.nb_events: 2}
        E8      MPI_COLLECTIVE_BEGIN()                  {.nb_events: 2}
        E9      MPI_COLLECTIVE_END(op=0, comm=0, root=-1, sent=0, recved=0)
                                                        {.nb_events: 2}
        Ea      Leave 2 (MPI_Barrier)                   {.nb_events: 2}
        Eb      THREAD_END()                            {.nb_events: 1}
    Sequences {.nb_sequences: 7}
        S0      {S6}        S1      {E1, E2, E3}
        S2      {E4, E5, E6}
        S3      {S1, S2}
        S4      {E7, E8, E9, Ea}
        S5      {S3, S4}
        S6      {E0, S5, Eb}
    Loops {.nb_loops: 2, .nb_allocated_loops: 2}
        L0      {.nb_loops: 2, .token: S3, .nb_iterations: [100, 1000]}
        L1      {.nb_loops: 1, .token: S5, .nb_iterations: [2]}
```

FIGURE 15 – Version abrégée des informations montrées par notre profiler sur la trace d'un ping-pong réalisé sur deux processus MPI.

`HTF` propose ainsi un traçage des programmes, une reconnaissance de leur structures, et plusieurs analyses à faible coût. Il faut cependant pouvoir vérifier que les traces ainsi générées sont bien valides, et que les performances de notre proposition sont comparables, voir meilleures que les outils déjà existant.

4 Évaluation

Dans cette section, nous évaluons `HTF`. Nous le comparons au format OTF2 [7], ainsi qu'à Pilgrim [12][13]. Nous vérifions dans un premier temps que les traces générées par `HTF` sont bien valides, et que HTF, OTF2 et Pilgrim stockent bien un nombre similaire d'évènements. Puis, nous évaluons le coût d'enregistrement d'un évènement, le surcoût de traçage, la taille des structures et des traces générées, et les algorithmes de compressions mis en place dans `HTF`. Nous évaluons également le temps d'analyse d'une trace par différents outils disponibles.

4.1 Paramètres expérimentaux

Pour évaluer les différents formats de trace, nous utilisons plusieurs applications :

— un **Ping-Pong MPI** (dont les traces ont déjà été présentées dans les figures 13 et 15), réalisant d'abord 100 itérations de MPI_Send/MPI_Recv entre deux machines, puis 1 000 000, et dont le code est disponible en annexe ;

— les **NAS Parallel Benchmarks** [29], un ensemble de programmes conçus pour évaluer les supercalculateurs. Nous utilisons l'implémentation MPI des kernels CG, FT, LU, MG et SP ;

— **AMG**, un solveur parallèle à multigrille algébrique utilisant à la fois MPI et OpenMP ;

— **MiniFE**, une application réalisant des calculs de problèmes de mécanique des fluides en utilisant la méthode des éléments finis avec MPI et OpenMP ;

— **Lulesh2.0**, une application résolvant un problème d'explosion de Sedov-Taylor, utilisant OpenMP et MPI ;

— **Quicksilver**, une application résolvant un problème de transport de particules de Monte-Carlo dynamique en utilisant OpenMP et MPI.

Ces différents benchmarks sont diversifiés, tant par leur régularité que par leur utilisation de MPI et d'OpenMP.

Chaque programme, sauf le Ping-Pong, est testé avec trois ensembles de paramètres différents, correspondants à une charge de calcul légère, moyenne, et élevée (nommés respectivement Small, Medium et Big). La Table 1 présente les différents paramètres utilisés pour réaliser ces mesures, ainsi que le nombre de threads utilisés pour les réaliser.

Toutes les mesures présentées ici sont réalisées en faisant une moyenne de résultats sur 10 mesures, en utilisant deux machines mises à notre disposition par Télécom SudParis. La machine Bran est équipée de deux processeurs Intel Xeon Gold 5220R, avec un total de 48 cœurs et 96 threads, 512Go de mémoire RAM répartis sur deux nœuds NUMA. La machine Sandor est équipée de deux processeurs AMD EPYC 7502, cumulant 64 cœurs et 128 threads, et 504Go de mémoire RAM répartis également sur deux nœuds NUMA. Nous avons réalisé toutes ces mesures avec un nombre fixe de processus MPI pour chaque benchmark, et la variable d'environnement `OMP_NUM_THREADS` fixée à 1.

Pour tester `HTF`, nous utilisons `EZTrace` et une surcharge de la librairie OTF2, afin de faire des appels au code de `HTF`. `HTF` intercepte et enregistre les mêmes évènements qu'`EZTrace`, que nous avons limités aux fonctions MPI et aux entrées et sorties de fonctions. Pilgrim n'intercepte que les fonctions MPI.

Pour chaque expérience, nous utilisons plusieurs configurations :

— NORMAL : l'application s'exécute normalement

Benchmark	$n_{process}$	Paramètre		
		Small	Medium	Big
NAS	16	Problem Size A	Problem Size B	Problem Size C
AMG	16	-n 100 100 100	-n 150 150 150	-n 200 200 200
MiniFE	16	-nx 100 -ny 100 -nz 100	-nx 200 -ny 200 -nz 200	-nx 300 -ny 300 -nz 300
Lulesh2.0	27	-s 10	-s 30	-s 50
Quicksilver	16	—lx 500 —ly 500 —lz 500		
		-n 1.000.000	-n 10.000.000	-n 20.000.000

TABLE 1 – Paramètres utilisés pour tester les différents benchmarks utilisés.

— OTF2 : l'application s'exécute et `EZTrace` intercepte les appels à MPI. `EZTrace` enregistre des événements avec la bibliothèque OTF2. Pour chaque fonction, `EZTrace` enregistre un événement indiquant le début de la fonction, un événement indiquant la fin de la fonction, ainsi que, dans certains cas, un événement supplémentaire indiquant un événement MPI (par exemple l'envoi ou la réception d'un message)
— HTF : l'application s'exécute et `EZTrace` intercepte les appels à MPI. `EZTrace` enregistre des événements avec la bibliothèque `HTF`.
— PILGRIM : l'application s'exécute et Pilgrim intercepte les appels à MPI, qu'il enregistre ensuite en utilisant son propre format.

Dans le cadre de Pilgrim et `HTF`, toutes les mesures ont été réalisés en stockant les timestamps sans compression (sauf précision contraire).

4.2 Validité des traces

Afin de vérifier la validité des traces générées par `HTF`, nous comparons le nombre d'évènements présents dans les traces d'`EZTrace` utilisant OTF2 et `HTF` et de Pilgrim. Pour la lecture des traces en OTF2, nous utilisons `otf2-print`. Pour Pilgrim, l'outil `pilgrim2text`. Pour `HTF`, nous utilisons notre outil `htf-info`. Nous vérifions le nombre d'évènements MPI collecté dans chacune des traces pour tous nos benchmarks en utilisant les paramètres Medium. Ces résultats sont présentés dans la Table 2.

On constate que `EZTrace`, `HTF` et Pilgrim collectent environ toujours le même nombre d'évènements, sauf pour AMG. Cela valide en partie les traces générées par `HTF`. Dans le cas d'AMG, la différence de nombre d'évènement entre `EZTrace` utilisant OTF2 et `HTF` peut s'expliquer par l'utilisation régulière de fonctions MPI, qu'`HTF` et Pilgrim n'interceptent pas.

4.3 Coût d'enregistrement d'un évènement

Nous comparons dans cette section le surcoût associé à la génération d'une trace sur le temps d'exécution du programme. Dans un premier temps, nous mesurons le surcoût à chaque enregistrement d'évènement en utilisant le programme de Ping-Pong, qui réalise 1000000 itérations d'un MPI_Send suivi d'un MPI_Recv.

La Figure 16 montre la latence mesurée par le programme Ping-Pong lorsque le tracage est désactivé ("NORMAL"), ou lorsque qu'il est activé, avec OTF2, `HTF` ou Pilgrim, sur la machine Sandor.

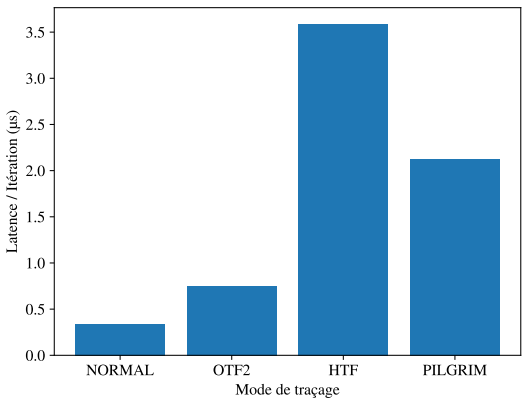
Benchmark	OTF2	HTF	Pilgrim
Ping Pong	400 408	400 408	400 408
NAS CG	671 328	671 328	671 312
NAS FT	1 536	1 536	832
NAS LU	2 446 988	2 446 988	2 446 988
NAS MG	138 480	138 480	136 880
NAS SP	308 448	308 448	430 288
LULESH	5 831 832	5 831 832	6 596 229
MiniFE	94 768	94 768	98 320
AMG	15 805 354	887 466	3 671 584

TABLE 2 – Nombre d'appels à MPI enregistré par `EZTrace`, `HTF` et Pilgrim sur différents benchmarks en configuration Medium.

La latence mesurée lorsqu'aucun outil ne trace l'application est de 330ns par itération. La latence mesurée avec `HTF` est de 3,6µs à chaque itération, contre 2,12µs pour Pilgrim, et 740ns pour OTF2.

Pour chaque itération de l'application, `EZTrace` enregistre 6 évènements (un Enter, un Leave, et un évènement ponctuel pour le MPI_Send et le MPI_Recv). Pilgrim n'enregistre que les évènements ponctuels, et n'enregistre donc que 2 évènements par itération. En prenant ces différences en compte, on peut alors voir que le coût d'enregistrement d'un évènement pour Pilgrim est de 894ns, contre 541 pour `HTF` et 69 pour OTF2.

Cette forte latence induite par Pilgrim et `HTF` s'explique par leurs algorithmes de détection de structure, qui rajoute inévitablement un temps de calcul à chaque ajout d'évènement. HTF collecte plus d'évènements, et induit naturellement une plus forte latence que Pilgrim.



	Latence ajoutée (ns)	Évènements / Itération	Latence / Évènement (ns)
OTF2	414	6	69
HTF	3 249	6	541
Pilgrim	1 789	2	894

FIGURE 16 – En haut : Latence (en µs) du programme de Ping-Pong MPI, en utilisant OTF2, Pilgrim et `HTF`. En bas : Tableau détaillant la latence pour chaque enregistrement d'un évènement.

4.4 Impact du traçage sur les performances d'applications

Afin d'évaluer l'impact des outils de traçage sur des applications réelles, nous mesurons les temps d'exécutions des benchmarks sur Sandor en utilisant OTF2, `HTF` et Pilgrim. Le temps d'exécution normal de l'application est également mesuré, et est utilisé pour calculer un temps d'exécution normalisé. Ces résultats sont présentés dans la Figure 17.

OTF2 a un surcoût compris entre 0,2 et 11%. Sur la plupart des benchmarks, Pilgrim a un surcoût similaire mais supérieur à OTF2 (entre 5 et 20%), mais dans le cas de Lulesh utilisant les paramètres Small, Pilgrim a un surcoût significatif (170%). Cela peut possiblement s'expliquer par le fait que Lulesh utilisant les paramètres Small s'exécute très rapidement (moins d'une seconde mesurée), et le surcoût de base de Pilgrim est donc significatif. Le surcoût d'HTF est du même ordre de grandeur que celui de Pilgrim (entre 5 et 15%).

Le surcoût induit par `HTF` reste donc raisonnable, bien que supérieur au surcoût induit par OTF2. Il est à noter que le surcoût mesuré sur les benchmarks les plus petits sont souvent plus forts, car les temps de calculs dans ces benchmarks sont plus faible par rapport à la latence induite par `HTF` ou Pilgrim.

4.5 Taille des structures

Nous comparons dans cette sous-section la taille de la structure des traces générées par Pilgrim et `HTF` sur Bran. Pour ce faire, nous avons modifié le code de Pilgrim, afin qu'il ne stocke aucun timestamp.

Les résultats de ces mesures sont présentés dans la Figure 18. Les structures de traces générées par `HTF` sont constamment plus lourdes que celles générées par Pilgrim, de plusieurs ordres de grandeur. Cela est d'autant plus vrai pour les applications les plus régulières, telles que les NAS. Pour les applications plus irrégulières, telles qu'AMG ou Quicksilver, la différence de taille entre `HTF` et Pilgrim devient moins conséquente.

Cette forte différence s'explique en partie par l'absence de compression inter-trace dans `HTF`. Elle s'explique également par le fait que Pilgrim utilise un algorithme de reconnaissance et de compression de structure avancé, mais qui force une structure très granulaire, contrairement à `HTF`, qui génère une structure très proche de celle du programme.

On remarque que, pour plusieurs applications, la taille de cette structure reste constante, et ne croît pas avec la taille des paramètres. Cela n'est pas le cas pour les applications plus irrégulières, telles que Quicksilver et AMG. En effet, en augmentant la taille des paramètres, on ne modifie pas la grammaire générée par la trace. On augmente uniquement le nombre d'itérations. Cela ne prend donc pas plus de place à stocker, et la structure du programme peut être enregistrée sur quelques dizaines de Kilo-octets (ou Méga-octets dans le cas des applications irrégulières).

4.6 Taille des traces

Nous comparons dans cette sous-section la taille des traces OTF2 à celles générées par Pilgrim et `HTF`, lorsque les timestamps sont enregistrés sans compression. La Figure 19 montre la taille de ces traces pour différents benchmarks, exécutés sur Sandor.

On observe que pour OTF2, `HTF` et Pilgrim, la taille des traces augmente lorsqu'on augmente la taille du jeu de donnée. Ces mesures montrent que, sans compression efficace, `HTF` n'est pas toujours meilleur qu'OTF2, puisque les traces générées sont du même ordre de grandeur, souvent plus petites. Pour le kernel NAS FT, les traces générées par `HTF` sont

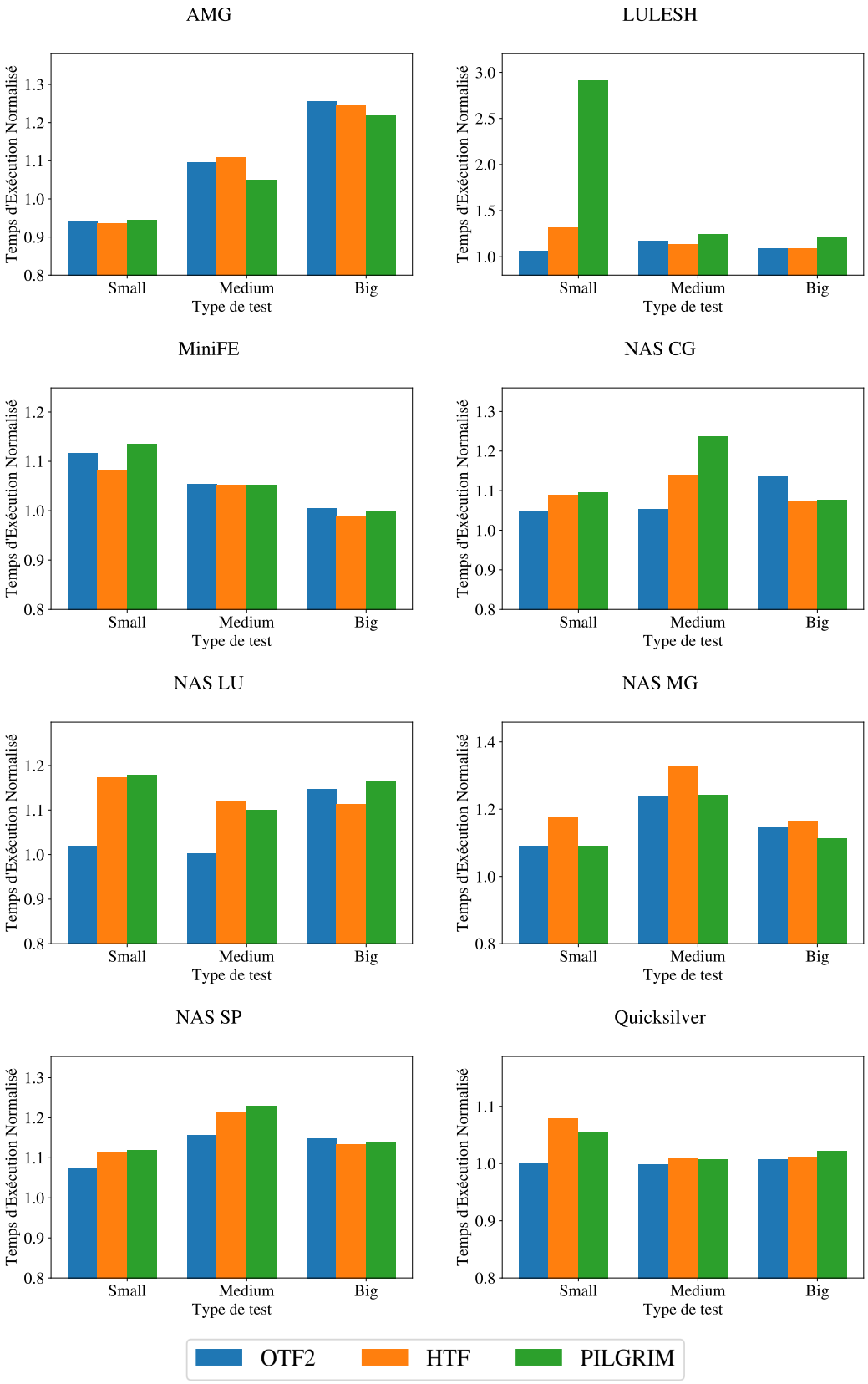


FIGURE 17 – Temps d'exécution de la génération d'une trace avec `EZTrace`, Pilgrim et `HTF`, normalisé par rapport au temps d'exécution sans traçage.

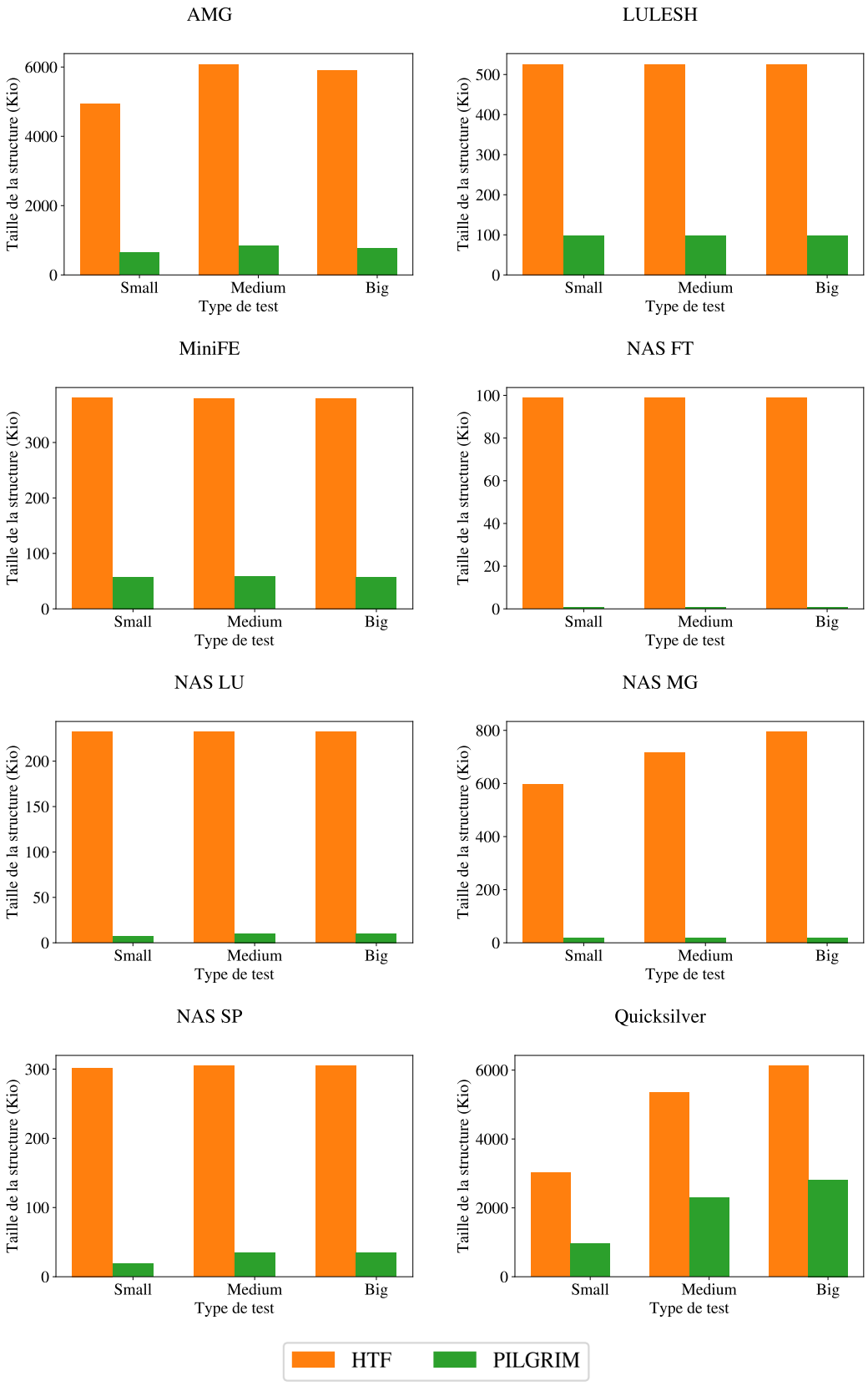


FIGURE 18 – Taille de la structure des traces (en Kibioctets) générées par Pilgrim et `HTF`.

plus grandes que celles générées avec OTF2. Cela s'explique par leur faible taille (quelque centaines de Ko), qui est donc du même ordre de grandeur que la taille de la structure. C'est donc le surcoût associé à la structure de la trace qui explique cette différence de taille.

On constate également que Pilgrim génère de manière constante des traces plus légères qu'OTF2 et `HTF`. Cela peut venir de la gestion très optimisée de la structure du programme que fait Pilgrim, mais également de la gestion de l'entrée et de la sortie des fonctions définies dans le programme : Pilgrim ne génère pas d'évènements pour ceux-ci, ce qui fait un certain nombre de timestamps en moins. Il n'est donc pas possible pour Pilgrim de fournir des analyses concernant ces fonctions spécifiques au programme tracé.

4.7 Évaluation des algorithmes de compression des timestamps

Nous comparons dans cette section l'efficacité des différentes méthodes de stockage des timestamps proposées par HTF. Les résultats de ces analyses réalisées sur Bran sont présentés dans la Figure 20.

Ces expériences montrent que la méthode de compression la plus efficace pour les traces ayant une taille conséquente est la combinaison de la méthode d'encodage adaptatif avec l'utilisation de ZSTD. Cette méthode hybride, notée MASKING_ZSTD sur les graphiques, fourni des résultats constamment supérieurs aux trois autres méthodes. On peut également constater que la méthode DYNAMIC_ZSTD n'offre pas des résultats significativement supérieurs à la méthode utilisant ZSTD de manière constante, contrairement à nos attentes.

De plus, les compressions à pertes proposées par Pilgrim (présentés dans la Figure 7) proposent des taux de compression bien supérieurs à ceux d'HTF.

4.8 Performance des analyses des traces

Nous comparons dans cette section les performances des outils de lecture des traces dans le cadre d'une analyse simple : compter le nombre d'appels à la fonction MPI_Send dans une fonction. Pour les traces générées par `EZTrace` utilisant OTF2 et Pilgrim, nous utilisons respectivement `otf2-print` et `pilgrim2text`, qui génèrent une frise chronologique de la trace. Ce frise est ensuite analysée par `grep` et `wc`. Pour `HTF`, nous utilisons notre outil `htf-info`. Nous avons analysé deux traces, générées lors de l'exécution d'AMG et de Lulesh avec les paramètres Big. Nous faisons ce choix car ces programmes génèrent des tailles de traces conséquentes. Les résultats de cette analyse, réalisée sur Sandor, sont présentés dans la Table 3.

`HTF` permet une analyse des deux traces en moins d'une seconde chacune, contre 17 et 19 secondes pour `EZTrace`, pour AMG et Lulesh respectivement. Pilgrim se situe entre `EZTrace` et `HTF`, avec 3 secondes pour AMG et 8 pour Lulesh.

	Temps d'analyse d'AMG (s)	Temps d'analyse de Lulesh (s)
OTF2	16.7	18.8
Pilgrim	3.1	8.1
HTF	0.5	0.6

Table 3 – Tableau comparant le temps nécessaire pour compter le nombre d'évènement MPI_Send sur des traces générées par AMG et Lulesh utilisant les paramètres Big, pour OTF2, `HTF` et Pilgrim.

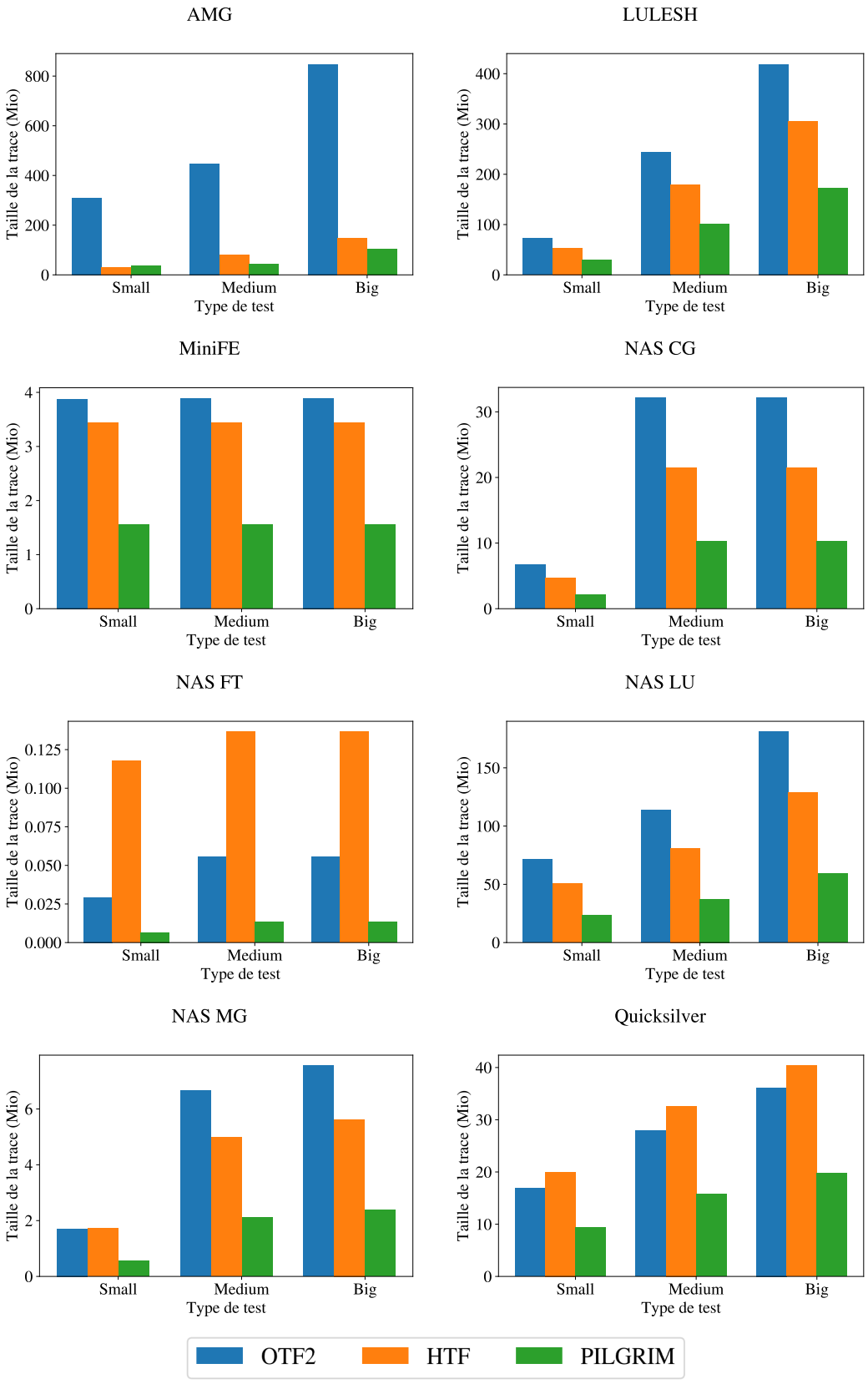


FIGURE 19 – Taille des traces (en Mébioctets) générées par `EZTrace`, `HTF` et Pilgrim, sans compression.

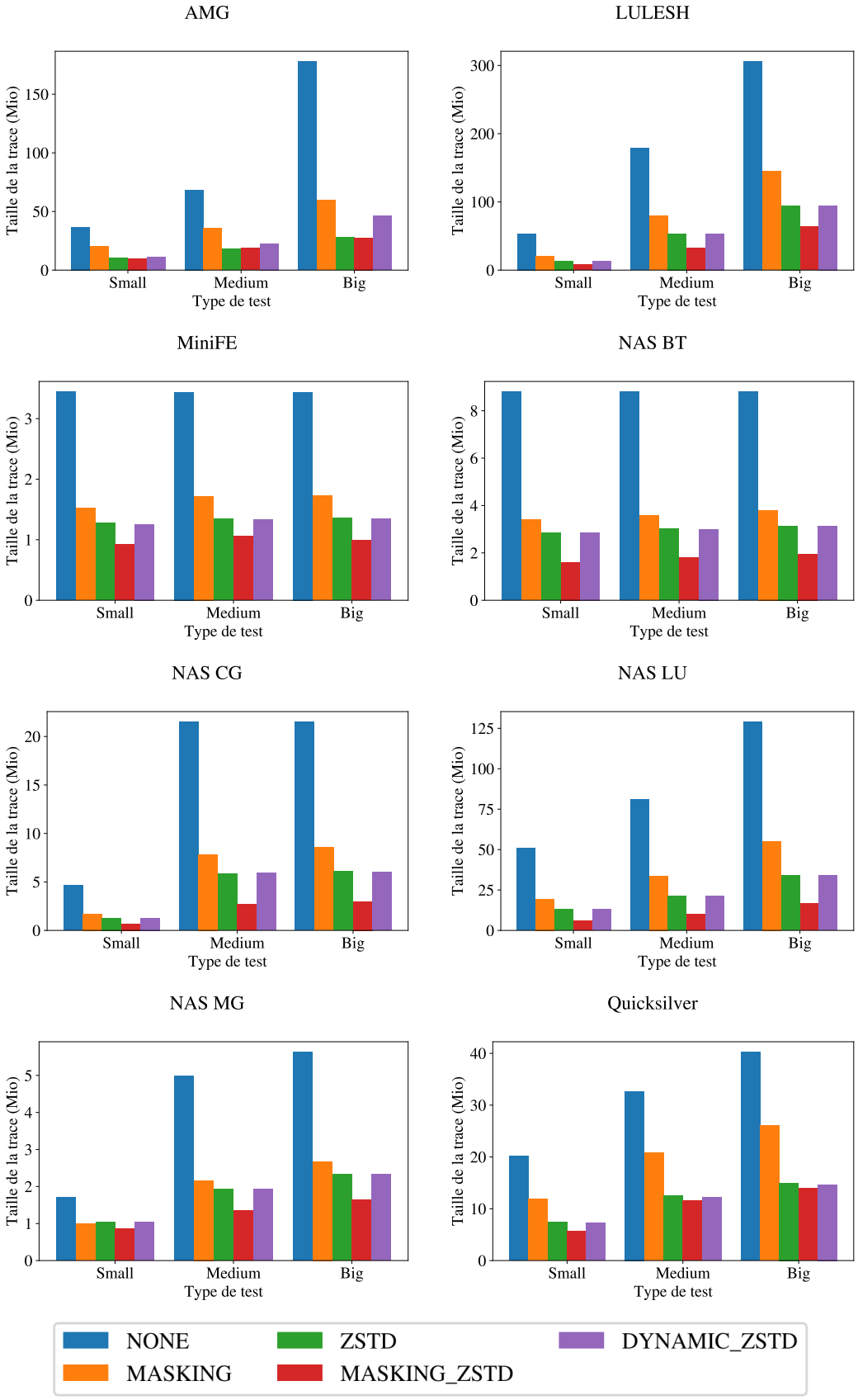


FIGURE 20 – Taille des traces obtenues avec `HTF` sur différents benchmarks en utilisant différentes méthodes de compression (en Mébioctets).

5 Conclusion

Le calcul exaflopique présente de nouvelles problématiques pour le passage à l'échelle des applications et des outils déjà existant, tels que les traces. Du fait du très grand nombre de machines, il est nécessaire de développer un nouveau format adapté à cette nouvelle échelle, et permettant de réaliser plusieurs analyses rapidement.

Notre proposition, `HTF`, est un nouveau format de trace prévu pour l'Exascale. Ce format répond bien à plusieurs problèmes se présentant à l'écriture des traces sur un système exaflopique : compression efficace, détection de structure, lecture simple et dynamique, et analyse rapide.

Vis à vis de la taille des traces, `HTF` est moins efficace que Pilgrim, puisqu'il produit des traces et des structures plus lourdes. Cependant, au niveau du surcoût associé au traçage, les deux sont équivalents, alors que `HTF` collecte plus de données, et propose plus d'opportunités d'analyse à faible coût.

Ainsi, `HTF` génère des traces compacts, avec un surcoût de traçage faible, et des analyses simples à implémenter et rapides d'exécution.

6 Perspectives Futures

Ce travail ouvre plusieurs perspectives, qui seront étudiées à court terme. Par exemple, concernant la compression des traces, plusieurs pistes d'amélioration restent ouvertes, telles que l'utilisation d'autres techniques de compression pour les timestamps (telles que SZ ou ZFP), ou un meilleur algorithme de détection de motifs.

Nous avons développé un algorithme de détection de boucles qui pourrait être plus efficace que celui actuellement implémenté. L'idée consiste à accéder efficacement aux emplacements de toutes les répétitions d'un certain token pour chaque niveau hiérarchique. Ainsi, lors de la reconnaissance d'une séquence, nous ne considérerions que les emplacements avec le dernier token similaire au dernier token du buffer. Cette approche réduirait significativement le nombre de séquences potentielles et, par conséquent, les comparaisons nécessaires. Cependant, l'implémentation d'un tel algorithme est compliquée par l'obtention des indices d'un token dans le buffer. Bien qu'une tentative d'utilisation d'un arbre AVL ait été faite, elle s'est avérée trop coûteuse, notamment pour les programmes irréguliers. Bien que l'utilisation d'un arbre visait à obtenir une complexité logarithmique pour l'accès aux emplacements, le surcoût associé à la structure auto-équilibrante de l'arbre était significatif et n'était pas pris en compte dans les calculs de complexité. En supposant un accès linéaire à la liste des emplacements dans le buffer, l'algorithme aurait également une complexité linéaire. Il nous reste donc à trouver un moyen d'implémenter cela efficacement.

L'une des analyses que nous aimerions ajouter à notre profiler est une implémentation de l'heuristique de score de contention proposée par Bouskiaa et al. [28], le score SCI [4]. Ce score se calculant en utilisant les durées des séquences et les durées des évènements, il serait relativement facile à calculer sans avoir, une fois de plus, à charger toutes les données en mémoire.

```
--- Reading bran_quicksilver.htf ----------------------------------
In S6
    100 µs  [                            ] E0_S      THREAD_BEGIN()
   5.53  s  [############################] L1        3 * S5 = L0 S4
    100 µs  [                            ] Eb_S      THREAD_END()

====================================================================
--- Reading bran_quicksilver.htf ----------------------------------
In S6/L1
   525  ms  [###                         ] S5        L0(1.000) S4
   4.00  s  [############################] S5        L0(8.000) S4
   1.00  s  [#######                     ] S5        L0(2.000) S4
```

FIGURE 21 – Concept d'une trace d'un programme lambda chargée avec les timestamps, dans une interface interactive avec `ncurses`.

À terme, nous souhaiterions que l'outil d'affichage de nos traces utilise `ncurses`, une bibliothèque fournissant une API pour le développement d'interfaces utilisateur à menu déroulant, qui nous semble très adaptée à la structure de notre trace. Nous souhaiterions nous inspirer de l'outil `ncdu`, qui utilise cette bibliothèque pour montrer interactivement la taille des différents dossiers dans une arborescence. Nous envisageons de proposer le développement d'une telle interface à des étudiants en tant de projet de développement informatique. Un concept d'une telle interface est présentée Figure 21.

La question d'une compression inter-trace, c'est-à-dire une reconnaissance de structure non pas à l'intérieur d'un seul thread, mais à travers les différents threads d'une même

4. Slowdown Caused by Interference

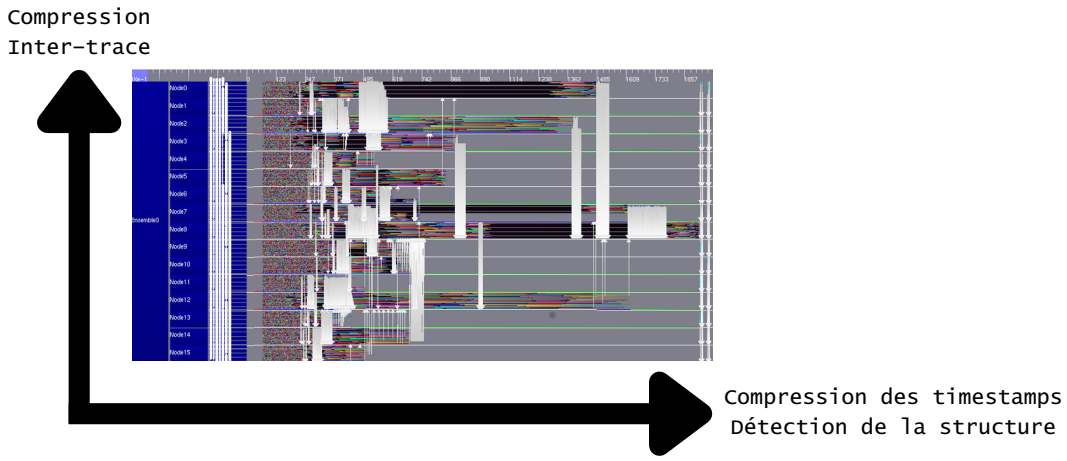


FIGURE 22 – Nos deux axes de scalabilité.

application est également un sujet à aborder, notamment pour la scalabilité en "hauteur", c'est-à-dire en nombre de threads. Cette idée de scalabilité en hauteur (à l'opposé d'une scalabilité en "largeur", c'est-à-dire en nombre d'évènement dans un seul thread) est illustrée dans la Figure 22. La possibilité de le confier à un étudiant en Master a été considérée, mais la détection de motifs récurrents parmi les threads ne suffira sans doute pas à lui tout seul à permettre une compression intéressante, et un travail de recherche sur ce point reste nécessaire.

Enfin, l'ajout de benchmarks utilisant StarPU ou CUDA est un objectif à considérer à court terme dans le cadre du projet NumpEx. Dans le cas de StarPU, nous nous intéresserons notamment à la reconnaissance de motifs récurrents parmi les tâches conçues par ce framework.

7 Conclusion Personnelle

Ce stage m'a permis de découvrir le monde de la recherche, car je n'en avais jusque là jamais fait à proprement parler. La seule introduction à la recherche que j'ai pu effectuer pendant mon cursus étaient les cours de Java NIO (proposé par Denis CONAN) et Infrastructure pour le Cloud (proposé par Pierre SUTRA) de Télécom SudParis, qui nous demandaient en effet de lire un article scientifique et d'en faire un état de l'art. Je me suis beaucoup plu dans ce milieu, que ce soit grâce à l'environnement de travail agréable, les collègues sympathiques, ou bien mes tuteurs qui m'auront très bien encadré pour me lancer sur ce projet, mais également pour mes premières recherches et lectures d'articles. Ce stage était censé être un moyen de voir si la recherche m'intéressait, et si une thèse me convenait, et c'est avec joie que je vais donc commencer une thèse sur ce même sujet en octobre de cette année.

Références

[1] *Perf Wiki*. adresse : https://perf.wiki.kernel.org/index.php/Main_Page (visité le 16/08/2023).

[2] S. L. Graham, P. B. Kessler et M. K. Mckusick, « Gprof: A call graph execution profiler », *ACM SIGPLAN Notices*, t. 17, nº 6, p. 120-126, 1982, issn : 0362-1340. doi : 10.1145/872726.806987.

[3] W. E. Cohen, « Tuning programs with OProfile », *Wide Open Magazine*, t. 1, p. 53-62, 2004.

[4] J. Vetter et C. Chambreau, « mpiP: Lightweight, Scalable MPI Profiling », en,

[5] *OpenMP profiling — Caliper 2.10.0 documentation*. adresse : https://software.llnl.gov/Caliper/OpenMP.html (visité le 17/08/2023).

[6] F. Trahay, F. Rue, M. Faverge, Y. Ishikawa, R. Namyst et J. Dongarra, « EZTrace: A Generic Framework for Performance Analysis », in *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, mai 2011, p. 618-619. doi : 10.1109/CCGrid.2011.83.

[7] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel et F. Wolf, « Open Trace Format 2 The Next Generation of Scalable Trace Formats and Support Libraries », en, 2011.

[8] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker et B. Mohr, « The Scalasca performance toolset architecture », *Concurrency and Computation: Practice and Experience*, t. 22, nº 6, p. 702-719, 2010. doi : https://doi.org/10.1002/cpe.1556.

[9] A. Knüpfer, C. Rössel, D. A. Mey et al., « Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir », en, in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel et M. M. Resch, éd., Berlin, Heidelberg : Springer Berlin Heidelberg, 2012, p. 79-91, isbn : 978-3-642-31475-9 978-3-642-31476-6. doi : 10.1007/978-3-642-31476-6_7.

[10] M. Noeth, P. Ratn, F. Mueller, M. Schulz et B. R. de Supinski, « ScalaTrace: Scalable compression and replay of communication traces for high-performance computing », *Journal of Parallel and Distributed Computing*, t. 69, nº 8, p. 696-710, août 2009. doi : 10.1016/j.jpdc.2008.09.001.

[11] J. Zhai, J. Hu, X. Tang, X. Ma et W. Chen, « CYPRESS: Combining Static and Dynamic Analysis for Top-Down Communication Trace Compression », in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, nov. 2014, p. 143-153. doi : 10.1109/SC.2014.17.

[12] C. Wang, P. Balaji et M. Snir, « Pilgrim: Scalable and (near) lossless MPI Tracing », in *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, p. 1-13. doi : 10.1145/3458817.3476151.

[13] C. Wang, Y. Guo, P. Balaji et M. Snir, « Near-Lossless MPI Tracing and Proxy Application Autogeneration », *IEEE Transactions on Parallel and Distributed Systems*, t. 34, p. 123-140, jan. 2023. doi : 10.1109/TPDS.2022.3215942.

[14] V. Danjean, R. Namyst et P.-A. Wacrenier, « An Efficient Multi-level Trace Toolkit for Multi-threaded Applications », en, in *Euro-Par 2005 Parallel Processing*, t. 3648, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg : Springer Berlin Heidelberg, 2005, p. 166-175, ISBN : 978-3-540-28700-1 978-3-540-31925-2. DOI : 10.1007/11549468_21.

[15] *Trace Event Format*, en. adresse : https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/preview?usp=embed_facebook (visité le 15/08/2023).

[16] L. M. Schnorr, *PajeNG*, original-date: 2012-08-28T19:37:19Z, avr. 2022. adresse : https://github.com/schnorr/pajeng (visité le 16/08/2023).

[17] R. Iakymchuk et F. Trahay, « LiTL: Lightweight Trace Library », Télécom Sud-Paris, rapp. tech., 2013.

[18] M. Wagner, A. Knupfer et W. E. Nagel, « Enhanced Encoding Techniques for the Open Trace Format 2 », *Procedia Computer Science*, t. 9, p. 1979-1987, 2012. DOI : 10.1016/j.procs.2012.04.216. (visité le 07/08/2023).

[19] A. Colin, F. Trahay et D. Conan, « PYTHIA: an oracle to guide runtime system decisions », in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, 2022, p. 106-116. DOI : 10.1109/CLUSTER51413.2022.00025.

[20] F. Trahay, É. Brunet, M. M. Bouksiaa et J. Liao, « Selecting Points of Interest in Traces Using Patterns of Events », in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2015, p. 70-77. DOI : 10.1109/PDP.2015.30.

[21] Y. Collet et M. Kucherawy, « Zstandard Compression and the application/zstd Media Type », Internet Engineering Task Force, rapp. tech. RFC 8478, oct. 2018. DOI : 10.17487/RFC8478.

[22] S. Di et F. Cappello, « Fast Error-Bounded Lossy HPC Data Compression with SZ », in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, ISSN: 1530-2075, mai 2016, p. 730-739. DOI : 10.1109/IPDPS.2016.11.

[23] P. Lindstrom, « Fixed-Rate Compressed Floating-Point Arrays », *IEEE Transactions on Visualization and Computer Graphics*, t. 20, août 2014. DOI : 10.1109/TVCG.2014.2346458.

[24] L. M. Schnorr, *StarVZ*, original-date: 2017-11-27T12:35:42Z, mai 2023. adresse : https://github.com/schnorr/starvz (visité le 22/08/2023).

[25] *Catapult - Using Trace Viewer Casually*. adresse : https://chromium.googlesource.com/external/github.com/catapult-project/catapult/+/refs/heads/master/tracing/docs/getting-started.md (visité le 22/08/2023).

[26] A. Knüpfer, H. Brunst, J. Doleschal et al., « The Vampir Performance Analysis Tool-Set », en, in *Tools for High Performance Computing*, M. Resch, R. Keller, V. Himmler, B. Krammer et A. Schulz, éd., Berlin, Heidelberg : Springer, 2008, p. 139-155, ISBN : 978-3-540-68564-7. DOI : 10.1007/978-3-540-68564-7_9.

[27] *ViTE's Project Page*. adresse : https://solverstack.gitlabpages.inria.fr/vite/ (visité le 22/08/2023).

[28] M. S. M. Bouksiaa, F. Trahay, A. Lescouet et al., « Using Differential Execution Analysis to Identify Thread Interference », *IEEE Transactions on Parallel and Distributed Systems*, t. 30, n° 12, p. 2866-2878, déc. 2019. doi : 10.1109/TPDS.2019.2927481.

[29] *NAS Parallel Benchmarks*. adresse : https://www.nas.nasa.gov/software/npb.html (visité le 22/08/2023).

A Code du Ping-Pong MPI

```c
/*
 * Copyright (C) CNRS, INRIA, Université Bordeaux 1, Télécom SudParis
 * See COPYING in top-level directory.
 * Code from https://gitlab.com/\ezt/\ezt/-/blob/master/test/mpi/mpi_ping.c
 */

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <mpi.h>

#define LEN      16
#define LOOPS    10000
#define WARMUP   100

static unsigned char *main_buffer = NULL;

void SEND(char* buffer, int len, int dest, int tag) {
  MPI_Send(buffer, len, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
}

void RECV(char* buffer, int len, int src, int tag) {
  MPI_Recv(buffer, len, MPI_CHAR, src, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

void fill_buffer(char *buffer, int len) {
  int i;
  for (i = 0; i < len; i++) {
    buffer[i] = 'a' + (i % 26);
  }
}

void compute(unsigned usec) {
  double t1, t2;
  t1 = MPI_Wtime();
  do {
    t2 = MPI_Wtime();
  } while ((t2 - t1) * 1e6 < usec);
}

int main(int argc, char **argv) {
  int comm_rank = -1;
  int comm_size = -1;
  char host_name[1024] = "";
  int dest;
  int len = LEN;
  int iterations = LOOPS;

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &comm_rank);

  if (gethostname(host_name, 1023) < 0) {
    perror("gethostname");
    exit(1);
```

```
  }

  if (comm_size != 2) {
    fprintf(stderr, "This program requires 2 MPI processes, aborting...\n");
    goto out;
  }

  fprintf(stdout, "(%s): My rank is %d\n", host_name, comm_rank);

  dest = (comm_rank + 1) % 2;

  main_buffer = malloc(len);
  fill_buffer(main_buffer, len);

  int i;

  for (i = 0; i < WARMUP; i++) {
    if (!comm_rank) {
      SEND(main_buffer, len, dest, 0);
      RECV(main_buffer, len, dest, 0);
    } else {
      RECV(main_buffer, len, dest, 0);
      SEND(main_buffer, len, dest, 0);
    }
  }

  double t1, t2;
  t1 = MPI_Wtime();
  MPI_Barrier(MPI_COMM_WORLD);
  for (i = 0; i < iterations; i++) {
    if (!comm_rank) {
      SEND(main_buffer, len, dest, 0);
      RECV(main_buffer, len, dest, 0);
    } else {
      RECV(main_buffer, len, dest, 0);
      SEND(main_buffer, len, dest, 0);
    }
  }
  MPI_Barrier(MPI_COMM_WORLD);

  t2 = MPI_Wtime();

  if (!comm_rank) {
    double latency = 1e6 * (t2 - t1) / (2 * iterations);
    printf("# iterations: %d\nLen: %d\nLatency: %lf\n", iterations, len, latency);
  }
  out: free(main_buffer);
  MPI_Finalize();

  return 0;
}
```