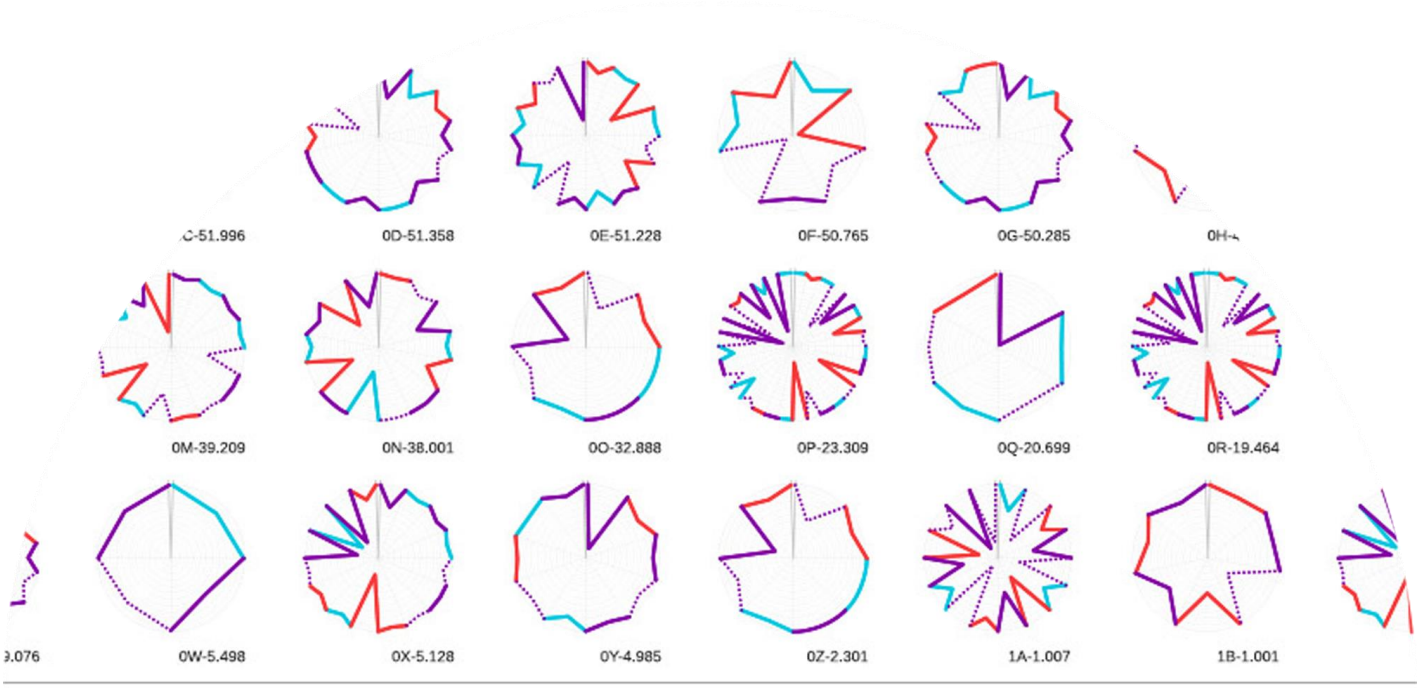
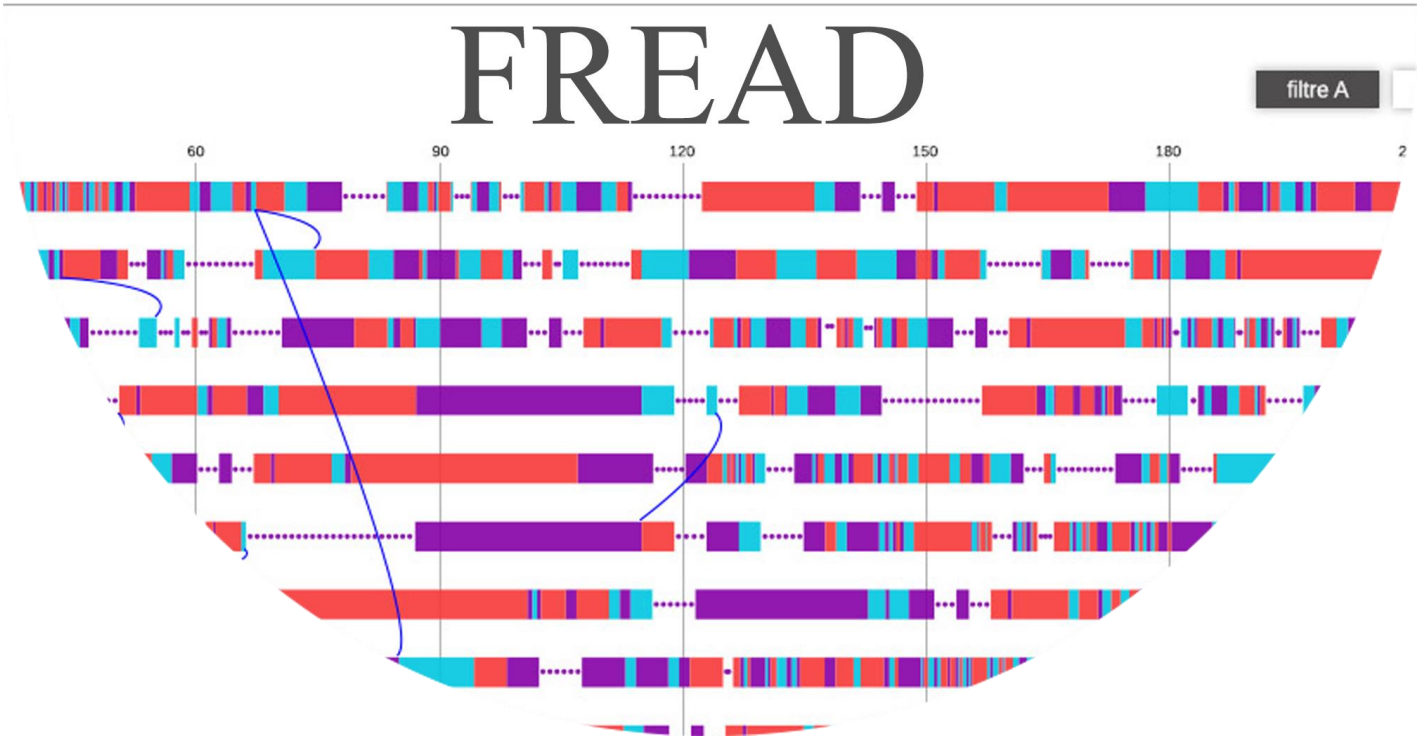


FREAD



FREAD

Manuel d'utilisation

Fast Reading Data

Comment améliorer le traitement de vos traces d'exécutions ?

Jérôme BERTHELIN, Emma GOLDBLUM, Maxime GUILLEM, Victor ROUQUETTE

jerome.berthelin@ensiie.fr, emma.goldblum@telecom-sudparis.eu, maxime.guillem@telecom-sudparis.eu, victor.rouquette@telecom-sudparis.eu

Contents

1 - Fast Reading Data

FREAD est un **logiciel de visualisation de traces d'exécution** de programmes multi-processus (ou en anglais, multithreaded). Comme d'autres logiciels de visualisation de traces d'exécution, il permet, à partir de données brutes relatant l'exécution d'un programme, de rendre des diagrammes **lisibles par le développeur,** dans le but d'analyser les performances du programme en vue de l'améliorer.

FREAD se veut plus facile d'utilisation que les logiciels déjà existants, de plus il permet la lecture de traces dépassants plusieurs Giga Octets sans risquer de voir l'ordinateur ralentir. FREAD se veut donc **ergonomique et puissant.**

FREAD ne se contente pas seulement d'afficher les traces d'exécutions, difficilement lisibles (elles peuvent s'étaler sur plusieurs centaines de milliers de lignes). Il permet d'afficher des **traces préalablement traitées** par un algorithme développé par **François Trahay** qui permet de détecter des motifs reconnaissables dans une trace. Nous nous basons sur cette **reconnaissance de motifs** pour optimiser les performances de rendu et l'ergonomie.

Le logiciel est distribué de manière libre, sous la **licence BSD**. Il est fait de manière à pouvoir être utilisé et modifié facilement par le plus grand nombre de personnes, chercheurs, entreprises ou particuliers.

Dans cette optique de rendre FREAD souple et adaptable à différents formats de traces, le parseur, prenant en charge le format PAJE dans un premier temps, a été pensé pour être **facilement remplacé** par un parseur lisant un autre format si besoin en est. Pour cela, un parseur adapté doit être codé et ajouté à FREAD, l'interfaçage se faisant via des fonctions prédéfinies qui seront appelées par FREAD lors de la lecture.

L'idée étant pour vous développeurs, travaillant sur des logiciels multi-processus de pouvoir, à travers la trace**, repérer l'exécution de certaines des fonctions clés du programme**, afin de voir si celles-ci travaillent de **manière optimale** dans la plupart des cas. Ainsi, avec FREAD, vous développeur aurez dans votre boite à outils un outil ergonomique, pratique, facile et rapide d'utilisation.

2 - Installation

FREAD est pour le moment dédié à des utilisateurs travaillant dans un environnement GNU/Linux. Il fonctionne parallèlement au logiciel de création de trace EZTrace. Afin d'installer FREAD il vous faudra suivre les instructions suivantes.

2.1 - Prérequis

FREAD se voulant léger et facile d'installation les prérequis d'installation se veulent aussi légers que possible.

- Pour la compilation, il est nécessaire que votre compilateur prenne en charge la **bibliothèque standard** de la **norme C++14**

- Pour le rendu, FREAD se base sur la librairie SFML. Pour l'instant il utilise la dernière version de la librairie (**SFML-2.4.1**).
  Disponible sur le site officiel (https://www.sfml-dev.org/) ou avec votre gestionnaire de paquets préféré (ex : http://packages.ubuntu.com/).
  Si vous rencontrez des problèmes à la compilation, assurez-vous que la librairie SFML est correctement identifiée dans le flag du makefile :

```
> LIBS := -lsfml-graphics -lsfml-window -lsfml-system -pthread
```

2.2 - Installation

Ouvrez votre Shell préféré et placez-vous dans le dossier contenant le makefile du projet. Il se trouve normalement à la racine. Ensuite entrez la commande :

```
> make
```

Le logiciel devrait se compiler sans rencontrer de problèmes. Dans le cas contraire vérifiez vos dépendances pour les liens externes. Assurez-vous aussi que le dossier obj dans lequel les shared-objects sont placés existent bien. Une erreur peut venir de là.

2.3 – Lancement

Pour lancer FREAD, placez-vous dans le **dossier src**, l'exécutable devrait s'y trouver après la compilation. Assurez-vous que le fichier **states.conf** se trouve bien dans le même dossier ; il est nécessaires à l'initialisation du parseur. Le format de la commande est :

```
> Fread <main.trace_path>
```

Un exemple de commande pour lancer FREAD :

```
> Fread ../mpi_ping_compressed/main.trace
```

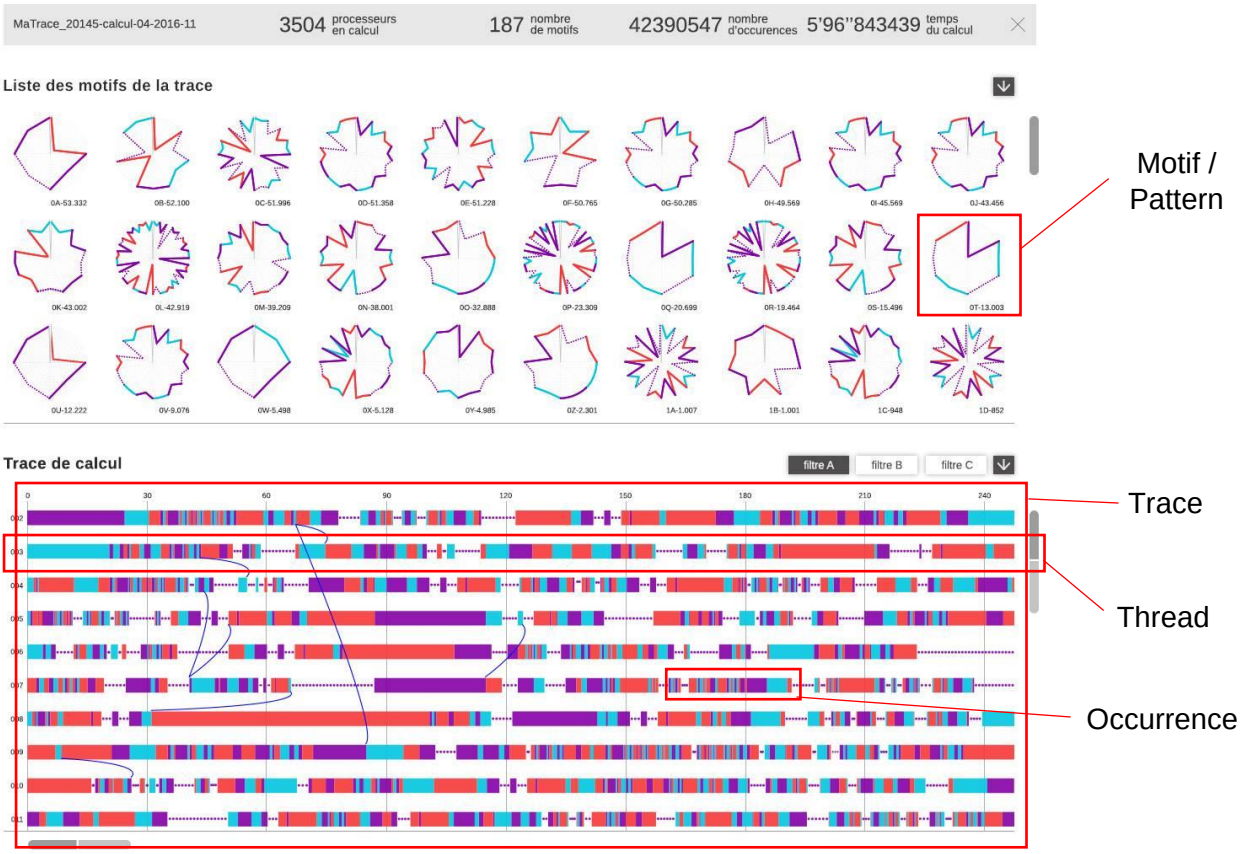
NB : La trace doit avoir été **générée par EZTrace** si vous voulez que FREAD puisse la lire correctement.

3 – Utilisation

L'utilisateur arrive sur notre outil et se voit proposer l'importation d'une trace, une fois celle-ci trouvée et chargée l'interface principale se dévoile.

3.1 - Écran d'accueil

Il est décomposé en 2 grandes parties, celle du dessus comportant tous les motifs trouvés dans la trace, puis celle du dessous comportant la **trace totale** sous la forme de longues lignes colorées représentant **chaque thread** et les différents événements les constituants.

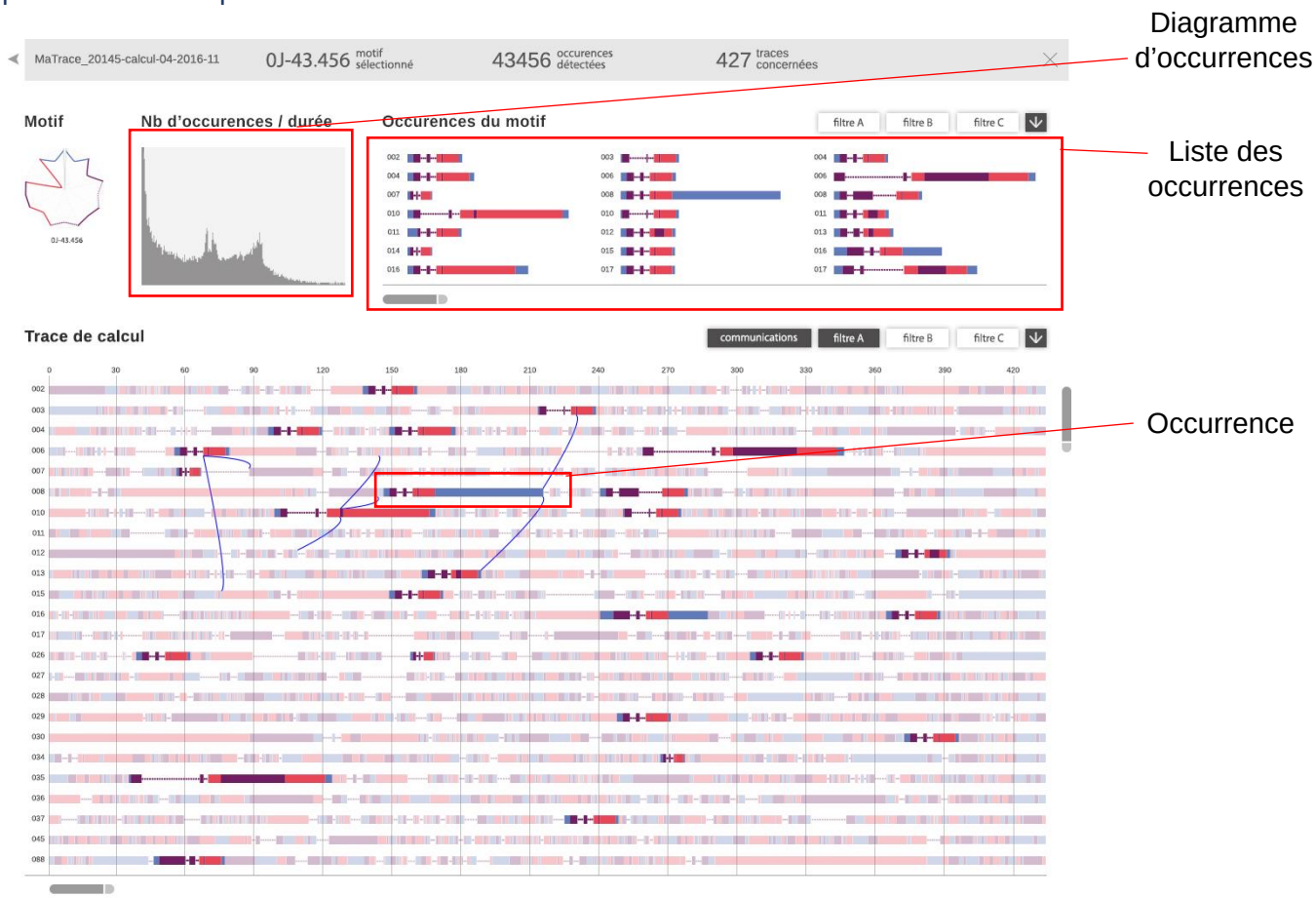


En haut de l'écran vous pouvez lire les **informations générales** de la trace (nombre de motifs, nombre de threads, nombre de lignes, temps total, etc …). Vous avez aussi la possibilité de vous déplacer dans la trace.

Pour ce qui est des motifs vous pouvez les **trier** selon différents critères (le nombre d'occurrences, le temps total, etc …).

3.2 - Écran d'un motif

Si vous cliquez sur un motif, l'interface se modifie pour que vous puissiez voir toutes les occurrences de ce motif à sa droite ainsi qu'un **histogramme** de répartition. De plus, ils sont repérables en sur-opacité sur la zone du bas qui se sera agrandie, pour détailler au mieux le profil de ce motif particulier.



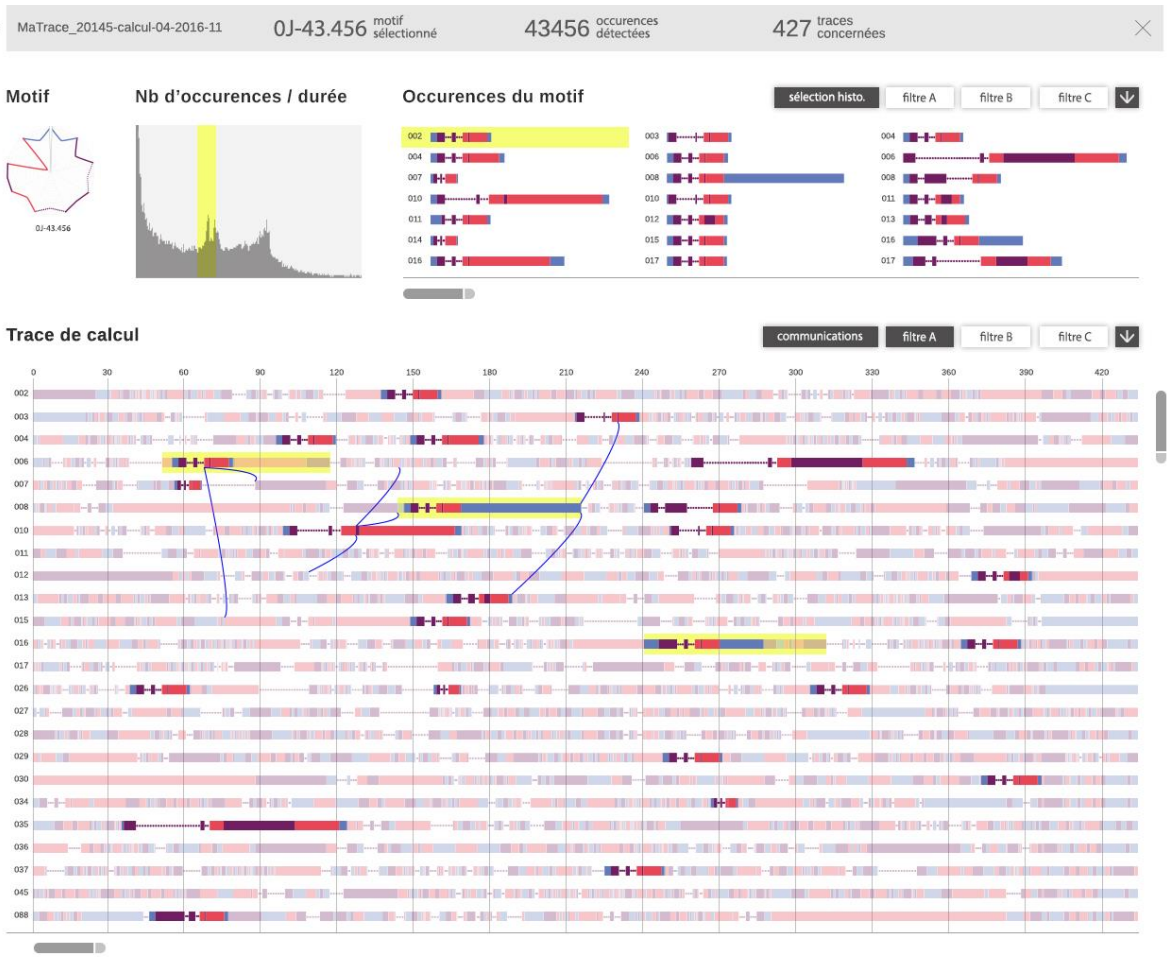
Vous pouvez choisir plusieurs modes de **tri** pour les occurrences du pattern, par taille, processus, date, etc …

3.3 - Sélectionner des occurrences

Quand vous affichez un motif en particulier, vous avez plusieurs possibilités de sélection d'occurrences afin de les faire apparaître en surbrillance dans la trace.

En cliquant sur une occurrence, celle-ci et son occurrence dans la trace apparaissent en **sur brillance**. Les liens affichés à l'écran ne concernent que l'occurrence et le reste de la trace n'ayant aucun lien avec ce pattern sont toujours **opacifiés.**

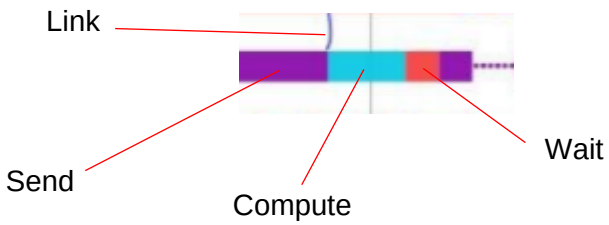
Vous avez aussi accès à un diagramme de répartition du nombre de motifs par intervalle de temps d'exécutions. Ce diagramme se base sur une **échelle logarithmique** afin de s'adapter le plus possible à l'affichage recherché.



Vous pouvez sélectionner une zone sur ce diagramme afin de **mettre en surbrillance** tous les motifs concernés. Ainsi grâce à ces schémas faciles et rapides à lire et à la connaissance que, vous, développeur avez de votre code vous pouvez lier les motifs avec certains morceaux de votre code afin d'en étudier les performances.

3.4 - Lecture d'une occurrence

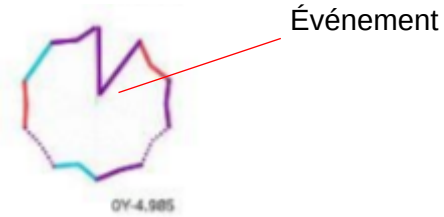
Maintenant que nous avons vu le fonctionnement général de l'affichage, intéressons-nous plus en détail sur les éléments affichés.



Une occurrence est constituée des différents états dans lesquels le thread peut se trouver. Les états sont **: WAIT, SEND, COMPUTE**. Ils sont représentés par les **différentes couleurs** (cf schéma ci-dessus). Vous pouvez savoir à quoi correspondent ces différents états dans le fichier de **configuration states.conf.**

3.5 - Lecture d'un motif / pattern

Un autre élément important à la lecture est le motif. En effet, le principe de FREAD repose sur une **factorisation en motifs** de la trace afin de déduire des comportements globaux du code. On peut ensuite voir à partir de ces comportements globaux les occurrences qui marchent vite ou non.



Le code **couleur reste le même** que pour les occurrences. Le motif se dessine dans un cercle **divisé équitablement** en parts représentants les événements du motif. Il est important de retenir que la taille des parts est **indépendante du temps** de l'action. Plus une part voit **sa ligne s'enfoncée** vers le centre et plus l'événement dur longtemps. L'enfoncement est calculé selon une **échelle logarithmique**.

3.6 - Diagramme de répartition

L'utilisateur a aussi accès à un diagramme de répartition du nombre de motifs par intervalle de temps d'exécutions. Ce diagramme se base sur une **échelle logarithmique** afin de s'adapter le plus possible à l'affichage recherché.

4 - Implémentation d'un parseur

Pour le moment, l'implémentation générale d'un parseur est définie.Cependant, des changements risquent d'être à prévoir. Dans cette section, nous allons nous appuyer en grande partie sur le parseur déjà existant gérant le langage PAJE.

4.1 – Structure de base

La partie contenue dans **parser.hpp** permet de **normaliser les communications** avec le core. C'est la classe parser qui gère les communications. De plus elle contient les fonctions qui permettent d'envoyer des messages. Afin de changer de parseur il vous faut aller dans **parser_specification.hpp** et changer la ligne :

```
#define PARSER paje
```

Il vous faut aussi ajouter l'include de votre parser. De plus, il est important de comprendre que le define permet ici de **changer le namespace** utilisé pour appeler les fonctions du parseur. En changeant le nom du namespace, vous dites alors au compilateur d'utiliser votre parseur **lors de la compilation**. En effet le changement de parseur ne se fait **pas dynamiquement.**

La structure globale du parseur est laissée au choix du développeur en fonction du format de la trace qui sera traitée. Sachant que les informations précédemment énoncées impliquent que le code du parseur doit se **trouver dans un namespace** propre à celui-ci.

4.2 - Fonctions du core vers le parseur

Ces fonctions sont utilisées par la classe parser dans parser.hpp pour faire suivre les requêtes au parseur. L'implémentation de ces fonctions est laissée à la responsabilité du développeur.

4.2.1 - Awake

Le prototype complet est :

```
void awake (std::string const& path);
```

Elle est appelée lorsque le **thread du parseur se lance**. Elle passe en paramètre l'emplacement du fichier de la trace. Elle permet au parseur de se lancer, de commencer sa configuration si nécessaire et de créer son contexte. Si des messages doivent être envoyés au core cela sera **fait dans le start**.

4.2.2 - Start

Le prototype complet est :

```
bool start();
```

Cette fonction est appelée lorsque le message **START** envoyé par le core est reçu. Cela signifie que le core est maintenant en capacité de recevoir des messages. Dans la fonction start le **parseur finit de s'initialiser** et envoie les containers et motifs de la trace au core afin qu'il puisse les stocker et gérer les premières étapes du stockage. Elle renvoie bool pour signifier que l'initialisation est terminée, ce qui permet d'envoyer le **message INITDONE** au core.

Le parseur n'a pas à envoyer les occurrences ici. Seulement des **informations générales** sur la forme de la trace sont envoyées **(Containers et Patterns).** De plus il est préférable que les messages de containers soient **envoyés avant** les patterns. Pour envoyer les messages vous pouvez utiliser les fonctions de la partie 4.3.

4.2.3 - getEventsBetweenTwoTimesInContainer

Le prototype complet est :

```
void getEventsBetweenTwoTimesInContainer(int container_id, float t_begin, float t_end);
```

Cette fonction permet de demander toutes les **occurrences moyennes** comprises entre ces deux dates dans le container « container_id ». Elles seront envoyées au core avec la fonction **sendOccurrenceToCore** en précisant que ce sont des occurrences moyennes et non pas l'occurrence précise d'un pattern.

4.2.4 - getOccurrenceAccuracy

Cette fonction n'est **pas encore complètement définie**. Cela viendra dans une version future. Elle devrait permettre de récupérer une occurrence précise d'un pattern à partir de l'id du pattern et l'id de l'occurrence.

4.3 - Fonctions du parseur vers le core

Ces fonctions sont utilisées par la classe parser dans parser.hpp pour faire suivre les messages du parseur vers le core. L'implémentation de ces fonctions est déjà faite. Pour utiliser les types utilisés en paramètre de ces fonctions, il vous faudra include les fichiers d'en-tête **FOccurrences.hpp, FPattern.hpp, FContainer.hpp,** …dans votre parseur.

4.3.1 - sendContainerToCore

Le prototype complet est :

```
void sendContainerToCore(FContainer const& container);
```

Elle envoie un container au core afin qu'il gère le stockage. Cette fonction ne doit être appelée que durant **la partie start** de l'initialisation. Dans le cas contraire le message risque de ne pas être traitée et d'introduire des erreurs.

4.3.2 - sendPatternToCore

Le prototype complet est :

```
void sendPatternToCore(FPattern const& pattern);
```

Elle envoie un container au core afin qu'il gère le stockage. Cette fonction ne doit être appelée que durant **la partie start** de l'initialisation. Dans le cas contraire le message risque de ne pas être traitée et d'introduire des erreurs.

4.3.3 - sendOccurrenceToCore

Le prototype complet est :

```
void sendPatternToCore(FPattern const& pattern);
```

Elle envoie une occurrence au core afin qu'il puisse la stocker et la renvoyer au render pour l'affichage. L'occurrence contient **l'id du pattern et l'id de l'occurrence** si c'est seulement une occurrence moyenne. Si c'est une occurrence précise, elle contient **aussi les timestamps** des événements du pattern correspondant. Si les événements sont indépendants de la notion de pattern alors la **liste des événements** est aussi comprise dans l'occurrence. Cette fonction est la **réponse classique** à une requête du core.

5 - Bibliographie et références

5.1 - Liens

FREAD repository : https://github.com/Terag/FREAD

Langage PAJE : http://paje.sourceforge.net/download/publication/lang-paje.pdf

EZTrace : http://eztrace.gforge.inria.fr/

ViTE 1.2 : http://vite.gforge.inria.fr/traceGeneration.php

Code des queues: C++ Concurrency in Action Practical Multithreading - *Anthony Williams*

SFML: SFML essentials – *Milcho G.Milchev*

5.2 - Licence

La licence est une licence BSD.