TP n°4
Analyse de trace avancée

Performance Des Systèmes Parallèles

2024-2025

L'objectif de ce TP est de vous faire manipuler des traces qui ont été générée par EZTrace.

Les traces vous sont fournies. Elles résultent de l'exécution de différentes applications MPI sur la machine Jean Zay du GENCI.

On s'intéresse à deux formats de trace : OTF2 (traces séquentielles), et Pallas (traces structurelles). Nous verrons dans un premier temps que les traces séquentielles d'OTF2 sont particulièrement indigestes à manipuler, le code étant extrêmement verbeux et difficile à comprendre. Nous utiliserons des analyses déjà codées, que vous compilerez et exécuterez.

Nous passerons ensuite sur le format de trace structurel Pallas, qui permet de coder plus facilement les mêmes analyses. Votre travail sera de coder les mêmes programmes que ceux utilisés en OTF2, permettant d'analyser le contenu de ces traces : matrices de communication MPI, détection de contention, histogrammes de durée de certaines fonctions etc.

1 Analyse de trace en OTF2

Dans cette section, on s'intéresse à utiliser des traces OTF2 pour analyser la performance d'applications. Vous n'allez pas coder les analyses directement (vous comprendrez rapidement pourquoi en lisant les codes). Vous allez par contre compiler et exécuter ces programmes sur des traces OTF2 fournies, afin de prendre en main des outils d'analyse de performance classiques.

Pour rappel, le format OTF2 stocke séquentiellement chaque évènement d'un thread (= appel de fonction) séquentiellement dans un gros *buffer*, qui est ensuite écrit sur le disque.

1.1 Installation de l'environnement de travail

On reprend ici la suite du tp3. Ainsi, on suppose que la bibliothèque OTF2 est installée dans le répertoire

`$HOME/PDSP/tp3`

Ainsi, il doit y avoir dans ce répertoire le fichier otf2.env permettant de sourcer les chemins vers l'installation OTF2. Si ce n'est pas le cas, reprenez le TP3 et faite la partie installation.

Une fois la bibliothèque OTF2 installée :

1. Déplacez vous dans **/pub/FISE_PDSP35/tp/tp4/otf2**
2. Copiez le script installOTF2Analysis.sh dans le répertoire de votre choix (typiquement, **$HOME/PDSP/tp4**)
3. Lisez ce script et lancez le. Il va copier le dossier **otf2_analysis** dans **$HOME/PDSP/tp4**, en vérifiant qu'une installation OTF2 existe bien. Si besoin, modifiez le chemin vers l'installation OTF2 si vous n'aviez pas utilisé le chemin par défaut du tp3.

Déplacez vous dans le dossier **$HOME/PDSP/tp4/otf2_analysis**. Vous remarquerez qu'il y a trois dossiers, chacun contenant le code d'une analyse particulière pour des traces OTF2. Dans la suite de cette section, on va les explorer une à une et les exécuter sur des traces OTF2 qui se trouvent dans le répertoire **/pub/FISE_PDSP35/tp/tp4/otf2/traces_otf2**.

1.2 Matrice de communication MPI

Placement de processus MPI En programmation MPI (distribution des calculs sur plusieurs nœuds), le placement des processus est critique pour la performance des communications. En effet, les communications inter-nœuds nécessitent de transiter par le réseau, ce qui est beaucoup moins performant qu'un échange de données local à la mémoire du nœud. Ainsi, placer les processus MPI sur les ressources en fonction de leur affinité d'échance de données permet d'améliorer significativement la performance des applications.

Pour déterminer cette affinité, une approche très commune est d'extraire une matrice de communication pour l'application MPI concernée. Cette matrice montre l'intensité des échanges de données point à point (typique des MPI_Send ou MPI_Isend) entre chaque paire de processus. La Figure 1 illustre une telle matrice de communication pour le benchmark NAS LU sur 512 rangs MPI.

Exécution en OTF2 On vous fournit une implantation de programme pour générer une matrice de communication à partir d'une trace OTF2 dans le répertoire **otf2-communication-matrix**.

1. Lisez le code qui se trouve dans src/otf2_comm_matrix.cpp et essayez de comprendre comment se charge en mémoire une trace OTF2, et comment on la parcourt.
2. Compilez le programme à l'aide du script compile.sh
3. Lancez ensuite le programme sur les trois traces fournies en OTF2 sur **/pub**. Ne les copiez pas dans votre home car cela prendrait de la place inutilement. Redirigez la sortie standard dans un fichier resultat.mat spécifique à chaque application, qui servira à générer un visuel de la matrice.
4. Utilisez le script python plot_comm_matrix.py sur les fichiers .mat pour générer les matrices de communication en pdf. Observez les sur les différentes applications pour voir les variations des pattern de communication.

Quel bilan faites-vous de l'utilisabilité des traces OTF2, et de la praticité/complexité du programme d'analyse pour les manipuler ?

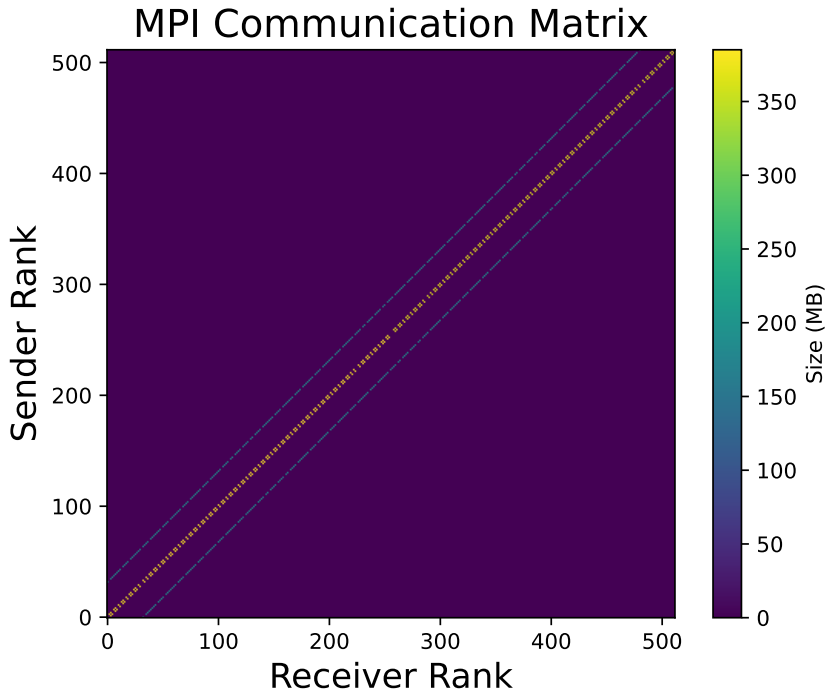

FIGURE 1 – Matrice de communication pour le benchmark NAS LU sur 512 rangs MPI.

1.3 Analyse de contention

Métrique de contention La contention est le ralentissement d'opérations utilisant le réseau dû à la congestion sur ce dernier. Mesurer la contention sur certains parties du code est classique en analyse de performance.

Nous allons ici utiliser une métrique, CSI, introduite par [1], qui estime l'impact de la contention dans un morceau de code (typiquement, une fonction) sur la performance totale d'une application, en l'occurence ici sur le thread qui exécute cette fonction.

Une valeur de CSI proche de 1 pour une fonction F indique que la majeure partie du temps l'exécution du thread provient de variations du temps d'exécution de F. Ainsi, il y a de fortes chances que de la contention sur des ressources partagées impacte la performance de F : un verrou, un accès disque etc. Toutefois, si la fonction produit toujours le même temps d'exécution, alors il n'y a pas d'effet de contention et le SCI sera proche de 0.

Cette métrique a été définie comme

$$\frac{\sum_i (d_i - \bar{d})}{thread\_duration}$$

où $\bar{d}$ est l'exécution la plus rapide de la fonction, et d_i est la durée de la ième occurence de la fonction.

Exécution en OTF2 On vous fournit une implantation de programme pour calculer des scores de contention à partir d'une trace OTF2 dans le répertoire **otf2_analysis/otf2-profile**.

1. Lisez le code qui se trouve dans src/otf2_profile.cpp et essayez de comprendre comment la trace est parcourue et quelles sont les options du programme.
2. Compilez le programme à l'aide du script compile.sh
3. Lancez ensuite le programme sur les trois traces fournies en OTF2 sur **/pub**. Ne les copiez pas dans votre home car cela prendrait de la place inutilement.
4. Analysez la sortie du programme et faites varier les options à l'exécution.

1.4 Histogrammes de durée de fonctions

Retour vers la contention L'analyse de contention avec le score CSI permet d'obtenir l'information de quelle fonction (ou partie de code) est susceptible d'être affectée par la contention. Une fois ces fonctions déterminées, une analyse plus poussée des effets de la contention sur cette fonction est nécessaire.

Afin de visualiser clairement les variations d'exécution pour une fonction donnée, une approche par histogramme est souvent adoptée. Elle consiste à afficher sous la forme d'un histogramme les durées des différentes occurences de la fonction. La Figure 2 présente un exemple d'affichage d'une telle analyse.

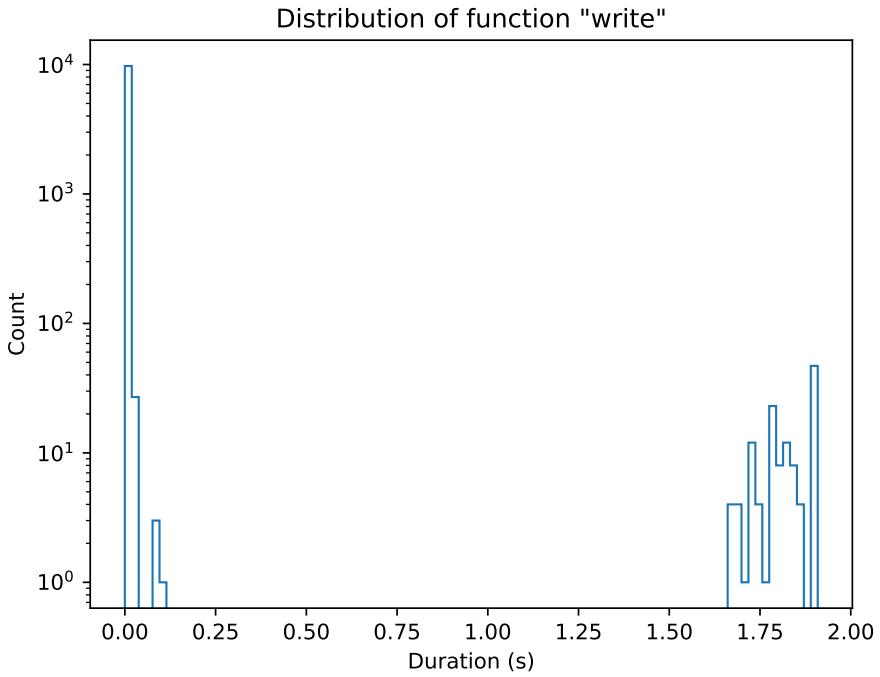

FIGURE 2 – Histogramme de la fonction *write* lors de l'entraînement d'un réseau de neurone ResNet50 sur le jeu de données ImageNet [2]

Ici, on voit clairement que les écritures s'exécutent normalement en $\sim$ 0.1 seconde, mais perdent parfois une facteur 20 de performance. Ainsi, l'exécution du programme va être retardé par des variations au niveau des appels système. Ces aspects sont particulièrement impactant sur les entraînement de réseaux de neurones, qui demandent des quantités de lecture/écriture très importantes à chaque *epoch* jusqu'à convergence du modèle. Il a été montré dans [3] qu'on peut éviter ces accès aux données à chaque étape en redistribuant les données (d'entrée et d'apprentissage) entre les différents *workers*, limitant ainsi les appels systèmes à la première itération. Le gain est de l'ordre de 20% sur l'exécution totale de l'application.

Travail attendu On vous fournit une implantation de programme pour calculer des histogrammes à partir d'une trace OTF2 dans le répertoire **otf2_analysis/otf2-histogram**.

1. Lisez attentive le code qui se trouve dans src/otf2_histogram.cpp et essayez de comprendre comment se calcule l'analyse. Etudiez les différentes options du programme pour quand vous le lancerez.
2. Compilez le programme à l'aide du script compile.sh
3. Lancez ensuite le programme sur les trois traces fournies en OTF2 sur **/pub**. Ne les copiez pas dans votre home car cela prendrait de la place inutilement.
4. Analysez la sortie du programme et plottez différents histogramme avec les scripts src/plot_*.py. Restreignez vous à générer pour une seul thread et une ou deux fonctions. Sélectionnez-les à partir de l'outil de profiling de la section précédente.

Bilan sur OTF2 Quel bilan faites-vous de l'utilisabilité des traces OTF2, et de la praticité/complexité du programme d'analyse pour les manipuler ?
Pensez vous que ce format de trace soit accessible à n'importe quel utilisateur ?
Est-il obligatoire de parcourir tous les évènements de la trace pour faire les analyses ci-dessus ? Est-ce un problème ?

Nous avons ainsi utilisé quelques analyses en OTF2. Quelles sont selon vous les avantages et inconvénients de l'utilisation des traces OTF2 ?

2 Analyse de performance avec un format de trace structurel : Pallas

Nous allons dans cette section nous intéresser au format de trace Pallas. La sous-section suivante est une brève présentation du format de trace Pallas pour vous permettre de l'utiliser pour coder les mêmes analyses que celles vues avec OTF2 précédemment.

2.1 Brève description de Pallas

Pallas est un format de trace récent et open-source développé dans le but de non seulement générer des traces avec un surcoût léger, mais aussi de stocker les données sous une format qui facilite leur analyse post-mortem. Pallas peut être utilisé avec des outils de traçage utilisant le format OTF2 tels qu'EZTrace. Ainsi, EZTrace génèrera des traces au format Pallas au lieu du format OTF2. On peut ainsi transformer une trace OTF2 en Pallas et vice-versa, facilitant ainsi la transition entre les deux.

Pallas collecte des évènements (aka des appels de fonctions), et les traite à la volée pour détecter la structure du programme (telles que les fonctions et les boucles). La liste des évènements d'un thread est ainsi représentée par une structure similaire à une grammaire, représentée par un ensemble de séquences comme le montre la Figure 3. Chaque séquence contient une liste de *tokens* (= des entiers de 32bits) qui peuvent correspondre soit à un évènement, à une séquence ou à un boucle de tokens.

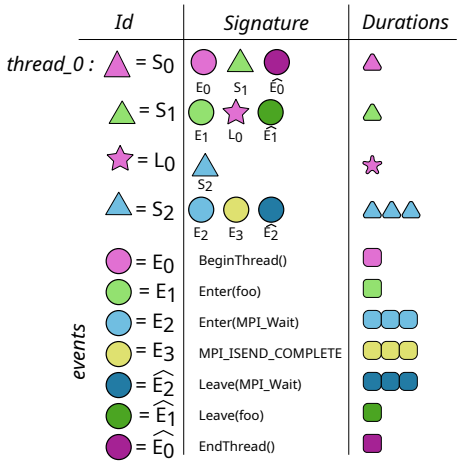

Figure 3 – Représentation hiérarchique de la trace d'un thread. La trace commence avec une séquence $S0$ qui contient l'évènement $E0$, puis la séquence $S1$, puis l'évènement $\hat{E}0$. Chaque séquence est composée d'une liste d'évènements (cercles), séquences (triangles), ou boucles (étoiles).

La détection de la structure se fait grace à un algorithme de détection basé sur le hashing des évènements (leur type + métadonnées), permettant ainsi de découvrir et factoriser des répétitions d'évènements, comme démontré par la Figure 4.

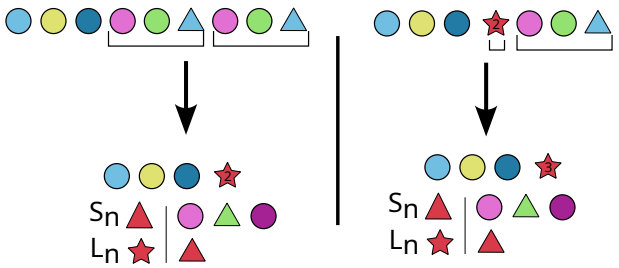

Figure 4 – Factorisation de la répétition d'un token séquence dans un token de boucle, comme présenté dans la Figure 3. Sur la gauche, 3 tokens sont répétés deux fois, et sont remplacés par une boucle à deux itérations. À droite, trois tokens composant une nouvelle itération de S_n apparaissent après la boucle. Ils sont donc enlevés et le compteur d'itération est augmenté pour S_n.

Lors de l'écriture de la trace sur le disque, Pallas stocke les tokens d'un thread (= sa structure) dans un petit fichier, et toute les autres données de performance (durée des évènements, leurs paramètres etc) dans un fichier de métadonnées séparé. Ainsi, une analyse peut simplement utiliser les fichiers de structure sans avoir besoin de charger les (gros) fichiers de métadonnées.
La Figure 5 illustre la structure de données mémoire d'une trace Pallas.

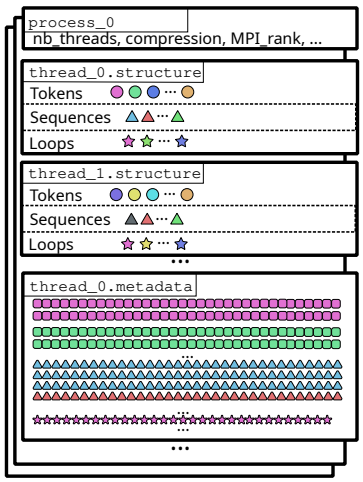

FIGURE 5 – Structure de données en mémoire d'une trace au format PALLAS.

Présenter en détails toute l'implantation de PALLAS n'est pas l'objectif de ce TP. Votre travail est de vous familiariser avec l'API de PALLAS afin de l'utiliser pour réaliser les mêmes programmes d'analyse de performance que ceux vu avec OTF2 précédemment. Pour cela, vous n'avez besoin de comprendre la structure de quelques classes seulement.

2.2 Installation de Pallas

2.2.1 Installation des sources

Comme pour OTF2, on vous fournit un script pour installer PALLAS. On installe également des algorithmes de compression qui peuvent être utilisés pour compresser les traces PALLAS. Nous n'utiliserons dans ce tp que des traces compressées avec ZSTD. L'algorithme de compression est automatiquement détectée par PALLAS lors de la lecture d'une trace. Ainsi, la décompression est automatique lors que l'on souhaite lire un trace, et l'utilisateur n'a pas à se soucier d'indiquer le bon algorithme.

Avant de manipuler des traces PALLAS, commençons par l'installer. Le script suit le même principe que celui pour OTF2 dans le tp3.

1. Lisez puis lancez le script /pub/FISE_PDSP35/pallas/installAll.sh
2. Vérifiez que tout se soit bien installé dans le répertoire **$HOME/PDSP/tp4** (ou tout autre répertoire de votre choix que vous auriez indiqué). Vous devriez avoir les sources de PALLAS installées ainsi qu'un dossier **$HOME/PDSP/tp4/pallas/pallas_analysis**

N'oubliez pas de sourcer le fichier pallas.env dans votre terminal avant d'appeler des programmes associés à la bibliothèque.

On vous fournit un ensemble de traces PALLAS pour tester vos programmes d'analyse dans le répertoire **/pub/FISE_PDSP35/tp/tp4/pallas/pallas_traces**. **Là encore, ne copiez pas ces traces dans votre $HOME, utilisez les directement en appelant votre programme avec le chemin vers les fichiers eztrace_log.pallas contenus dans les dossiers de chaque trace.**

2.2.2 API de Pallas

. Pour vous familiariser avec PALLAS et ses structures internes, nous allons parcourir deux headers.

pallas.h Ouvrez le fichier $HOME/PDSP/tp4/install/pallas/include/pallas/pallas.h. Ce fichier contient les briques élémentaires d'une trace : tokens, évènements, séquences et boucles.

1. Parcourez ces différentes classes (Token, Event, EvenSummary, Sequence, Loop, Thread) et essayez de comprendre leur imbrication.

pallas_archive.h Ouvrez le fichier $HOME/PDSP/tp4/install/pallas/include/pallas/pallas_archive.h.

Ce fichier contient les définitions des *Locations*, *Definitions* et *Archives*. Si vous avez été attentif au code d'OTF2, PALLAS reprend les mêmes notions ici :

— Les *Archives* sont les abstractions les plus proches du fichier de trace. Il y a une par processus.

— Les *Locations* représentent les threads, qui sont groupés par *LocationGroups* (processus, machines, etc.).

— Les *Definitions* servent à stocker les données locales aux threads qui ont vocation à être parsées : *String*, *regions* (=nom des fonctions dans les évènements), *attributes*).

Tout cela doit vous sembler bien mystérieux. Néanmoins, nous vous allez vite comprendre les quelques structures de données essentielles au parcours des traces, ce qui nous suffira amplement dans la suite de ce TP. Essentiellement, nous allons parcours toutes les *archives* d'une trace, puis itérer sur chaque *thread* de ces archives et parcours toutes les *sequences* contenues dans ces threads.

pallas_print PALLAS fournit un petit programme d'exploration de trace rapide : pallas_print. Essayez de l'utiliser sur les traces dans le répertoire **/tmp/pallas_traces**.

Jouez avec les options suivantes pour personnaliser la sortie du programme :

```
Usage : pallas_print [options] <trace file>
        -h      Show this help and exit
        -v      Enable verbose mode
        -T      Enable per thread mode
        -d      Show durations
        -t      Hide timestamps
        -S      Enable structure mode (per thread mode only)
        -s      Do not unroll sequences (structure mode only)
        -l      Do not unroll loops (structure mode only)
        --thread thread_id      Only print thread <thread_id>
        -f      Generate a flamegraph file
        -c      Generate a csv file
```

2.3 Matrice de communication

Dans cette section, on reprend l'approche de la Section 1.2 mais cette fois en format PALLAS au lieu d'OTF2.

Allez inspecter le répertoire **$HOME/PDSP/tp4/pallas/pallas_analysis/pallas-comm-matrix**. Un script compile.sh vous permet de compiler automatiquement le programme et générer un exécutable dans mcpallas-comm-matrix/install/bin.

On vous fournit un squelette de programme déjà exhaustif dans le répertoire **src**. Votre travail est de compléter le fichier src/pallas_comm_matrix.cpp. Pour cela, il est **vivement** conseillé d'ouvrir en parallèle les fichiers headers de PALLAS (pallas.h et pallas_storage.h notamment) qui contiennent toutes les fonctionnalités de l'API PALLAS dont vous aurez besoin.

Remarquez qu'avec PALLAS, on a seulement besoin de parcours les fichiers de structure de chaque thread. En effet, nous n'avons pas besoin des durées des évènements, mais seulement des attributs de certains d'entre eux.

Travail attendu

1. Commencez par lire **ATTENTIVEMENT** le fichier source du programme. Analysez bien en bas du fichier la fonction main pour comprendre ce que l'on souhaite faire.
2. Votre travail est de compléter le *TODO* qui se trouve dans la fonction compute_matrix. Une aide textuelle vous est fournie pour vous guider pas à pas. N'hésitez pas à demander de l'aide si besoin.
3. Testez votre programme avec les traces fournies. Par défaut, le programme crée un fichier de sortie pallas_comm_matrix.mat à chaque exécution. Vous pouvez changer ce nom de fichier en passant un paramètre au programme, afin de générer un fichier de sortie dédié à chaque trace.
4. Plottez vos matrices de communication avec le script python plot_comm_matrix.py qui vous est fourni.
5. **[Si vous avez du temps] :** Parallélisez à l'aide de directives OpenMP le calcul de la matrice de communication !

Comparaison avec OTF2 Quelles sont les avantages d'utiliser PALLAS par rapport à OTF2 en terme de structuration des données dans le format de trace ?
Quelle est la complexité de l'algorithme en OTF2 et en PALLAS, en tenant compte de la structuration interne des données dans les deux formats de trace ?

Comparez notamment les lignes de code nécessaires à l'élaboration de ce programme en OTF2 et en PALLAS.

2.4 Calcul de score de contention

Dans cette section, on reprend l'approche de la Section 1.3 mais cette fois en format Pallas au lieu d'OTF2. Pallas stocke les appels de fonctions sous forme de séquences (un évènement ENTER, la fonction et une évènement LEAVE) pour chaque thread. Pallas stocke également des données statistiques (durée min, max et moyenne) de chaque séquence (c'est à dire, les statistiques liées à toutes les occurences de cette séquence dans un thread donnée). Nous allons évidemment utiliser ces informations pour calculer le score de contention. Quelles différences voyez vous avec le format OTF2 ? Quels sont les avantages/inconvénients de la structure de stockage des données en Pallas ?

Allez inspecter le répertoire **$HOME/PDSP/tp4/pallas/pallas_analysis/pallas-contention**. Un script compile.sh vous permet là-aussi de compiler automatiquement le programme et générer un exécutable dans mcpallas-contention/install/bin.

Comme précédemment, vous allez devoir compléter un programme déjà implanté dans le répertoire **src**. Tout comme pour la matrice de communication, il est **vivement** conseillé d'ouvrir en parallèle les fichiers headers de Pallas (pallas.h et pallas_storage.h notamment) qui contiennent toutes les fonctionnalités de l'API Pallas dont vous aurez besoin.

Travail attendu

1. Commencez par lire **ATTENTIVEMENT** le fichier source du programme. Analysez bien en bas du fichier la fonction main pour comprendre ce que l'on souhaite faire, ce qui est déjà codé et ce qu'il vous reste à faire.
2. Votre travail est de compléter le *TODO* qui se trouve dans la fonction sequence_duration_thread. Une aide textuelle vous est fournie pour vous guider pas à pas. N'hésitez pas à demander de l'aide si besoin.
3. Testez votre programme avec les traces fournies. Analysez le fichier de sortie pour chaque trace, et trouvez les fonctions les plus impactées par la contention. Gardez les en tête car nous utiliserons dans la section suivante pour générer des histogrammes de durée de ces fonctions.

Comparaison avec OTF2 Quelle est la complexité de l'algorithme de calcul de score de contention avec Pallas ? Quelle est la différence avec OTF2 ?
Dans le cas de cette analyse, quels avantages déduisez vous à l'utilisation de traces structurelles et quels sont les inconvénients auxquels vous pouvez penser ?

2.5 Génération d'histogrammes de durée de fonctions

Dans cette section, on reprend l'approche de la Section 1.4 mais cette fois en format Pallas. Rappelez vous que Pallas stocke les appels de fonctions sous forme de séquences (un évènement ENTER, la fonction et une évènement LEAVE) pour chaque thread. A chaque séquence est associé un vecteur de durée de chacune des occurences de cette séquence.

D'un point de vue conceptuel, quelle va être la structure de notre programme
Allez inspecter le répertoire **$HOME/PDSP/tp4/pallas/pallas_analysis/pallas_profiling**. Un script compile.sh vous permet là-aussi de compiler automatiquement le programme et générer un exécutable dans **pallas-contention/install/bin**.

Comme précédemment, vous allez devoir compléter un programme déjà implanté dans le répertoire **src**. Tout comme pour les analyses précédentes, il est **vivement** conseillé d'ouvrir en parallèle les fichiers headers de Pallas (pallas.h et pallas_storage.h notamment) qui contiennent toutes les fonctionnalités de l'API Pallas dont vous aurez besoin.

Travail attendu Regardez la fonction main du fichier src/pallas_profiling.cpp On peut fournir en paramètre du programme une liste de threads à inspecter, et même restreindre à seulement certaines fonctions. Afin de ne pas avoir des temps d'analyse trop élevés, on va se restreindre aux deux fonctions ayant le score de contention le plus élevé.

Dans cette analyse, on veut donc filtrer les séquences qui correspondent aux fonctions que l'on passe en paramètre du programme. Pour cela, il nous faut déjà considérer seulement les séquences dont le contenu est

un token de type *TypeEvent* (car sinon la séquence contient d'autres séquences et n'est donc pas une fonction). Une fois ces séquences trouvées, il faut récupérer l'évènement associé au token *TypeEvent*, puis de récupérer sa *region* en utilisant une méthode de la classe *Thread* :

```
Thread.getRegionStringFromEvent(pallas::Event* e)
```

Vous pouvez utiliser le code suivant pour trouver le nom d'une fonction pour une *Sequence* s* donnée :

```
const pallas::Sequence* s = thread->sequences[i];

// To get the region (function/task name), find a TypeEvent token in the list of tokens of the
     sequence
pallas::Event* e = NULL;
long unsigned int j = 0;

// Look for the first token that is an event
while (j < s->tokens.size()) {
  // if not found, continue //
  if (s->tokens.at(j).type != pallas::TypeEvent)
    j++;
  // if found, get the associated event and break
  else {
    e = thread->getEvent(s->tokens.at(j));
    break;
  }
}

// if we did not find such event (could be a sequence of sequence or loops), just skip the
    sequence
if (!e)
  continue;

// get the region form the event
std::string region = thread->getRegionStringFromEvent(e);
// sanity check
if (region == "INVALID")
  continue;
```

1. Pour chaque trace, déterminez avec le programme de la Section 2.4 les deux fonctions ayant le score de contention le plus élevé, tout thread confondu.
2. Lisez **TRÈS ATTENTIVEMENT** le code source du programme, notamment les variables et fonctions entre les lignes 24 et 71.
3. Votre travail est de compléter le *TODO* qui se trouve dans la fonction find_sequence_durations. Une aide textuelle vous est fournie pour vous guider pas à pas. N'hésitez pas à demander de l'aide si besoin. Votre code doit rendre fonctionnel le reste de la fonction sans avoir à modifier le code au dessus et en dessous de la zone *TODO*.
4. Testez votre programme avec les traces fournies en vous restreignant aux deux fonctions trouvées dans 1. pour chacune des traces. Utilisez les deux scripts de plotting python fourni pour générer les histogrammes de durée.

Comparaison avec OTF2 Quelle est la complexité de l'algorithme de calcul des histogrammes avec Pallas ? Quelle est la différence avec OTF2 ?

Avec les trois analyses OTF2 et Pallas que nous avons vus, quel format de trace vous semble :

— générer le moins de surcoût au traçage ?

— générer des traces les plus compactes ?

— faciliter l'implantation de programmes d'analyse des traces générées ?

— limiter le coût d'exploration des données dans les traces lors des analyses ?

En utilisant les réponses à ces questions, établissez un bilan global des pour/contre pour OTF2 et Pallas, en allant du coût de traçage à l'utilisation des traces.

3 Pour aller plus loin

Vous devriez maintenant être capables d'écrire vos propres programmes d'analyse de traces au format Pallas. On s'est surtout concentré ici sur des traces MPI, mais Pallas est capable de tracer tout type de paradigme de programmation en utilisant la généricité des modules de traçage EZTrace.

En vous inspirant des programmes vus dans la section précédente, vous pouvez coder une analyse qui :

— pour chaque thread, calcule le temps passé dans chaque bibliothèque de calcul (MPI, OpenMP, autre) et le temps inactif par rapport à la durée totale

— en lui ajoutant un paramètre, permet de détailler le temps passé dans chaque fonction appelée d'une bibliothèque (ou liste de bibliothèques) donnée

— génère un barplot de ces informations

Vous pouvez en coder d'autres si vous le souhaitez (par exemple, trouver des *late sender* comme vu en cours).

Références

[1] M. S. M. Bouksiaa, F. Trahay, A. Lescouet, G. Voron, R. Dulong, A. Guermouche, E. Brunet, and G. Thomas, "Using differential execution analysis to identify thread interference," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 12, p. 2866–2878, Dec. 2019.

[2] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet : A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[3] T. T. Nguyen, F. Trahay, J. Domke, A. Drozd, E. Vatai, J. Liao, M. Wahib, and B. Gerofi, "Why globally re-shuffle ? Revisiting data shuffling in large scale deep learning," in *IPDPS 2022 : 36th International Parallel & Distributed Processing Symposium*. Lyon (virtual), France : IEEE, May 2022, pp. 1–12.