UNIVERSITÉ DU QUÉBEC À MONTRÉAL

PpFf : UNE BIBLIOTHÈQUE C++ POUR LE TRAITEMENT PARALLÈLE DE FLUX DE DONNÉES

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

IULIAN CIOBANU

JANVIER 2021

REMERCIEMENTS

Tout grand projet est rarement réalisé par une seule personne. Celui-ci ne fait pas exception. Je suis reconnaissant à tous ceux et celles qui m'ont aidé et soutenu pendant ces années de travail.

Tout d'abord, je voudrais remercier mon directeur de recherche, le professeur Guy Tremblay. Sans son soutien et son aide, ce mémoire n'aurait pas pu être terminé. Grâce à sa riche expèrience, il m'a guidé dans mes recherches. Il m'a donné de nombreuses idées, suggestions et critiques constructives. Je le remercie aussi pour son support financier au cours de l'hiver et de l'été 2020.

Je dois également un grand merci au professeur Marco Aldinucci, de l'Université de Turin, pour son soutien technique pour l'application de `FastFlow`.

Je souhaite aussi remercier les analystes et techniciens systèmes des laboratoires informatiques et le personnel administratif qui ont permis à mon mémoire de se dérouler sans obstacle technique ou administratif.

Enfin, je tiens à remercier à ma famille, qui a toujours été là, m'ayant aidé moralement à surmonter les difficultés que j'ai eues durant ces années de recherche.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES LISTINGS

RÉSUMÉ

Les applications de traitement de flux sont utilisées pour traiter et analyser les données qui arrivent de façon continue provenant de sources différentes. Celles-ci incluent des applications de sécurité, des applications informatiques générant des capteurs, divers types d'applications de surveillance, des applications du domaine de la finance, de la gestion de réseau informatique et des télécommunications. Ces applications sont, dans de nombreux cas, complexes. Leur complexité augmente encore plus lorsque les données doivent être traitées en parallèle.

Afin de traiter de façon simple et efficace les flux de données, ce mémoire propose `PpFf`, une bibliothèque `C++` avec une *API* simple, de style fonctionnelle, fondée sur une approche «diviser-pour-régner» mais non récursive, qui permet de traiter des données en flux incrémental, mais aussi des collections en lot (*batch*). `PpFf` permet aussi aux programmeurs d'exposer facilement le parallélisme dans des applications de traitement de données — autant du parallélisme de flux que du parallélisme de données — et ce en obtenant des performances intéressantes, ce qui est possible grace à une mise en œuvre qui utilise la bibliothèque `FastFlow`, une bibliothèque de bas niveau de traitement de flux de données en `C++`.

Mots clés : Programmation parallèle, flux de données, traitement de flux, `FastFlow`.

INTRODUCTION

Le traitement de flux de données devient de plus en plus important en raison de la grande quantité de données continuellement générées, provenant de diverses sources telles que des capteurs, des indicateurs boursiers, des dispositifs de réseau, etc. Afin de traiter rapidement une grande quantité de données, notamment en exploitant les capacités des processeurs multicœurs modernes, une application doit être conçue en parallèle. La conception et la mise en œuvre d'applications parallèles efficientes pour traiter des flux de données posent des défis aux développeurs. Les coûts de communications (Amarasinghe *et al.*, 2011), les conditions de course (*data race*) (Wu *et al.*, 2015), les interblocages (Haque, 2006) et les déséquilibres de charge de travail entre les fils d'exécution (Amarasinghe *et al.*, 2011) sont quelques exemples de problèmes qui demandent des efforts supplémentaires aux programmeurs. De plus, la complexité du code peut diminuer la productivité et, par conséquent, augmenter les coûts de développement. Ce mémoire vise à proposer une bibliothèque, en C++, qui permet traiter de façon simple les flux de données en tentant de dissimuler cette complexité.

0.1 Définition du problème

Une application qui traite un flux de données peut être considérée comme un pipeline à travers lequel les données du flux sont produites, traitées et consommées en continu. Le parallélisme dans le traitement d'un tel flux consiste à traiter des éléments distinct du flux de façon concurrente — *parallélisme de flux* (*flow parallelism* ou *stream parallelism*) — et à répliquer un même opérateur sur des sous-groupes d'éléments du flux — *parallélisme de données* (*data parallelism*). La transformation, le tri ou la réduction par un opérateur binaire sont des exemples d'opérations pour lesquelles du parallélisme de données peut être exploité.

Les applications parallèles sont généralement codées à l'aide de bibliothèques comme `FastFlow` (Aldinucci *et al.*, 2014) ou `TBB` (Reinders, 2007). Ces outils permettent aux utilisateurs d'implémenter des solutions robustes et portables avec une abstraction de haut niveau qui masque la complexité des mécanismes de concurrence, tels que la gestion des *threads* et les synchronisations. Bien que les modèles offerts par ces outils aient pour but de simplifier le développement d'applications parallèles, ils n'offrent pas une interface standard. Notre bibliothèque veut introduire un modèle simple, où le programmeur crée un pipeline par l'intermédiaire d'une interface (API, *Application Programming Interface*) qui expose clairement les opérations de transformation, et où chaque transformation est relativement simple — ce qui correspond à une application du principe «diviser-pour-régner» mais de façon *non récursive*.

Réécrire une application est généralement coûteux en termes de temps de développement. Les approches visant à pallier ce défaut sont souvent des outils de programmations parallèles développés pour introduire le parallélisme dans du code séquentiel existant. Des exemples sont `OpenMP` (Chandra *et al.*, 2001) et `OpenACC` (Farber, 2016) qui utilisent une approche basée sur l'ajout de directives — des commentaires spéciaux traités par le compilateur — et `Cilk` (Leiserson et Plaat, 1998) qui est une extension simple du langage `C`. Malheureusement, ces outils ne sont pas bien adaptés au traitement de flux de données.

Ces dernières années, plusieurs bibliothèques écrites en *C++* ont été conçues pour traiter des flux de données. Parmi les plus récentes, on retrouve `RaftLib` (Beard *et al.*, 2017), `StarPU` (Aumage *et al.*, 2018a) et `SkePU` (Ernstsson et Kessler, 2015). Malgré le fait que ces bibliothèques soient des outils performants, aucune d'entre elle ne facilite la description des opérations d'un flux comme le permet le chaînage des opérations.

Des outils qui supportent le traitement de flux de données sont disponibles aussi dans d'autres langages de programmation que `C/C++`. Parmi les plus connus on retrouve `Spark` (Frampton, 2015) (Java, Scala et autres langages), les `Streams` de `Java 8` (Warburton, 2014) et `Flink` (Apache, 2014) (Java et Scala). Notre bibliothèque possède une `API` semblable à celle des `Streams` de `Java` 8, mais en `C++`, et elle est souvent plus simple à spécifier notamment grâce aux *templates*.

0.2 Objectifs

Dans le contexte du langage de programmation `C++`, il existe plusieurs bibliothèques qui offrent des algorithmes de traitement de flux de données. Cependant, plusieurs des constructions pour exprimer des algorithmes parallèles se limitent à des opérations de transformation et et de réduction. Ce mémoire a comme objectif d'enrichir ces opérations avec de nouvelles opérations, et ce dans une interface simple à utiliser. Ceci, associé à de nouvelles fonctionnalités telles que les expressions lambda (Josuttis, 2012), aide un programmeur à écrire des opérations complexes pour un flux de données. Cette nouvelle bibliothèque en `C++`, appelée `PpFF`, est mise en œuvre avec la bibliothèque `FastFlow`.

Étant donné la ressemblance avec les `Streams` de `Java 8`, les performances de `PpFf` seront évaluées en les comparant à celles des `Streams`. Comme nous le verrons, les résultats indiquent que `PpFf` peut en effet traiter des données à haut débit. Nous comparerons aussi les performances de `PpFf` avec celles de `FastFlow`, afin d'analyser les surcoûts introduits par rapport à `FastFlow`. En plus de mesurer les performances de notre bibliothèque, nous illustrerons également son expressivité en implémentant certains cas d'utilisation typiques rencontrés dans les applications de traitement de flux de données.

0.3 Organisation du mémoire

Les chapitres qui forment le cœur du mémoire sont organisés comme suit.

Le chapitre 1 Logiciels et bibliothèques de traitement de flux de données : Spark, Java, FastFlow et autres bibliothèques C++ présente des outils existants portant sur le traitement des flux de données. Tout d'abord, il introduit les architectures utilisées par divers outils, puis il présente les modèles de programmations permettant d'exprimer les traitements de données.

Le chapitre 2 Description de l'API de PpFf présente l'API de notre bibliothèque. Plus précisément, le chapitre présente les méthodes les plus importantes de `PpFf` à l'aide de quelques exemples. À des fins de comparaison, certains de ces exemples sont aussi

présentés en `Java` et en `C++` avec `FastFlow`. Un résumé sous forme de tableau de toutes les méthodes implémentées est aussi présenté en annexe.

Le chapitre 3 Mise en œuvre de PpFf explique comment nous avons mis en œuvre notre bibliothèque en utilisant la bibliothèque de bas niveau `FastFlow`.

Finalement, le chapitre 4 Évaluation des performances : Comparaisons de `PpFf` avec Java et FastFlow présente une évaluation des performances sur deux cas d'utilisation typiques.

CHAPITRE I

LOGICIELS ET BIBLIOTHÈQUES DE TRAITEMENT DE FLUX DE DONNÉES : SPARK, JAVA, FASTFLOW ET AUTRES BIBLIOTHÈQUES C++

La programmation parallèle devient de plus en plus une nécessité avec l'apparition des architectures multicœurs. Jusqu'à il y a quelques années, une des principales façons d'exécuter un programme plus rapidement était grâce à l'augmentation de la vitesse d'horloge du processeur. De nos jours, la vitesse d'horloge a atteint ses limites et les performances d'un programme ne peuvent souvent être améliorées que si on l'exécute en parallèle.

Il existe de nombreuses approches et de nombreux langages de programmation parallèle. Le paradigme de programmation parallèle *par flux de données* offre une approche prometteuse pour la programmation de systèmes multicœurs. C'est cette approche que nous allons brièvement décrire avant de présenter quelques langages existants qui mettent en œuvre cette approche.

1.1 Parallélisme de flux de données

Un *flux de données* est une suite potentiellement infinie de données, générées par différentes sources, qui peut être traitée de façon incrémentale par des processus interconnectés. Typiquement, le traitement des données d'un flux s'effectue à l'aide d'une séquence d'opérations — un *pipeline* d'opérations. Les opérations typiques de traitement de flux des données comprennent notamment le tri, le filtrage, la transformation, l'accumulation.

Lorsque ces opérations sont exécutées par différents *threads*, on parle alors de *parallélisme de flux de données*. Le traitement complet d'une suite d'éléments de donnée se fait, conceptuellement, en faisant migrer d'une opération à une autre les divers éléments, dans l'ordre dans lequel ils sont émis par les sources des données. Un élément de donnée doit donc parcourir plusieurs étapes de traitement, mais divers éléments peuvent être en cours de traitement à un instant donné, d'où le parallélisme.

Un tel traitement en pipeline peut aussi être utilisé pour des collections *finies* de données. Connu comme le modèle de traitement par lots (*batch processing*), le traitement de telles collections peut se faire en traitant chaque collection comme un tout, ou se faire élément par élément à travers un pipeline d'opérations et avec du parallélisme de flux de données.

Il existe de nombreux systèmes de traitement de flux de données. La conception, le modèle et l'architecture de ces systèmes diffèrent, de sorte qu'ils possèdent des fonctionnalités et des performances différentes. Ce chapitre présente un aperçu de trois systèmes de traitement de flux de données : `Apache Spark` (Sect. 1.2), les `Streams` de `Java` 8 (Sect. 1.3) et `FastFlow` (Sect. 1.4). Ce sont trois parmi les nombreux systèmes actuellement disponibles, que nous avons choisi de présenter parce qu'ils ont servi d'inspiration (Spark et `Streams` de Java) et de base (`FastFlow`) au système que nous avons développé, `PpFf`, que nous présenterons aux chapitres 2 et 3. Ce chapitre se termine par la présentation d'autres bibliothèques `C++` plus récentes pour traiter des flux de données : `RaftLib` (Sect. 1.5.1), `StarPU` (Sect. 1.5.2) et `SkePU` (Sect. 1.5.3).

1.2 Apache Spark

`Apache Spark` (Apache, 2012) est un engin de traitement de flux de données doté d'une API expressive qui permet aux développeurs d'exécuter efficacement plusieurs opérations sur une collection de données. `Spark` est un outil de traitement par lots, mais qui a aussi des capacités de traitement de flux incrémental de données. `Spark`, comme les *streams* de `Java`, utilise la chaîne de méthodes, dans un style fonctionnel, et fournir de nombreuses méthodes. Apparu avant même les *stream* de Java 8, Spark se concentre principalement

sur l'accélération du traitement des données en gardant en mémoire les données de travail, ce qui permet notamment l'accélération du traitement des algorithmes itératifs.

Alors que le traitement en mémoire contribue considérablement à obtenir de hautes performances, Spark est également rapide avec le traitement de données sur disques en raison de l'optimisation qui peut être réalisée en analysant l'ensemble complet des tâches à l'avance. Ceci peut être réalisé en créant des graphes acycliques dirigés (DAG) qui représentent toutes les opérations qui doivent être exécutées, les données à être utilisées, ainsi que les relations entre elles. À l'aide de ce graphe, le système d'exécution a la capacité de planifier et d'ordonnancer le travail de façon efficace.

`Spark` utilise un modèle appelé *Resilient Distributed Datasets* (`RDD`) (Salloum *et al.*, 2016) pour représenter les collections de données, modèle qui gère efficacement la persistence lorsque les données ne peuvent pas toutes être traitées en mémoire ou lorsqu'elles doivent être préservées pour éviter des recalculs. De plus, `Spark` implémente le modèle *Discretized Streams* (Zaharia *et al.*, 2013) pour le traitement de flux incrémental de données. Nous expliquons ces deux modèles dans les sous-sections qui suivent.

1.2.1 *Resilient Distributed Dataset* (RDD)

Un `RDD` est une abstraction de données en mémoire qui évite la réplication de données et minimise les accès au disque. L'utilisation de `RDD`s permet aux applications de mettre en cache des données à travers différentes étapes de traitement, ce qui accélère considérablement la réutilisation pour les traitements futurs. De plus, les `RDD`s mémorisent les opérations utilisées pour les construire et utilisent un mécanisme de *checkpoint*. Lorsqu'une panne survient, les `RDD`s requis peuvent être reconstruits, et ce sans devoir tout refaire les calculs.

Les `RDD`s sont conçus pour être partitionnés, donc répartis sur différentes machines. Chaque partition contient des enregistrements qui peuvent être créés par des opérations de transformations. Les transformations incluent notamment les opérations `map`, `filter`, `groupBy` et `join`.

Listing 1.1: Un segment de code Spark/Java pour compter le nombre d'occurrences de mots. Exemple provenant du site Web pour Apache Spark (https://spark.apache.org/examples.html).

```
JavaRDD<String> textFile =
    sc
    .textFile("hdfs://...");

JavaPairRDD<String, Integer> counts =
    textFile
    .flatMap( s -> Arrays.asList(s.split(" ")).iterator() )
    .mapToPair( word -> new Tuple2<>(word, 1) )
    .reduceByKey( (a, b) -> a + b );

counts
    .saveAsTextFile("hdfs://...");
```

1.2.2 *Discretized Streams*

Spark traite un flux incrémental de données à l'aide du modèle *Discretized Streams* — D-streams. Ce modèle est une autre abstraction introduite en Spark. Spark décompose un flux de données en une série de lots, en fonction de courts intervalles de temps, lots appelés *micro-batches*. Chaque *micro-batch* stocke ses données dans un RDD. Ensuite, la *micro-batch* est traitée et ses résultats sont stockés dans des RDDs intermédiaires.

1.2.3 Exemple : wordCount

Afin d'illustrer le modèle de programmation avec les RDDs de Spark, le listing 1.1 montre le code source d'un segment de code de décompte des mots, code écrit en Java.[1] Plus précisément, ce segment de code compte le nombre d'occurrences des divers mots dans un fichier texte et est composé de plusieurs opérations :

1. https://spark.apache.org/examples.html

— `textFile`, qui lit les lignes à partir d'un fichier. Le fichier est identifié par le paramètre fourni en argument à `textFile`.
— `flatMap`, qui divise chaque ligne en mots, qui sont ensuite transmis en aval sous forme d'éléments de données individuels.
— `mapToPair`, qui crée une paire (de type `Tuple2`) avec une clé indiquant le mot et une valeur associée égale à 1.
— `reduceByKey`, qui regroupe les mots similaires ensemble et compte le nombre d'occurrences de chaque mot.

1.3 Streams de Java 8

La version 8 de `Java`, qui a introduit les `Streams`, a changé la façon dont les développeurs peuvent traiter, tant de façon séquentielle que parallèle, les collections de données. La manipulation de collections de données à l'aide de `Streams` ressemble à un langage de requêtes de base de données. Au lieu de parcourir explicitement les données à l'aide de boucles (`for`), un développeur spécifie simplement une suite d'opérations *fonctionnelles* à effectuer sur les éléments d'une collection. Urma et al. (2014) fournissent une description détaillée des nouveaux concepts introduits dans `Java 8`. Les `Streams` et les *expressions lambdas* sont les fonctionnalités les plus remarquables ajoutées dans l'API (Oracle-Documentation, 2018). Ils sont conçus pour traiter les collections de données de manière simple et efficace. Les sous-sections qui suivent décrivent quelques-uns des concepts de `Java 8`.

1.3.1 Expressions lambdas

Une *expression lambda* — appelée aussi *fonction anonyme* — est un bloc de code avec des paramètres qui peut être défini, comme n'importe quelle valeur, puis qui peut être exécuté ultérieurement. Par exemple, la fonction anonyme du listing 1.2, qui reçoit deux valeurs de type entier en argument et renvoie leur somme, est affectée à la variable `a`. Un appel à la méthode `add` — de l'interface `Addable` — est ensuite effectué avec des arguments effectifs (`x` et `y`).

Listing 1.2: Un exemple d'une expression lambda qui reçoit deux valeurs de type entier en argument, affectée à la variable `a`. Un appel est ensuite effectué et le résultat est affecté à la variable `result`.

```
interface Addable { int add (int x, int y); }

int x = 3;
int y = 2;

// Affectation de l'expression lambda a une variable.
Addable a = (int w, int z) -> { return w + z; };

// Appel de l'expression lambda.
int result = a.add(x, y);
```

Listing 1.3: Un exemple de remplacement d'une classe anonyme par une expression lambda.

```
// Specification d'un thread avec une classe interne anonyme.
Thread th = new Thread( new Runnable() {
    public void run() {
      ... // Code a executer par le thread.
    }
});

// Specification d'un thread avec une expression lambda.
Thread th = new Thread( () -> {
     ... // Code a executer par le thread.
});
```

Le concept d'expression lambda n'est pas nouveau. Il est utilisé depuis longtemps dans les langages de programmation fonctionnelle tels que `Lisp` (Steele, 1984) ou `Haskell` (Hudak et Wadler, 1990; Hutton, 2016). Les expressions lambda rendent le code plus concis et, dans le cas de `Java`, permettent de l'étendre avec des concepts de langages de programmation fonctionnelle.

Plus spécifiquement, en `Java`, le concept d'expression lambda est lié à celui d'interface fonctionnelle (*functional interface*). Une expression lambda peut être spécifiée à la place d'une valeur dont le type est une interface fonctionnelle. Par exemple, le listing 1.3 montre un exemple où un `Thread th`, déclaré en utilisant la syntaxe de classe anonyme, peut être écrit plus facilement avec une expression lambda.

Dans une expression lambda, lors de la compilation, les types des arguments peuvent être automatiquement déterminés par le compilateur. Cette fonctionnalité permet de passer des méthodes comme arguments plutôt que de construire un objet d'une classe spécifique. Ceci permet à un programmeur de construire facilement des pipelines d'opérations fonctionnelles.

1.3.2 Itérations

Une itération est le processus qui consiste à traverser une collection d'éléments pour traiter chacun d'entre eux. Avec les `Streams` `Java`, une itération peut être exécutée de deux manières : par une itération *externe* ou *interne*.

Une itération est dite *externe* lorsque le développeur contrôle la traversée des éléments. L'accès et l'opération sur chaque élément de la collection sont définis par l'utilisateur. Par contre, une itération est dite *interne* lorsque la collection contrôle elle-même tous les détails du processus d'itération. L'utilisateur fournit uniquement les opérations permettant de traiter les éléments, sans se soucier de la manière dont les éléments sont accédés et fournis. Le listing 1.4 montre un exemple d'une comparaison entre une itération externe et interne.

Listing 1.4: Un exemple comparant une itération externe et interne. Dans le cas de l'itération externe, le développeur définit une boucle explicite pour traiter les éléments de la collection. Le traitement appliqué à chaque élément consiste à convertir la chaîne en lettres majuscules si elle débute par la lettre «J». Le même traitement est appliqué dans le cas de l'itération interne. Par contre, dans ce cas, c'est le `Stream` — produit par l'appel `myList.stream()` — qui contrôle le processus d'itération ; l'usager indique simplement les opérations à effectuer, en les chainant les unes à la suite des autres.

```
List<String> myList =
        Arrays.asList("Tom", "John", "Harry", "Jonathan");

// Iteration externe avec ajout explicite (style imperatif).
List<String> resultExtern = new ArrayList<String>();
for (String s: myList){
    if (s.startsWith("J")) // Selection
       // Transformation et ajout dans le resultat.
       resultExtern.add(s.toUpperCase());
}

// Iteration interne avec pipeline d'operations (style fonctionnel).
List<String> resultIntern =
    myList
    .stream()
    .filter(s -> s.startsWith("J")) // Selection.
    .map(String::toUpperCase)       // Transformation.
    .collect(Collectors.toList());  // Collecte du resultat.
```

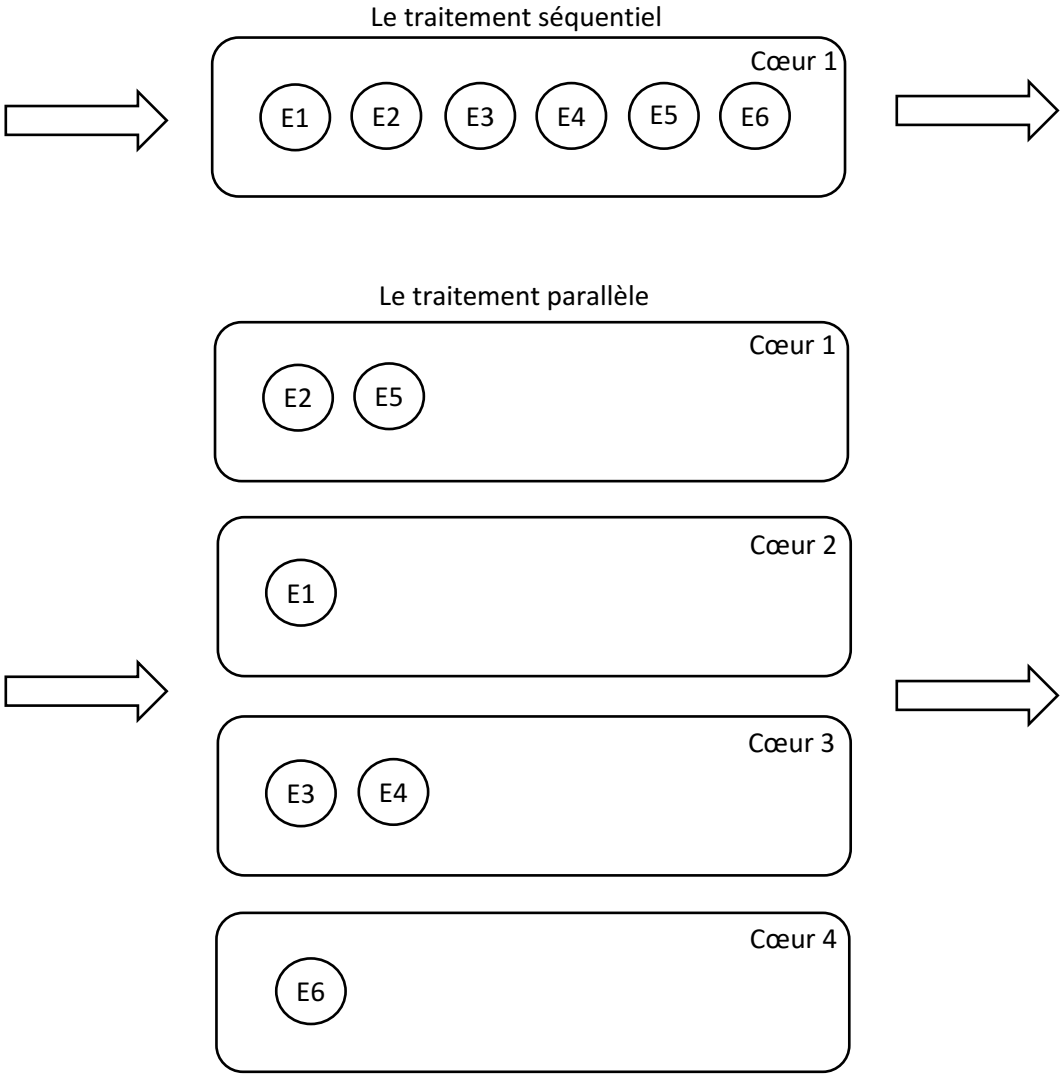

Figure 1.1: Une comparaison entre traitement séquentiel et parallèle. Les six éléments du flux (`E1`, ..., `E6`) sont répartis dans quatre sous-flux. Chaque sous-flux est traité par l'un de quatre cœurs disponibles. Les résultats sont combinés après le traitement.

Listing 1.5: Un exemple d'optimisation du traitement d'un flux en utilisant la technique d'évaluation court-circuitée.

```
List<Employee> employees;
Optional<Employee> employee = employees.findFirst();
```

L'approche interne est attrayante pour les opportunités qu'elle offre aux compilateurs, notamment l'optimisation de l'exécution et les mécanismes de nettoyage nécessaires en arrière-plan. Un autre avantage des itérations internes est l'efficacité, le traitement pouvant être réparti entre les cœurs de la machine pour une exécution parallèle.

Lorsqu'un flux s'exécute en parallèle, `Java` partitionne le flux en plusieurs sous-flux. Les opérations parcourent et traitent ces sous-flux en parallèle, puis combinent les résultats. La figure 1.1 illustre la comparaison entre un traitement séquentiel et un traitement parallèle, dans cet exemple avec quatre cœurs. Les éléments du flux sont partitionnés en sous-flux et chaque sous-flux résultant est traité par l'un de quatre cœurs disponibles.

1.3.3 Flux

Un flux est défini comme une séquence immuable d'éléments fournissant une variété d'opérations et de méthodes permettant de traiter une collection de données. Le flux prend en charge les opérations d'agrégation (Oracle, 2017) séquentielles et parallèles sans se soucier de la manière dont les éléments sont stockés ou accessibles. Pour effectuer un traitement, les opérations de flux sont composées dans un `pipeline`. Un `pipeline` est composé d'une source, de zéro ou plusieurs opérations intermédiaires et d'une opération terminale. Une source peut être constituée d'une collection ou de tout objet implémentant l'interface qui définit le mécanisme permettant d'extraire les données de la source. Les opérations sur les flux adoptent un mécanisme d'évaluation paresseuse. L'évaluation paresseuse est une méthode d'optimisation du traitement qui retarde l'étape du calcul jusqu'à ce que le résultat soit nécessaire. En `Java`, le traitement sur les éléments du flux est réalisé seulement à l'activation de l'opération finale et les éléments source ne

sont consommés qu'au besoin. Cela permet au compilateur d'optimiser le traitement des données dans le `pipeline`. Une autre technique d'optimisation utilisée par `Java` est l'évaluation court-circuitée (*short-circuiting* en anglais). Dans un flux, une telle évaluation a pour effet que seuls les éléments nécessaires sont consommés. [2] Par exemple, dans le listing 1.5, l'opérateur `findFirst` retourne le premier employé trouvé dans une liste d'employés ; les éléments restants du flux sont ignorés.

L'un des principaux avantages des flux est qu'ils peuvent être évalués soit de façon séquentielle, soit en parallèle. L'évaluation séquentielle est réalisée en exécutant toutes les opérations en `pipeline` sur chaque élément du flux. Lorsqu'un flux est évalué en parallèle, il utilise un type spécial d'itérateur appelé `Spliterator`. Ce dernier partitionne le flux de manière récursive en se divisant lui-même pour créer des flux enfants. Ce mécanisme permet aux *threads* de traverser plusieurs flux en parallèle. Les *threads* sont gérés par un groupe de *threads* de type `fork/join`.

1.3.4 *Threads* de type *fork/join*

Introduit en `Java 7`, le *framework* `fork/join` permet aux développeurs de spécifier des tâches pouvant être subdivisées et exécutées en parallèle sur des machines multicœurs. Il est basé sur deux opérations : `fork` et `join`. L'opération `fork` a pour rôle de diviser une tâche en plus petites sous-tâches indépendantes, et ce récursivement jusqu'à ce qu'elles soient assez simples pour être exécutées de manière indépendante et asynchrone. L'opération `join` a pour rôle de fusionner les résultats de toutes les sous-tâches de manière récursive en un seul résultat. Les sous-tâches obtenues par l'opération `fork` sont soumises à un `ForkJoinPool`. Ce dernier est composé d'un ensemble de travailleurs. Le nombre de travailleurs dans un `ForkJoinPool` est généralement limité par le nombre de cœurs de la machine. Chaque travailleur peut exécuter une tâche à la fois. Les tâches en attente d'exécution sont stockées dans une file appartenant à un travailleur. Une tâche

2. L'évaluation court-circuitée peut aussi être vue comme une forme d'évaluation paresseuse, bien que la documentation `Java` distingue ces deux formes d'optimisation.

en cours d'exécution peut générer de nouvelles tâches, qui sont ensuite mises en file pour une exécution ultérieure. Lorsqu'un travailleur a terminé l'exécution d'une tâche et qu'il n'a plus aucune tâche dans sa propre file, il essaie de prendre une tâche d'un file d'un autre travailleur à l'aide d'un algorithme de vol de travail (*work stealing algorithm* (Frigo *et al.*, 1998)). Cet algorithme permet un équilibrage efficace de la charge de travail entre les divers travailleurs.

1.3.5 Exemple : wordCount

Afin d'illustrer le modèle de programmation avec les `Streams` de `Java 8`, le listing 1.6 montre le code source d'un segment de code `Java` de décompte de mots. Plus précisément, ce segment de code compte le nombre d'occurrences des mots dans un chaine et est composé de plusieurs opérations *chaînées* les unes à la suite des autres :

— `lines`, qui renvoie un flux séquentiel de lignes à partir du fichier. Le fichier est repéré par le paramètre `inputFile` fourni en argument.
— `parallel`, qui marque le flux en tant que flux parallèle. Cette opération permet ainsi de partitionner et d'exécuter le `pipeline` en parallèle.
— `flatMap`, qui divise chaque ligne en mots qui sont ensuite transmis en aval sous forme d'éléments de données individuels.
— `map`, qui supprime tous les caractères qui ne sont pas des lettres puis transforme les lettres majuscules du mot en minuscules.
— `filter`, sélectionne dans le flux seulement les mots qui ne sont pas vides.
— `map`, qui crée une paire (de type `Entry`) avec une clé représentée par le mot et une valeur associée égale à 1.
— `collect`, collecte les éléments dans un `Map` et additionne le nombre d'occurrences de chacun des mots à l'aide de l'expression lambda.
— `entrySet`, renvoie un flux de type clé–valeur. La clé est le mot et la valeur est son nombre d'occurrences dans le fichier.
— `stream`, crée un nouveau flux de données à partir de l'ensemble de paires.
— Finalement `collect`, qui combine tous les éléments dans une liste.

Listing 1.6: Un pipeline Java pour compter le nombre d'occurrences de mots.

```
public static void main(String[] args) throws IOException {
  // Le texte a analyser, sous forme d'une chaine.
  String text = "Lorem ipsum dolor sit amet, consectetur \n" +
     " adipiscing elit. Lorem ipsum dolor sit amet, consectetur \n" +
     " adipiscing elit. Lorem ipsum dolor sit amet.";

  // Le texte a analyser decompose en une liste de lignes.
  ArrayList<String> lines =
    new ArrayList<String>(Arrays.asList(text.split("\\n")));

  // Le pipeline qui traite la liste de lignes.
  List<Map.Entry<String,Integer>> wordsCount =
    lines
    .stream()
    .parallel()
    .flatMap(line->Arrays.stream(line.trim().split(" ")))
    .map(word->word.replaceAll("[^a-zA-Z]", "").toLowerCase())
    .filter(word->word.length() > 0)
    .map(word->new SimpleEntry<>(word, 1))
    .collect(toMap(e->e.getKey(), e->e.getValue(), (v1,v2)->v1+v2))
    .entrySet()
    .stream()
    .collect(Collectors.toList());

  wordsCount.forEach( x -> System.out.println("'" + x.getKey() +
                                              "' => " + x.getValue()) );
}
```

Résultat de l'exécution:

```
  'dolor' => 3
  'lorem' => 3
  'amet' => 3
  'adipiscing' => 2
  'ipsum' => 3
  'elit' => 2
  'consectetur' => 2
  'sit' => 3
```

1.4 FastFlow

`FastFlow` est une bibliothèque `C++` qui, à la base, offre un ensemble de mécanismes de bas niveau pour traiter les flux de données et s'exécutant sur des machines multicœurs avec mémoire partagée. La facilité de développement avec `FastFlow` est rendue possible en utilisant un ensemble de *squelettes algorithmiques* offerts par la bibliothèque (Aldinucci *et al.*, 2014). Son efficacité provient principalement de la mise en œuvre optimisée de mécanismes de communication de bas niveau et de sa conception en couches. Les squelettes algorithmiques offerts par `FastFlow` peuvent être utilisés pour exprimer les modèles les plus courants de parallélisme. Ces squelettes algorithmiques peuvent être imbriqués pour créer des modèles hiérarchiques de parallélisme plus complexes.

`FastFlow` a été conçue selon plusieurs principes : conception en couches, efficacité des communications et synchronisation, et support pour le parallélisme de flux.

1.4.1 Conception en couches

`FastFlow` a été conçue sous la forme d'une pile de couches pour permettre d'implémenter des mécanismes d'abstraction et d'optimisation à plusieurs niveaux. La figure 1.2 montre les trois principales couches : *High-level patterns*, *Core patterns* et *Building blocks*.

Le niveau le plus bas de la pile, *Building blocks*, gère les communications asynchrones entre les canaux de communication et gère le cycle de vie des flux. Plus précisément, cette couche offre des services pour les couches supérieures. Elle gère les queues, les processeurs et les fils d'exécutions (*threads*).

Au-dessus de la couche *Building blocks* se trouve la couche *Core patterns*, qui fournit des squelettes (gabarits) pour modéliser différentes formes de parallélisme de flux. Les trois modèles fournis dans cette couche sont *task-farm*, *pipeline* et *feedback*, qui permettent de construire des flux parallèles variés.

Au sommet de la pile se trouve la couche *High-level patterns*, qui permet d'exprimer le parallélisme de plus haut niveau. Elle fournit plusieurs méthodes qui couvrent les paradigmes de programmation parallèles les plus courants : parallélisme de flux, parallélisme de données et parallélisme de tâches.

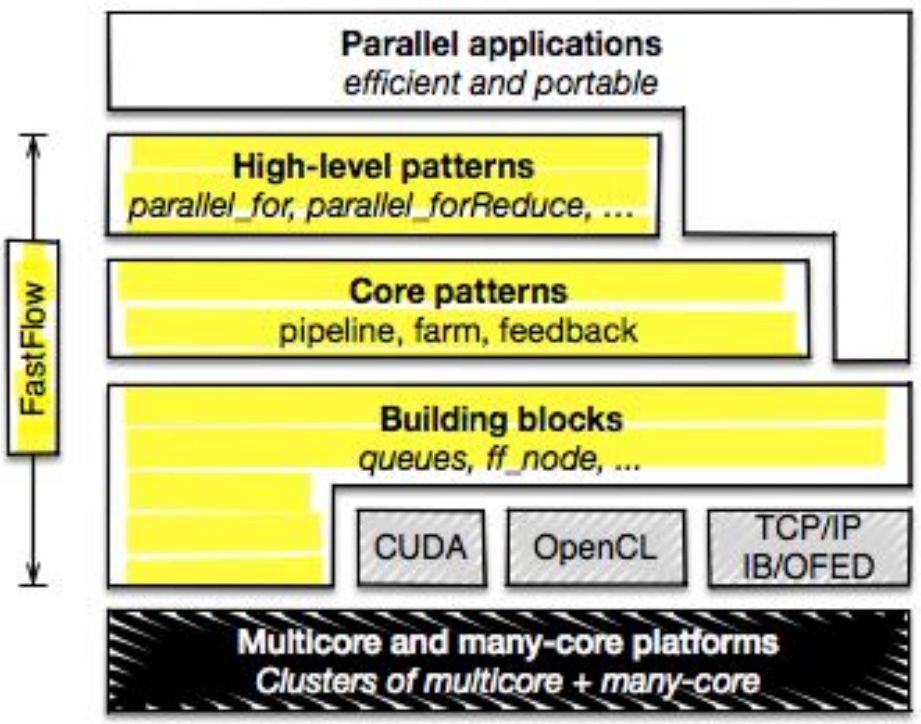

Figure 1.2: Les couches de `FastFlow` — figure tirée de (Torquati, 2015).

1.4.2 Efficacité

L'idée de `FastFlow` est de fournir au programmeur des files (*queues*) *MP (Multiple Producer)* et des files *MC (Multiple Consumer)* sans verrouillage, et ce afin de supporter des accès rapides aux flux de données. Généralement, dans les applications de flux de données, les canaux de communication sont implémentés via des files de type *MPMC (Multiple Producer/Multiple Consumer)*. Dans ce modèle, les acteurs se synchronisent simultanément pour accéder aux données. Ces synchronisations sont habituellement supportées par une ou plusieurs opérations atomiques — par exemple, `Compare-And-Swap` — qui se comportent comme des barrières de mémoire, ce qui augmente le coût des synchronisations. Afin d'éviter les barrières de mémoire, `FastFlow` bâtit les files *MPMC* en assemblant des files, sans barrière de mémoire, de type *SPSC (Single Producer/Single Consumer)*. Dans ce modèle, des files d'exécution regroupent ou émettent les données des files d'entrée vers les files de sortie. Selon son rôle, une file d'exécution peut être un `Emitter` ou un `Collector`. Alors que l'`Emitter` lit les données, le `Collector` écrit sur une ou plusieurs files de types *SPSC*. Contrairement aux opérations atomiques, ce mécanisme nécessite seulement des copies de mémoire — la performance offerte par cette solution découle de la vitesse supérieure de la copie par rapport à la barrière de mémoire.

1.4.3 Parallélisme de flux

Dans `FastFlow`, le parallélisme de flux est représenté par un flux de données comportant une série d'étapes, séquentielles ou parallèles, appelées *stages*. Chaque *stage* lit les données à partir du flux d'entrée, effectue des calculs et traitements, puis écrit le résultat sur le flux de sortie. Le calcul représente une séquence de transformations sur les données. Le parallélisme est obtenu en exécutant chaque *stage* simultanément sur des éléments indépendants ou sur des sous-séquences d'éléments.

Dans `FastFlow`, le parallélisme peut être obtenu en exploitant directement les modèles parallèles disponibles dans la couche *Building blocks*. En particulier, cela peut être réalisé des deux façons suivantes :

— en définissant des activités concurrentes séquentielles en sous-classant la classe `ff_node` ;
— en construisant des modèles parallèles complexes en composant de manière hiérarchique des activités concurrentes séquentielles, soit avec des *pipelines* — `ff_pipeline` — ou des *farms* — `ff_farm`.

La classe `ff_node`

Dans `FastFlow`, `ff_node` est la classe de base pour un noeud de traitement. Elle fournit les mécanismes appropriés pour définir des activités séquentielles pour le traitement de données apparaissant sur un canal d'entrée et fournissant les résultats correspondants sur un canal de sortie.

La classe `ff_node` définit plusieurs méthodes, les trois plus importantes étant les suivantes :

```
virtual void* svc(void* task) = 0
virtual int svc_init() { return 0; }
virtual void svc_end() {}
```

La première méthode, `svc` (mnémonique «service»), est la plus importante car c'est celle qui définit le comportement du nœud lors du traitement des éléments du flux d'entrée. Quant aux deux autres méthodes, elles sont appelées lorsque le traitement représenté par le nœud est démarré `(svc_init)` et juste avant qu'il soit terminé `(svc_end)`. Ces trois méthodes virtuelles peuvent être redéfinies dans des sous-classes `ff_node` spécifiées par le programmeur afin d'implémenter le traitement, l'initialisation ou la finalisation du code.

La classe `ff_pipeline`

Un *pipeline* est utilisé pour modéliser les traitements exprimés par des *stages*. Il est représenté par la classe `ff_pipeline`. Un pipeline peut comporter plusieurs *stages*, peut être construit comme un pipeline à n *stages*, ou encore comme un pipeline de pipelines. Un *stage* peut être de type `ff_node` ou `ff_farm`.

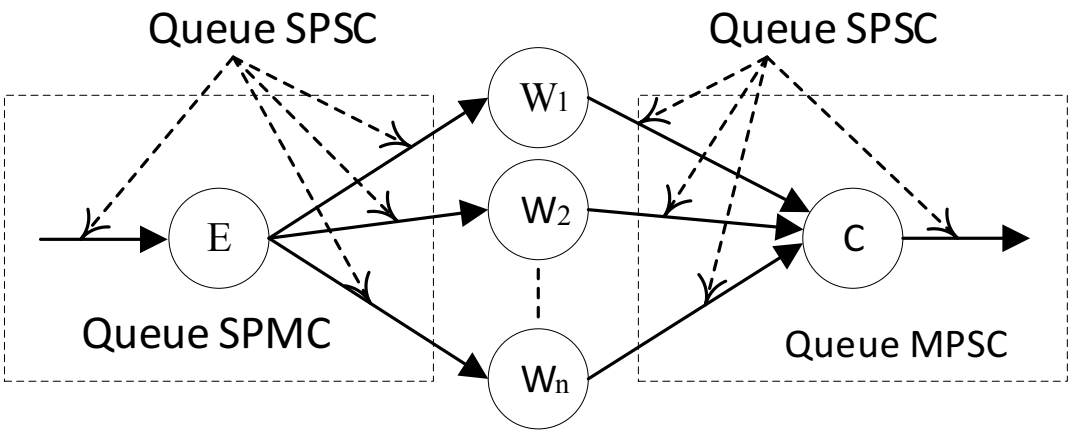

Figure 1.3: Un *farm* de `FastFlow` est composé de trois parties : l'`Emitter (E)`, le *pool* de `workers` (W1... Wn) et le `Collector (C)`. Les canaux de communication sont implémentés en assemblant des files de types *Simple Producer/Simple Consumer (SPSC)* — figure tirée de (Aldinucci *et al.*, 2010).

La classe `ff_farm`

Un *farm* est basé sur la réplication d'une fonction. Dans `FastFlow`, un *farm* est représenté par un objet de la classe `ff_farm`. Comme le montre la figure 1.3, un *farm* est composé de trois parties distinctes : un `Emitter`, un *pool* de `workers` et un `Collector`. L'`Emitter` est responsable de la répartition des éléments du flux au *pool* de `workers`, lesquels traitent et produisent les données de sortie ; l'`Emitter` distribue les éléments d'entrée aux travailleurs selon une certaine politique d'ordonnancement afin d'équilibrer la charge des travailleurs. Les résultats sont ensuite rassemblés par le `Collector` dans un seul flux de sortie. Les communications entre `Emitter` et `Collector` se font via des canaux de communication sans barrières de mémoire, tel que décrit précédemment.

1.4.4 Exemple : WordCount

Cette section présente un exemple d'un programme réalisé en `FastFlow`. Illustré dans le listing 1.7, ce programme compte le nombre d'occurrences des divers mots dans un fichier texte.

Listing 1.7: Le code source `FastFlow` d'une application pour compter le nombre d'occurrences de mots.

```
std::string DEFAULT_INPUT_FILE = "Words.txt";

typedef std::vector<std::string> Words;

struct linesFromFileStage: ff_node {
   std::string const &path;
   linesFromFileStage(std::string const &path): path(path){}

   void* svc(void* task) {
      std::ifstream file(path);
      std::string* line = new std::string;
      while (std::getline(file, *line)) {
          ff_send_out(line);
          line = new std::string;
      }
      return EOS;
   }
};

struct splitInWordsStage: ff_node {
   std::string delimiter = " ";
   void* svc(void* task) {
      std::string line = *((std::string*)task);
      Words* words = new Words();
      size_t start = 0, end = 0;
      do {
          end = line.find(delimiter, start);
          size_t len = end - start;
          words->push_back( line.substr(start, len) );
          start += len + delimiter.length();
      } while (end != std::string::npos);

      return words;
   }
};
```

```
struct flatStage: ff_node {
   std::string delimiter = " ";
   void* svc(void* task) {
       for (auto &elem: *(Words*)task) {
           ff_send_out(&elem);
       }
       return GO_ON;
   }
};

struct groupByKeyStage: ff_node {
   typedef std::unordered_map<std::string, int> CONTAINER;
   CONTAINER &container;
   groupByKeyStage(CONTAINER &container): container(container){}
   void* svc(void* task) {
      container[*((std::string*)task)] += 1;
      return GO_ON;
   }
};
```

```
int main(int argc, char *argv[]) {
    std::unordered_map<std::string, int> result;
    std::string inputFile = DEFAULT_INPUT_FILE;

    if (argc >= 2) { inputFile = argv[1]; }

    linesFromFileStage linesFromFile(inputFile);
    splitInWordsStage splitInWords;
    flatStage flat;
    groupByKeyStage groupByKey(result);

    ff_pipeline ffp;
    ffp.add_stage(&linesFromFile);
    ffp.add_stage(&splitInWords);
    ffp.add_stage(&flat);
    ffp.add_stage(&groupByKey);

    if (ffp.run_and_wait_end() < 0) error("running pipe");
    return 0;
}
```

Un pipeline de `n stages` distincts est créé en instanciant les `n` différents `stages` ; ensuite leurs références sont transmises dans le bon ordre au pipeline. Dans l'exemple du listing 1.7, les `stages` sont représentés par les opérations du programme de décompte du nombre de mots — les quatre lignes avec les appels `ffp.add_stage(...)` (p. 25). Les quatre opérations réalisent les fonctions suivantes :

— `linesFromFile`, qui renvoie un flux séquentiel de lignes à partir du fichier. Le fichier est identifié par le paramètre `inputFile` fourni en argument.
— `splitInWords`, qui divise chaque ligne en mots qui sont ensuite transmis en aval sous forme d'une collection de mots.
— `flat`, qui décompose la collection en mots individuels, lesquels sont ensuite transmis dans le flux en tant qu'éléments individuels de données.
— `groupByKey`, qui regroupe les mots similaires ensemble et ensuite compte le nombre d'occurrences de chaque mot.

1.5 Autres bibliothèques C++

Nous présentons ici quelques autres bibliothèque C++ de traitement de flux de données.

1.5.1 RaftLib

`RaftLib` (Beard *et al.*, 2017) est une plus récente bibliothèque `C++` conçue pour traiter des flux de données. Écrit à l'aide de templates, `RaftLib` vise à paralléliser de manière transparente une application, tout en minimisant le réusinage du code hérité. Comme `FastFlow`, il fournit aux programmeurs un ensemble de mécanismes de bas niveau pour traiter les flux de données. Les programmes résultants sont capables de fonctionner sur des plates-formes multi-cœurs avec mémoires partagées ainsi que sur des plates-formes avec mémoires distribuées.

`RaftLib` construit le flux de traitement sous forme d'un graphe en connectant des nœuds à des ports. Les nœuds, appelés noyaux de calcul, sont des classes `C++` qui étendent la classe `raft::kernel`. Dans le constructeur de cette classe sont définis les ports. Les

ports sont d'entrée ou de sortie pour chaque noyau de calcul. Chacun de ces noyaux implémente une méthode d'exécution où des opérations de traitement sont effectuées. Le graphe est construit dans la méthode `main` où les noyaux déclarés sont liés en connectant des ports à l'aide d'opérateurs de liaison.

La communication entre les noyaux s'effectue via des files de type `FIFO` (*First-In, First-Out*), et les données envoyées dans le flux sont assurées d'arriver dans l'ordre. Les données entrent dans le nœud via le port d'entrée défini dans chaque classe de noyau de calcul. Une fois les données traitées, elles sont envoyées à la sortie via le port de sortie défini dans la même classe. Les files sont réalisées à l'aide de différents types de tampons tels que mémoire partagée et dynamique (*heap memory*). L'ètat du traitement est conservé en interne par chaque noyau de calcul, mais il n'est pas conservé entre les noyaux. Cela permet une parallélisation beaucoup plus simplifiée.

Dans `RaftLib`, il existe plusieurs options pour échanger les données entre des noyaux de calcul. Le processus comprend deux étapes : lire les données à partir des ports d'entrée et écrire les données sur les ports de sortie. Une fois les données envoyées au port de sortie, elles sont disponibles pour le noyau en aval.

Il existe deux types de méthodes d'envoi et de réception des données : les méthodes sans copie (*zero copy*) et les méthodes avec copie (Beard, 2016). *Zero copy* signifie que les données ne sont pas déplacées du noyau vers le contexte de l'application, ce qui élimine les copies inutiles des données. Les méthodes avec copie peuvent être plus rapides pour les transferts de petite taille, par exemple, les données de type primitif.

Les deux bibliothèques, `RaftLib` et `PpFf`, ont plusieurs caractéristiques en commun. Elles sont écrites en utilisant les *templates* de `C++` avec une parallélisation automatique. Les divers nœuds d'un flux pour `RaftLib` et `PpFf` s'exécutent dans des *threads* différents. Dans ce contexte, `PpFf` permet plus de flexibilité en utilisant aussi le parallélisme introduit par les instances parallèles d'un `farm`.

Même si les deux bibliothèques ont plusieurs caractéristiques en commun, leur conception est différente. Tandis que `RaftLib` a été conçue pour prendre en charge le traitement

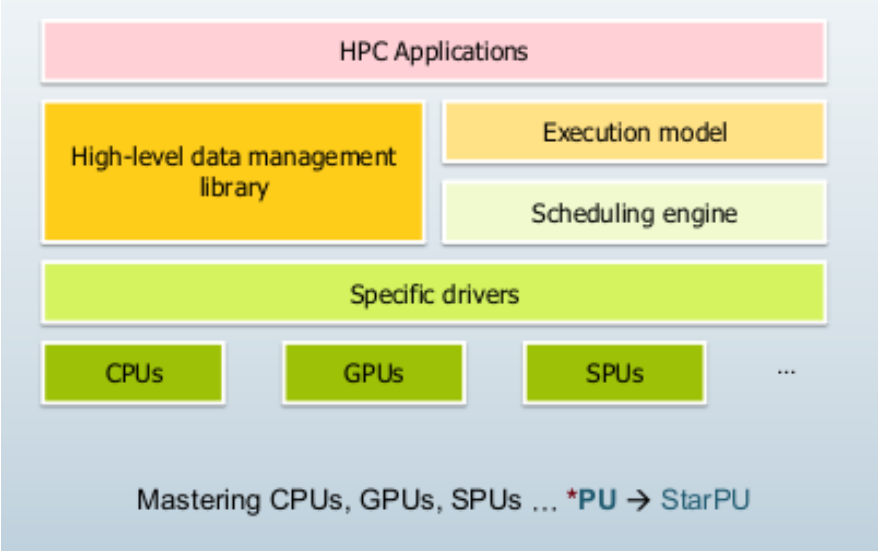

Figure 1.4: La structure interne de `StarPU` — figure tirée de (Aumage *et al.*, 2018b).

des flux par lot et distribué sur des plates-formes hétérogènes, `PpFf` a été conçue pour traiter des données de flux incrémental, mais aussi des collections en lot. Et avec son style fonctionnel, `PpFf` vise la simplicité et l'efficacité en utilisant le chaînage de méthodes et les expressions lambda.

1.5.2 StarPU

Un outre outil pour le traitement de flux de données est `StarPU` (Aumage *et al.*, 2018a). `StarPU` est un système d'exécution visant à permettre aux programmeurs de générer des tâches parallèles sur des unités de traitement comme des *CPU*s et *GPU*s, tout en les déchargeant de la nécessité d'adapter spécialement leurs programmes à la machine cible.

Le modèle de programmation `StarPU` est un modèle basé sur les tâches. Les applications soumettent des tâches de calcul et `StarPU` distribue ces tâches aux unités de traitement disponibles. La figure 1.4 montre la structure interne de `StarPU`. Le composant responsable de la distribution de tâches est le «*Scheduling engine*». Les données qu'une tâche manipule sont automatiquement transférées entre les unités de traitement et la mémoire principale par le composant «*High-level data management library*». Ce composant facilite le travail

des développeurs en les libérant de la charge des transferts de données. De plus, il optimise le traitement en évitant les transferts inutiles : les données sont conservées là où elles ont été utilisées la dernière fois, pour bénéficier de la localité.

Un programme `StarPU` est écrit sous forme de graphe de tâches, chaque tâche travaillant sur un ensemble de données. Le programmeur doit spécifier pour chaque tâche les données d'entrée et de sortie, créant ainsi des dépendances entre les tâches. De cette façon, une application `StarPU` génère un graphe des tâches au moment de l'exécution. Les tâches sont asynchrones, et donc soumettre une tâche à `StarPU` est une opération non bloquante. Comme tout autre système, `StarPU` introduit des coûts pour gérer les tâches. Ces coûts peuvent être importants si la quantité de travail à effectuer n'est pas assez grande.

La partie calcul d'une tâche est appelée *codelet*. Donc, dans `StarPU`, l'exécution d'une tâche consiste à appliquer un *codelet* sur une structure de données. Un *codelet* peut ensuite être implémenté sur des architectures hétérogènes, telles que `CUDA` ou `OpenCL`.

Dans `PpFf`, ce processus de création de la chaîne de traitement du flux est simplifié par l'utilisation du chainage de méthodes fonctionnelles. Chaque méthode effectue un traitement spécifique sur le flux. De plus, dans `PpFf`, l'équivalent d'un *codelet* `StarPU` peut être défini lors de la déclaration du flux de traitement, à l'aide d'expressions lambda. Avec moins de code, les expressions lambda facilitent la compréhension de la partie de traitement du flux.

1.5.3 SkePU

`SkePU` (Ernstsson et Kessler, 2015) est une bibliothèque `C++` conçue pour les systèmes parallèles hétérogènes. Elle propose un ensemble de squelettes algorithmiques pour spécifier les tâches parallèles sur des systèmes *multi-GPU* utilisant `CUDA` ou `OpenCL`.

`SkePU` fournit sept fonctions squelettes à usage général pour traiter les données en parallèle : `Map`, `Reduce`, `MapReduce`, `MapArray`, `MapOverlap`, `Scan` et `Generate`. Ces fonctions sont implémentées pour différents types de processeurs et plates-formes, y

compris des processeurs séquentiels et multicœurs en utilisant `OpenMP`, `OpenCL` et `CUDA` pour l'exécution mono et *multi-GPU*.

Dans `SkePU`, les données sur lesquelles les fonctions opèrent sont conservées dans des conteneurs dits «intelligents». Ces conteneurs sont disponibles sous forme de vecteurs et matrices avec des éléments de type générique. Les données sont transférées uniquement lorsque cela est nécessaire en gardant une trace des parties qui sont allouées et téléchargées sur le `GPU`. Si un calcul est effectué, en changeant le conteneur dans la mémoire `GPU`, il n'est pas immédiatement transféré dans la mémoire hôte. La copie n'est effectuée que si l'élément est utilisé. Ce mécanisme s'appelle *lazy memory copy*.

`SkePU` donne aux développeurs la possibilité de définir leurs propres opérateurs. Cela est réalisé via des *user functions*, lesquelles sont des opérateurs utilisés pour instancier et initialiser une fonction squelette. Il y a deux façons de définir une *user function* : par une fonction *template* ou par une expression lambda. Une fonction *template* est toujours préférable car elle pourrait ensuite être partagée entre des instances.

Malgré le fait que `SkePU` supporte une large gamme de squelettes algorithmiques en cachant les détails du parallélisme, de la communication et de la synchronisation, cela reste une bibliothèque de bas niveau de traitement de flux de données. Par rapport à `PpFf`, dans `SkePU` les fonctions sont déclarées de façon impérative. `PpFf` propose une manière plus simple de définir les traitement via le chaînage de méthodes. De plus, `PpFf` offre de nombreuses méthodes prédéfinies pour les opérations les plus utilisées de traitement de flux.

CHAPITRE II

DESCRIPTION DE L'API DE PPFF

Ce chapitre présente l'API de la bibliothèque `PpFf`, c'est-à-dire, l'interface avec laquelle interagit le développeur. La conception de l'API de `PpFf` permet aux utilisateurs de tirer parti de la simplicité d'utilisation tout en cachant la complexité des mécanismes concurrents utilisés. La figure 2.1 présente une vue d'ensemble des diverses méthodes exportées par `PpFf`, exprimée à l'aide d'un diagramme de classes UML. [1] Le type principal exporté par l'API est le type `Flow`, qui permet de définir un *flux de données* (Sect. 2.2). Ensuite, sur la base du type de fonctionnalité exportée, l'API se divise en quatre catégories : `Source` (Sect. 2.3), `Transformation` (Sect. 2.4), `Agrégation` (Sect. 2.5) et `Execution` (Sect. 2.6). Quant à la catégorie `Agrégation`, elle est elle-même divisée deux sous-catégories, selon le type de résultat produit : valeur simple (par ex., résultat booléen ou entier) ou collection.

L'objectif de notre API était d'obtenir une API semblable à celle des `Streams` de `Java` 8, basée sur le chaînage de méthodes, donc la construction d'une séquence de traitements dans un style qui ressemble à un `pipeline`.

Le chapitre présente également le code source d'un exemple, `WordCount`, pour illustrer l'utilisation de l'API et l'effet des principales méthodes — un exemple semblable à celui présenté dans l'article ayant popularisé l'approche `MapReduce` (Dean et Ghemawat, 2004).

La bibliothèque `PpFf` est mise en œuvre au-dessus de la bibliothèque `FastFlow`, mise en œuvre qui sera décrite au prochain chapitre.

---

1. Tous les diagrammes de classes UML présentés dans ce mémoire ont été produits à l'aide de `PlantUML`, un outil qui permet de spécifier un diagramme UML *de façon textuelle* et qui génère ensuite, de façon automatique, le graphique correspondant, ici en format `png` : `https://plantuml.com/`.

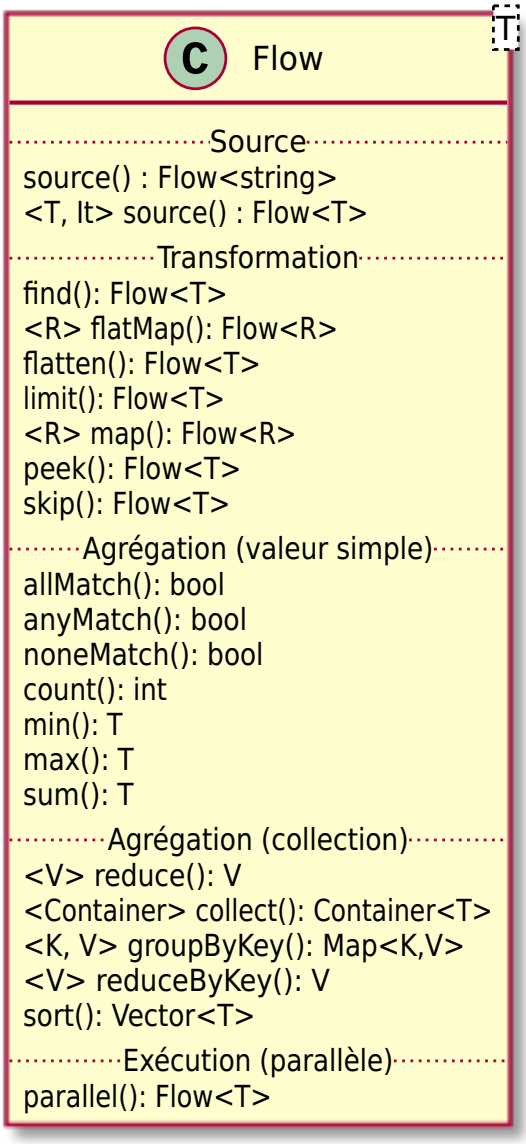

Figure 2.1: Les différentes méthodes exportées par l'API de `PpFf`, regroupées selon leur type de fonctionnalité.

Listing 2.1: Des extraits du code source de l'application `WordCount` qui compte le nombre d'occurrences des mots dans un texte.

```
// Un conteneur pour les mots d'une ligne.
typedef std::vector<std::string> Words;

// Un Reducer pour combiner les elements.
Reducer<std::string, int> reducer(
          0,
          [](int count, std::string _) { return count + 1; },
          std::plus<int>{}
        );

// Le resultat final, un map obtenu par traitement d'un flux.
std::unordered_map<std::string, int> currentResult =
    Flow
    ::source(inputFile)
    .parallel(4)
    .flatMap<std::string, Words, std::string>(splitInWords)
    .map<std::string, std::string>(toLowercaseLetters)
    .reduceByKey<std::string, std::string, int>(reducer);
```

2.1 Exemple : l'application WordCount

L'application `WordCount` présentée dans ce chapitre sera utilisée pour illustrer le fonctionnement de `PpFf`. Des extraits du code source de l'application sont présentés dans le listing 2.1. L'application compte le nombre d'occurrences des mots dans un fichier texte. Cette application est composée en combinant plusieurs opérations, qui forment le pipeline de traitement :

— La première opération d'un pipeline sert à définir la *source* du flux de données. Ici, la source est constituée par les lignes contenues dans un fichier. C'est la méthode statique `source()` qui permet d'extraire et retourner un flux avec les lignes du fichier. Le fichier est spécifié par un nom de fichier (une chaine, `inputFile`) fourni en argument à la méthode `source`.

— L'appel à `parallel(4)` permet de répartir les éléments du flux entre divers *threads* — ici, quatre (4) instances de *farm* — et donc d'exécuter les étapes qui suivent en parallèle.

— L'opération `flatMap` décompose chaque ligne en mots individuels en appliquant la fonction `splitInWords` sur chacune des lignes.

— L'opération `map` transforme chacun des mots en remplaçant les lettres majuscules d'un mot en lettres minuscules en appliquant la fonction `toLowercaseLetters`.

— Finalement, l'opération `reduceByKey` regroupe les mots similaires ensemble et compte le nombre d'occurrences de chaque mot, et ce par l'utilisation du `Reducer` (cf. Sect. 2.5.3).

2.2 Flux de données : type Flow

L'interface de programmation proposée par la bibliothèque `PpFf` consiste en un ensemble de méthodes qui permettent à l'utilisateur de manipuler des flux de données de manière simple et efficace. L'interface s'inspire de celle introduite pour les `Streams` de `Java` 8. L'annexe A présente et décrit brièvement les méthodes exportées par l'API, regroupées selon leur fonctionnalité, puis en ordre alphabétique à l'intérieur d'une catégorie.

Comme on peut le voir dans le tableau de l'annexe A, la déclaration des méthodes utilise la programmation *générique* de C++, c'est-à-dire les *templates*. Cela permet aux utilisateurs d'avoir une interface générique unique, de sorte qu'une méthode peut être réutilisée pour n'importe quel type de données.

Un autre point clé dans cette interface est son expressivité. Même avant sa conception détaillée, nous nous étions donné comme objectif de fournir un système suffisamment intuitif et expressif pour le traitement de flux de données. Le chaînage de méthodes et l'application d'expressions *lambda* sont deux caractéristiques de `PpFf` qui simplifient l'utilisation et facilitent la compréhension du code.

Listing 2.2: La transformation d'une collection d'entiers en une autre collection d'entiers en appliquant une expression lambda sur chacun des éléments.

```
std::vector<int> elems = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

std::vector<int> currentResult =
  Flow::source<int>(elems.begin(), elems.end())
      .map<int, int>( [](int* in){ *in *= 3; return in; } )
      .collect<int, std::vector>();
```

2.3 Catégorie Source : méthodes `source`

La spécification de la source de données est la première méthode utilisée pour créer un flux. Plus précisément, l'API fournit deux méthodes **statiques** pour créer un `Flux` à partir de diverses sources, soit des collections ou des fichiers de données. Des travaux futurs pourraient étendre l'interface pour prendre en charge plus des sources de données.

Les signatures pour les deux méthodes sont données dans l'annexe A. Tandis que la première méthode, `source(path)`, produit les données d'un flux à partir du contenu (les lignes) d'un fichier texte, la deuxième, `source(begin_iterator, end_iterator)`, produit les données d'un flux à partir des éléments d'un conteneur STL.

Les deux méthodes pour spécifier la source d'un flux sont illustrées, respectivement, dans les exemples des listings 2.1 et 2.2. La méthode `source` du listing 2.1 envoie dans le flux chaque ligne du fichier. Le paramètre `inputFile` de la méthode spécifie le chemin où se trouve le fichier. Dans le deuxième exemple, celui du listing 2.2, les données sont consommées à partir de `elems`, un `vector`. La méthode `source` envoie dans le flux les données de type `int` du `vector elems` en utilisant ses itérateurs de début et fin.

2.4 Catégorie Transformation

Cette section décrit plus en détail les principales opérations de la catégorie `Transformation`. Ces opérations permettent d'exprimer des requêtes de traitement de données complexes

Listing 2.3: La génération des noms de tous les employés d'une collection dont le salaire est supérieur à 1000.

```
std::vector<std::string> result =
  Flow::source<Employee>(elems.begin(), elems.end())
      .find<Employee>( [](Employee* e) -> bool
                      { return e->salary > 1000; } )
      .map<Employee, std::string>( [](Employee* e)
                                  { return &e->getName(); } )
      .collect<std::string, std::vector>();
```

telles que le filtrage et le «mappage». L'annexe A présente les signatures détaillées de ces méthodes.

2.4.1 Méthode `map`

La méthode `map` est utilisée pour transformer une collection d'objets en une autre collection d'objets en appliquant une fonction — typiquement une expression lambda — sur chacun des objets. Par exemple, dans le listing 2.2, l'expression lambda passée en paramètre à la méthode `map` multiplie par 3 chaque élément reçu. Le résultat renvoyé par cette expression lambda est un pointeur. `PpFf` s'appuie sur le modèle de programmation de `FastFlow`, tel que décrit dans le prochain chapitre. Ce modèle est basé sur le découplage des données et de la synchronisation, et où seuls des pointeurs vers les données sont transmis plutôt que les données elles-mêmes. Une autre utilisation typique de la méthode `map` consiste à obtenir une information de chacun des objets d'une collection. Par exemple, le listing 2.3 montre un exemple où la méthode `map` permet d'obtenir les noms des divers employés d'une collection.

2.4.2 Méthodes `flatten` et `flatMap`

Les méthodes `flatten` et `flatMap` ont pour rôle d'aplanir un flux multiniveau. Alors que la méthode `flatten` crée un flux unique à partir du contenu des divers conteneurs,

la méthode `flatMap` crée un flux unique à partir du contenu des divers conteneurs intermédiaires résultant de l'application d'une fonction fournie en argument et appliquée à chaque élément du flux. Notons que cette dernière méthode (`flatMap`) correspond en fait à une opération `map` suivie immédiatement d'une opération `flatten`. Les signatures détaillées pour ces deux méthodes sont données dans l'annexe A.

Dans le cas de la méthode `flatMap`, le type `Container` est le type intermédiaire résultant de l'application de la fonction `taskFunc` fournie en argument sur le flux. Par exemple, dans le listing 2.1, `Container` est de type `Words` — un vecteur de chaines de caractères. Dans cet exemple, les lignes d'un fichier divisées en mots sont accumulées dans un conteneur et ensuite le contenu du conteneur est transmis sur le flux, un élément à la fois.

2.4.3 Méthode `find`

Une opération courante consiste à sélectionner les éléments d'un ensemble de données qui satisfont une certaine propriété. L'API de `PpFf` fournit une telle fonctionnalité via la méthode `find`. Plus précisément, la méthode `find` sélectionne les éléments d'un flux selon un prédicat de sorte que seuls les éléments qui satisfont le prédicat sont envoyés à l'étape suivante. À noter qu'il est obligatoire que le prédicat retourne une expression booléenne.

Un exemple qui illustre l'utilisation de la méthode `find` est présenté dans le listing 2.3 : l'appel à la méthode `find` sélectionne uniquement les employés du flux dont le salaire est supérieur à 1 000 $.

2.5 Catégorie Agrégation

La dernière étape dans le traitement d'un flux est la collecte des éléments pour produire le résultat. Les méthodes fournies par l'API de la bibliothèque `PpFf` offrent quatre fonctionnalités principales :

— Collecter les éléments du flux dans un conteneur ;
— Réduire les éléments de flux en une seule valeur ;
— Regrouper les éléments selon une clé ;
— Regrouper les éléments selon une clé et réduire en une valeur associée.

Listing 2.4: Un pipeline pour identifier l'employé le plus agé.

```
Employee currentResult =
  Flow::source<Employee>(employees.begin(), employees.end())
      .parallel(4)
      .max<Employee>( [](Employee* older, Employee* e)
                      { if (e->age > older->age) *older = *e; } );
```

2.5.1 Collecte des éléments d'un flux dans un conteneur

Afin de collecter les éléments d'un flux dans un conteneur, l'API de `PpFf` fournit la méthode `collect`. Cette méthode retourne un conteneur STL. Le type pour les éléments du conteneur et le type du conteneur sont donnés par les types indiqués en paramètres *template* de la méthode `collect`.

L'exemple fourni dans le listing 2.3 montre l'utilisation de la méthode `collect`. La méthode collecte les noms des employés d'une collection d'`Employee`s dans un `vector` de `string`s.

2.5.2 Réduction des éléments d'un flux en une seule valeur

Une réduction consiste à combiner les éléments d'un flux en un seul résultat *qui n'est pas un flux*. La méthode `max` est l'une des méthodes offertes par l'API de `PpFf` qui illustre ce type de fonctionnalité. Cette méthode prend un comparateur en argument pour comparer les éléments du flux. Le listing 2.4 montre un exemple où les éléments de type `Employee` d'un flux sont comparés afin de trouver l'employé le plus âgé.

De façon similaire à la méthode `max`, l'API de `PpFf` fournit aussi la méthode `min`, utilisée pour trouver l'élément minimum d'un flux. Ceci est fait en utilisant le comparateur reçu comme argument pour comparer les éléments du flux.

L'API fournit d'autres méthodes qui réduisent les éléments d'un flux à une seule valeur. Parmi les plus simples on trouve `count` et `sum`. En utilisant `count`, on obtient le nombre

d'éléments d'un flux, alors qu'en utilisant `sum`, on additionne les éléments d'un flux. Une autre méthode plus générale fournie par l'API de `PpFf` est `reduce`, qui *combine* les éléments d'un flux en utilisant un `Reducer`, notion présentée dans la section suivante.

2.5.3 Classe `Reducer`

La méthode `reduce` est l'une des méthodes les plus complexes de l'API. Afin de réduire les éléments d'une collection à une seule valeur, la méthode `reduce` doit couvrir quatre cas distincts :

— Réduire les éléments d'une collection sans spécifier de valeur initiale ;
— Réduire les éléments d'une collection à partir d'une valeur initiale fournie par l'utilisateur ;
— Réduire les éléments d'une collection en parallèle sans spécifier de valeur initiale ;
— Réduire les éléments d'une collection en parallèle à partir d'une valeur initiale fournie par l'utilisateur.

Pour implémenter tous ces cas nous aurions dû surcharger la méthode `reduce` plusieurs fois. Dans le but de garder une API simple à utiliser, nous avons introduit la classe `Reducer`. Cette classe est d'autant plus utile que sa fonctionnalité est réutilisée dans d'autres méthodes de l'API — voir la section 2.5.5

Un objet de la classe `Reducer` est utilisé pour spécifier comment réduire les éléments d'un flux à une valeur unique. Le listing 2.5 présente la spécification de la classe `Reducer` avec les signatures des quatre constructeurs associés.

Un `Reducer` peut être construit avec une valeur initiale, une fonction `accumulator` et une fonction `combiner`. La valeur initiale et la fonction `combiner` sont optionnelles. [2] Les constructeurs de `Reducer` permettent de fournir la fonctionnalité de la méthode `reduce` en couvrant les quatre cas distincts mentionnés au début de la section.

---

2. Notons que si la valeur initiale n'est pas spécifiée, alors c'est le premier élément du flux qui est utilisée pour initialiser l'accumulateur, et donc le flux doit contenir *au moins un élément* pour qu'un résultat puisse être produit.

Listing 2.5: La signature de la classe `Reducer` avec ses quatre constructeurs.

```
template<typename In, typename Out>
class Reducer {
    Reducer(std::function<Out(Out, In)> const& accumulator);

    Reducer(Out initialValue,
            std::function<Out(Out, In)> const& accumulator);

    Reducer(std::function<Out(Out, In)> const& accumulator,
            std::function<Out(Out, Out)> const& combiner);

    Reducer(Out initialValue,
            std::function<Out(Out, In)> const& accumulator,
            std::function<Out(Out, Out)> const& combiner);
}
```

Listing 2.6: Un autre pipeline pour identifier l'employé le plus agé, mais avec un `Reducer` et sans parallélisme.

```
Reducer<Employee, Employee>
       reduceAgeMax( [](Employee e1, Employee e2)
                     { return e1.age > e2.age ? e1 : e2; } );

Employee currentResult =
    Flow::source<Employee>(employees.begin(), employees.end())
        .reduce<Employee, Employee>(reduceAgeMax);
```

Listing 2.7: Un exemple d'utilisation d'un `Reducer`, exécuté en parallèle et avec une valeur initiale spécifiée explicitement.

```
int n = 100;
std::vector<int> elems(n);
for (unsigned int i = 0; i < elems.size(); i++) {
    elems[i] = i;
};

// Avec valeur initiale explicite... pour illustrer l'effet!
Reducer<int, int> reducer(3,
                          std::plus<int>{},
                          std::plus<int>{});

int currentResult =
    Flow::source<int>(elems.begin(), elems.end())
        .parallel(2)
        .reduce<int, int>(reducer);

std::cout << "Result: " << currentResult;   // == 4953
```

La fonction `accumulator` est utilisée pour effectuer l'opération de réduction sur les éléments du flux. Cette fonction prend deux paramàtres : le premier est le résultat partiel de l'opération de réduction — la valeur de l'accumulateur — et le deuxième est l'èlément suivant du flux. Par exemple, le listing 2.6 montre l'exemple du listing 2.4, mais réécrit avec la méthode `reduce` et un `Reducer`. Le constructeur de la classe `Reduce` prend comme argument la fonction `accumulator` tout en ignorant la valeur initiale est la fonction `combiner`. La fonction `accumulator` effectue l'opération de réduction en trouvant l'employé le plus âgé.

Le troisième paramètre du constructeur d'un `Reducer`, la fonction `combiner`, est utilisé lorsque le flux est traité en parallèle, en utilisant les instances parallèles d'un *farm*. Dans un tel cas, le flux est divisé en sous-flux — i.e., chaque instance d'un *farm* traite un sous-ensembles des éléments — qui sont réduits en parallèle avec la fonction `accumulator`.

Listing 2.8: Un exemple d'exécution en parallèle de la méthode `reduce` sans utiliser la fonction `combiner`.

```
int n = 100;
std::vector<int> elems(n);
for (unsigned int i = 0; i < elems.size(); i++) {
    elems[i] = i;
};

int currentResult =
    Flow::source<int>(elems.begin(), elems.end())
        .parallel(2)
        .reduce<int, int>(3, std::plus<int>{});

std::cout << "Result: " << currentResult;    // == 4953
```

Les résultats partiels produits par les diverses instances d'un *farm* sont ensuite combinés, dans le *thread* principal, avec la fonction `combiner`. Le listing 2.7 montre un exemple d'utilisation d'un `Reducer` mais exécuté en parallèle. Dans cet exemple, à des fins d'illustration, une valeur initiale est spécifiée, égale à `3`, alors que les fonctions `combiner` et `accumulator` sont toutes deux indiquées explicitement comme étant la fonction d'addition `std::plus<int>{}`.

Il existe des situations où la fonction `accumulator` est identique à la fonction `combiner`, par exemple, celui illustré dans le listing 2.7. L'API de `PpFf` fournit une surcharge de la méthode `reduce` pour pouvoir omettre la fonction `combiner` même si l'exécution est en parallèle. Le listing 2.8 montre l'exemple du listing 2.7, mais réécrit en omettant la fonction `combiner`.

La description (semi-)formelle d'un `Reducer` — décrite par mon directeur de recherche — est présentée dans l'annexe B.1.

2.5.4 Regroupement des éléments selon une clé

Souvent, une opération sur une collection de données consiste à regrouper ses éléments dans un ensemble en fonction d'une ou plusieurs propriétés. Notre API fournit une fonctionnalité similaire via la méthode `groupByKey`.

Le listing 2.9 montre un exemple où les employés sont regroupés selon leur âge. Plus précisément, ce listing présente un des tests unitaires pour `groupByKey`, qui montre que les employés d'un conteneur sont regroupés dans un `map` où la clé est l'âge des employés et la valeur est un `vector` contenant les employés de même âge.

La méthode `groupByKey` comporte en fait deux paramètres, mais le deuxième paramètre est optionnel. Ce paramètre est une fonction qui s'applique sur les éléments sélectionnés pour produire les éléments du `map`. Par exemple, le listing 2.10 présente presque le même exemple que dans le listing 2.9, où les employés sont regroupés selon leur âge, mais dans ce deuxième cas, la valeur résultante ajoutée au conteneur est le nom de l'employé — et non l'employé lui-même. On obtient donc un `map` où la clé est un âge et la valeur est un `vector` *des noms d'employés* ayant cet âge — et non un `vector` des employés.

La description (semi-)formelle de `groupByKey` — décrite par mon directeur de recherche — est présentée dans l'annexe B.2.

2.5.5 Regroupement des éléments selon une clé et réduction d'une valeur associée

Une opération de regroupement et réduction peut être verbeuse et source d'erreurs lorsqu'elle est implémentée dans un style impératif. L'API fournit une fonction qui combine ces deux opérations, et ce dans un style fonctionnel : la méthode `reduceByKey`.

Le listing 2.11 montre un exemple où une telle fonctionnalité est utile. Le listing présente un des tests unitaires pour `reduceByKey`, qui dénombre les employés en fonction de leur catégorie d'âge.

Listing 2.9: Un pipeline pour regrouper les employés selon leur âge. (Ce segment de code est un extrait d'un test unitaire. Les détails exacts de l'assertion ont été omis.)

```
typedef std::unordered_map<int, std::vector<Employee>>
        EMPLOYES_PAR_AGE;

// Definition (omise) d'un vecteur d'objets Employee.
std::vector<Employee> employees = ...;
employees[3].age = employees[4].age = 18;
employees[0].age = employees[1].age = employees[2].age = 22;
employees[7].age = employees[8].age = 33;
employees[5].age = employees[6].age = employees[9].age = 55;

EMPLOYES_PAR_AGE result =
   Flow::source<Employee>(employees.begin(), employees.end())
       .groupByKey<Employee, int, Employee>( // Regroupe selon l'age.
           [](Employee* e) { return &e->age; }
        );

EMPLOYES_PAR_AGE expected = {
   {18, {employees[3], employees[4]}},
   {22, {employees[0], employees[1], employees[2]}},
   {33, {employees[7], employees[8]}},
   {55, {employees[5], employees[6], employees[9]}}
};

// Assertions (omises) qui montrent que result dénote
// un map équivalent à celui représenté par expected...
```

Listing 2.10: Un pipeline pour regrouper les noms d'employés selon leur âge. (Ce segment de code est un extrait d'un test unitaire. Les détails exacts de l'assertion ont été omis.)

```
typedef std::unordered_map<int, std::vector<std::string>>
        NOMS_EMPLOYES_PAR_AGE;

// Definition (omise) d'un vecteur d'objets Employee
// ou le nom de employes[0] est "Employee0", etc.
std::vector<Employee> employees = ...;
employees[3].age = employees[4].age = 18;
employees[0].age = employees[1].age = employees[2].age = 22;
employees[7].age = employees[8].age = 33;
employees[5].age = employees[6].age = employees[9].age = 55;

NOMS_EMPLOYES_PAR_AGE result =
   Flow::source<Employee>(employees.begin(), employees.end())
       .groupByKey<Employee, int, std::string>(
          [](Employee* e) { return &e->age; },
          [](Employee* e) { return &e->name; }
       );

NOMS_EMPLOYES_PAR_AGE expected = {
   {18, {"Employee3", "Employee4"}},
   {22, {"Employee0", "Employee1", "Employee2"}},
   {33, {"Employee7", "Employee8"}},
   {55, {"Employee5", "Employee6", "Employee9"}}
};

// Assertions (omises) qui montrent que result dénote
// un map équivalent à celui représenté par expected...
```

Listing 2.11: Un pipeline pour compter le nombre d'employés de chaque âge. (Ce segment de code est un extrait d'un test unitaire. Les détails exacts de l'assertion ont été omis.)

```
typedef std::unordered_map<int, int> NO_EMPLOYES_PAR_AGE;

// Definition (omise) d'un vecteur d'objets Employee.
std::vector<Employee> employees = ...;
employees[3].age = employees[4].age = 18;
employees[0].age = employees[1].age = employees[2].age = 22;
employees[7].age = employees[8].age = 33;
employees[5].age = employees[6].age = employees[9].age = 55;

Reducer<Employee, int> reducer(
      0,
      [](int count, Employee _) { return count + 1; }
    );

NB_EMPLOYES_PAR_AGE result =
   Flow::source<Employee>(employees.begin(), employees.end())
       .reduceByKey<Employee, int, int>(
             reducer, [](Employee* e) { return &e->age; }
     );

NB_EMPLOYES_PAR_AGE expected = {
   {18, 2},
   {22, 3},
   {33, 2},
   {44, 1},
   {55, 2}
};

// Assertions (omises) qui montrent que result dénote
// un map équivalent à celui représenté par expected...
```

La méthode `reduceByKey` conserve les éléments dans un conteneur de type clé–valeur — donc un dictionnaire (`map`). La clé de chaque élément du flux est cherchée dans le conteneur. Si la clé n'est pas trouvée, une nouvelle paire clé–valeur est ajoutée. Si la clé est trouvée, la fonction de réduction est appliquée sur l'élément du flux et sa valeur associée dans le conteneur. L'opération de réduction — un objet `Reducer` (section 2.5.3) — est spécifiée par l'argument de la méthode `reduceByKey`. Dans le cas particulier de l'exemple fourni dans le listing 2.11, l'opération a pour rôle de compter le nombre d'employés ayant le même âge. Le résultat retourné par la méthode `reduceByKey` est un map de type clé–valeur : la clé représente l'âge d'employés et la valeur associée représente le nombre d'employés qui ont l'âge indiqué par la clé.

La méthode `reduceByKey` est une opération avec une mise en œuvre complexe, encore plus lorsqu'elle est utilisée en parallèle. Toutefois, la bibliothèque `PpFf` cache cette complexité, et donc il est tout à fait possible et intéressant d'utiliser `reduceByKey` en parallèle, et dans ce cas le temps d'exécution peut être réduit — sous certaines conditions que nous examinerons au chapitre 4.

Lorsque le flux est traité en parallèle, il est divisè en sous-flux. Chaque instance crée et manipule son propre *container*, son propre *map*, donc sans interférence avec les *threads* qui traitent les autres sous-flux. Puis, lors de la combinaison des résultats des sous-flux, ces divers *containers/maps* sont fusionnés les uns avec les autres dans le *thread* principal.

2.6 Catégorie Execution

Dans `PpFf`, par défaut, chaque nœud ajouté dans le flux du traitement s'exécute sur un *thread* différent. En plus, la bibliothèque `PpFf` permet d'introduire aussi du parallélisme de données, par le biais d'instances de *farm*. Ces instances sont créées en appelant la méthode `parallel` de l'API.

La méthode `parallel` reçoit un argument entier. La valeur de cet argument permet de spécifier le nombre d'instances parallèles d'un *farm* entre lesquelles seront répartis les éléments du flux à traiter. Par exemple, dans le listing 2.1, l'appel à la méthode `parallel(4)` indique de répartir les éléments du flux entre quatre (4) instances parallèles d'un *farm*.

2.7 Expressivité de PpFf par rapport à FastFlow et Java

Dans cette section, nous examinons, de façon *informelle*, l'expressivité de `PpFf` en la comparant à celles de `FastFlow` et `Java` à l'aide de quelques exemples.

2.7.1 PpFf vs. FastFlow

Écrit au-dessus de la bibliothèque `FastFlow`, `PpFf` essaie de simplifier le traitement des collections de données par rapport à `FastFlow`. Dans ce qui suit, à l'aide d'exemples simples, nous présenterons ce qui distingue `PpFf` de `FastFlow`.

Chainage de méthodes et expressions *lambda*

`FastFlow` est une bibliothèque de bas niveau pour traiter des collections et flux de données. Travailler avec des collections n'est pas toujours facile lorsque, comme en `FastFlow`, le traitement doit être décrit en indiquant explicitement, pour chaque élément du flux, soit le ou les nouveaux éléments à émettre sur le flux (`ff_send_out(...)` ou `return task`), soit le fait que le traitement de l'élément ne produit aucun résultat (`return GO_ON`)

Par exemple, supposons que dans une université, on doit identifier les étudiants boursiers ayant une moyenne de `A`, puis les regrouper par département — pour simplifier l'exemple, on suppose que les seules notes littérales permises sont `A`, `B`, etc., donc sans `+/-`
Supposons aussi qu'on ait les types auxiliaires suivants :

```
// Nom d'un department.
typedef std::string Dept;

// Conteneur pour les etudiants d'un meme department.
typedef std::vector<Student> Students;

// Conteneur pour regrouper les etudiants par department.
typedef std::unordered_map<Dept, Students> Students_by_Dept;
```

Listing 2.12: a) Les opérations auxiliaires utilisées par le pipeline `FastFlow` du listing 2.12 b), qui vise à regrouper les étudiants boursiers par département.

```
// Premiere operation dans le flux = source de donnees.
struct source : ff_node {
    Students students;
    source(Students students) : students(students){}
    void* svc(void*) {
        for (auto &elem: students) {
            ff_send_out(&elem);
        }
        return EOS;
    }
};
// Operation de filtrage pour les etudiants boursiers.
struct findStudentsAverage : ff_node_t<Student, Student> {
    Student* svc(Student* task) {
        if (task->average == "A") {
            return task;
        }
        return GO_ON;
    }
};
// Operation de regroupement des etudiants par departement.
struct groupByDepartment : ff_node_t<Student, void> {
    Students_by_Dept& container;
    groupByDepartment(Students_by_Dept& container) :
                                container(container){}
    void* svc(Student* task) {
        auto mapIt = container.find(task->department);
        if (mapIt != container.end()) {
            mapIt->second.push_back(*task);
        } else {
            Students students = {*task};
            container[task->department] = students;
        }
        return GO_ON;
    }
};
```

Listing 2.12: b) Un pipeline `FastFlow` pour regrouper les étudiants boursiers par département. Les opérations auxiliaires utilisées par le pipeline sont définies dans le listing 2.12 a). (Ce segment de code est un extrait d'un test unitaire. Les détails exacts de l'assertion ont été omis.)

```
// Conteneur pour le resultat du traitement.
Students_by_Dept result;
// Creation du pipeline.
ff_Pipe<> ffp( new source(students),
               new findStudentsAverage,
               new groupByDepartment(result) );
```

Les listings 2.12 a) et 2.12 b) montrent un exemple de solution avec un pipeline `FastFlow` pour ce problème. Le listing 2.12 a) contient les définitions des opérations requises pour créer le flux. La création du flux est illustrée dans le listing 2.12 b). On peut observer que pour chaque opération dans le flux, on doit définir une nouvelle fonction qui traite chaque élément. De plus, il est difficile de comprendre d'un coup d'œil ce que fait le code : les opérations sont d'abord définies dans différents nœuds (`struct` de type `ff_node` ou `ff_node_t`) et sont ensuite utilisées dans la création du flux.

Avec `PpFf`, ce problème peut être résolu plus simplement en utilisant le chainage de méthodes et des expressions *lambda*. `PpFf` propose de nombreuses méthodes prédéfinies pour les opérations de traitement de flux les plus utilisées. Le listing 2.13 montre l'exemple du listing 2.12, mais réécrit avec `PpFf`. On peut noter que le code est plus facile à comprendre. Les opérations sont définies à la création du flux. De plus, le nombre de lignes de code utilisé dans la version `PpFf` est considérablement réduit comparativement à `FastFlow`.

Le principal avantage de l'utilisation du chainage de méthodes est donc que le code est plus lisible, et les opérations à exécuter sont passées au flux sous forme de blocs d'instructions, et ce avec des expressions *lambda*. De plus, avec `find`, il n'est pas nécessaire de traiter

Listing 2.13: Un pipeline `PpFf` pour regrouper les étudiants boursiers par département. Le type `auto` a été utilisé pour les paramètres des expressions lambda, pour rendre le code plus succinct. (Ce segment de code est un extrait d'un test unitaire. Les détails exacts de l'assertion ont été omis.)

```
// Resultat du traitement.
Students_by_Dept result =
    Flow
    ::source<Student>(students.begin(), students.end())
    .find<Student>(
        [](auto s) { return s->average == "A"; } )
    .groupByKey<Student, Dept, Student>(
        [](auto s) { return &(s->department); } );
```

l'envoi de signaux (`return GO_ON`) lorsqu'un élément ne satisfait pas une condition, donc ne produit pas d'élément à la sortie du flux.

L'utilisation d'expressions *lambda* permet, entre autres, de facilement changer le traitement effectué par une opération d'un pipeline. Par exemple, prenons l'exemple du listing 2.12 où on sélectionne les étudiants boursiers ayant une moyenne de `A`. Que se passe-t-il si on veut plutôt sélectionner les étudiants qui habitent dans une certaine ville ? Dans `FastFlow`, une solution serait d'implémenter une nouvelle sorte de nœud et d'utiliser ce nœud dans la création d'un flux. Il est un peu ennuyeux d'avoir à écrire beaucoup de code pour des fonctionnalités simples lorsqu'elles ne sont peut-être utilisées qu'une ou deux fois. Idéalement, on souhaite minimiser les efforts de développeurs et offrir des opérations simples à mettre en œuvre.

`PpFf` offre la possibilité d'utiliser des expressions *lambda* directement dans la création du flux. Ici, il suffirait donc de modifier l'appel à `find` comme suit lors de la création du pipeline pour sélectionner les étudiants provenant de Montréal :

```
.find<Student>(
    [](auto s) { s->city_address == "Montreal"; } )
```

Listing 2.14: Un pipeline `FastFlow` pour regrouper les étudiants boursiers par département en utilisant les instances parallèles d'un *farm*. (Ce segment de code est un extrait d'un test unitaire. Les détails exacts de l'assertion ont été omis.)

```
// Creation des instances de travailleurs d'un farm.
std::vector<ff_node*> workers;
for ( uint32_t i = 0; i < PAR_LEVEL; i++ ) {
    workers.push_back( new ff_Pipe<>(new findStudentsAverage) );
}

Students_by_Dept result;
ff_Pipe<> ffp( new source(students),
               new ff_farm(workers),
               new groupByDepartment(result) );
```

Traitement en parallèle

De nos jours, tous les nouveaux ordinateurs sont des ordinateurs multicœurs. Au lieu d'un seul processeur, ils en disposent de plusieurs. Les deux bibliothèques, `FastFlow` et `PpFf`, peuvent améliorer le traitement des grandes collections de données si les ressources d'une telle machine peuvent être exploitées efficacement.

Par défaut, chaque nœud d'un flux dans `FastFlow` et `PpFf` s'exécute dans un *thread* différent. Le traitement d'un flux peut être davantage parallélisé si on utilise aussi les instances parallèles d'un *farm*. Dans un tel cas, le flux est divisé en sous-flux — i.e., chaque instance d'un *farm* traite un sous-ensemble des éléments. Cette approche peut améliorer les performances, mais augmente aussi la complexité. Dans `FastFlow`, l'utilisateur doit créer explicitement le `farm` et les travailleurs associés. Le listing 2.14 montre l'exemple du listing 2.12 b), mais réécrit en uililisant des instances parallèles d'un *farm* — où on utilise `PAR_LEVEL` travailleurs. Dans le cas de `PpFf`, ce mécanisme est plus simple pour l'utilisateur : il suffit d'ajouter un appel à la méthode `parallel` dans le flux, en spécifiant en argument le nombre de sous-flux à utiliser. Le listing 2.15 montre un exemple pour illustrer la simplicité d'utilisation de cette méthode de `PpFf`.

Listing 2.15: Un pipeline `PpFf` pour regrouper les étudiants boursiers par département en utilisant les instances parallèles d'un *farm*. (Ce segment de code est un extrait d'un test unitaire. Les détails exacts de l'assertion ont été omis.)

```
    // Rappel des types
    typedef std::vector<Student> Students;
    typedef std::unordered_map<Dept, Students> Students_by_Dept;

    Students students;
    // ...

    Students_by_Dept result =
        Flow
        ::source<Student>(students.begin(), students.end())
        .parallel(PAR_LEVEL) // Ajout
        .find<Student>(
           [](auto s) { return s->average == "A"; } )
        .groupByKey<Student, Dept, Student>(
           [](auto s) { return &(s->department); } );
```

Listing 2.16: Un pipeline `Java` pour regrouper en parallèle les étudiants boursiers par département

```
List<Student> students;
// ...

Map<String, List<Student>> studentsByDept =
    students
    .stream()
    .parallel()
    .filter( s -> s.average == "A" )
    .collect( Collectors.groupingBy( s -> s.department ) );
```

2.7.2 PpFf vs. Java

Regroupement des étudiants boursiers par département

L'interface fournie par `PpFf` s'inspire de celle introduite pour les `Streams` de `Java` 8. Basé sur le chaînage de méthodes, un programme utilisant la bibliothèque `PpFf` est assez semblable à un programme utilisant des `Streams`. Dans ce qui suit, à l'aide de deux exemples, nous présentons quelques similitudes et différences entre `PpFf` et `Java`.

Le premier exemple est le même que celui présenté précédemment, où on veut regrouper, par département, les étudiants boursiers ayant une moyenne de `A`. Les codes pour les deux versions équivalentes sont présentés dans les listings 2.15 (`PpFf`) et 2.16 (`Java`).

La première méthode utilisée dans les deux programmes permet d'extraire les données de la source, source qui est constituée d'une collection d'objets du type `Student` :

```
// PpFf
Flow::source<Student>(students.begin(), students.end())

// Java
students.stream()
```

Dans `PpFf`, cette méthode est une méthode statique de la classe `Flow`. Par contre, en `Java`, on crée le flux via la méthode d'instance `stream()` de la classe `Collection`.
La deuxième méthode, qui a le même nom pour les deux versions, est la méthode `parallel`. Comme expliqué précédemment, la parallélisation se fait de façon différente dans les deux programmes. `PpFf` exécute chaque opération d'un pipeline sur un *thread* différent, donc avec du parallélisme de flux. Lorsque la méthode `parallel` est enchaînée, le traitement du flux est davantage parallélisé, en utilisant en plus du parallélisme de données. La constante passée en argument à `parallel()` permet de spécifier le nombre d'instances d'un `farm` à utiliser. Par contre, `Java` gère la création de *threads* par l'intermédiaire du *framework* `ork/join` (cf. section 1.3.4). Nous verrons, à la section 4.5, sous quelles conditions l'approche de `PpFf` est préférable... ou pas.
Comme `Java`, `PpFf` permet l'utilisation d'expressions *lambdas*, qui rendent le code plus concis. Dans notre exemple, les expressions *lambda* sont utilisées pour filtrer les étudiants ayant une moyenne de `A` :

```
// PpFf
.find<Student>( [](auto s) { return s->average == "A"; } )

// Java
.filter( s -> s.average == "A" )
```

Les deux méthodes, `find` de `PpFf` et `filter` de `Java`, ont les mêmes fonctionnalités. Par contre, en `PpFf`, à cause de la façon dont nous avons mis en œuvre la bibliothèque avec des *templates* génériques, certaines informations additionnelles de type doivent être fournies — «`find<Student>` » vs. simplement «`filter`».

Tant `PpFf` que `Java` adopte un mécanisme d'évaluation paresseuse, qui retarde l'étape du calcul jusqu'à ce que le résultat soit nécessaire. Dans les deux cas, le traitement sur les éléments du flux ne démarre qu'après l'enchaînement d'une méthode finale de collecte (agrégation) des éléments :

```
// PpFf
.groupByKey<Student, Dept, Student>(
    [](auto s) { return &(s->department); } );

// Java
.collect( Collectors.groupingBy( s -> s.department ) );
```

Alors que `PpFf` possède une méthode explicite pour regrouper les éléments par clé, `Java` définit plutôt, par l'intermédiaire de la classe `Collectors`. diverses méthodes de regroupement — plus d'une quarantaine. L'approche `PpFf` semble plus simple. Par contre, là aussi des informations additionnelles de type doivent être fournies pour les *templates*.

L'application WordCount

Listing 2.17: L'objet `Reducer` utilisé par l'application `WordCount` en `PpFf`.

```
Reducer<std::string, int> reducer(
          0,
          [](int count, std::string _) { return count + 1; },
          std::plus<int>{}
        );
```

Listing 2.18: La classe `ReduceByKey` utilisée par l'application `WordCount` en Java.

```
class ReduceByKey implements Consumer<String> {
    @SuppressWarnings("unchecked")
    private Map<String,Integer> map = (Map) new HashMap();

    public void accept( String s ) {
        map.put(s, map.getOrDefault(s, 0) + 1);
    }

    public void combine( ReduceByKey other ) {
        for ( Map.Entry<String,Integer> entry : other.map.entrySet() ) {
            Integer value = map.getOrDefault(entry.getKey(), 0);
            map.put( entry.getKey(), value + entry.getValue() );
        }
    }

    public Map<String,Integer> toMap() {
        return map;
    }
}
```

Listing 2.19: Des extraits du code source de l'application `WordCount` en `PpFf`.

```
typedef std::vector<std::string> Words;

std::unordered_map<std::string, int> currentResult =
    Flow
    ::source(inputFile)
    .parallel(4)
    .flatMap<std::string, Words, std::string>(splitInWords)
    .map<std::string, std::string>(toLowercaseLetters)
    .reduceByKey<std::string, std::string, int>(reducer);
```

Listing 2.20: Des extraits du code source de l'application `WordCount` en Java.

```
Map<String,Integer> wordsCount =
        Files.lines( Paths.get(inputFile) )
        .parallel()
        .flatMap( WordCount::splitInWords )
        .map( WordCount::toLowerCaseLetters )
        .collect( ReduceByKey::new,
                  ReduceByKey::accept,
                  ReduceByKey::combine )
        .toMap();
```

Le deuxième exemple que nous présentons est pour l'application `WordCount`, application qui a servi aussi dans nos expériences (cf. Sect. 4.2). Ce programme compte le nombre d'occurrences des divers mots dans un fichier texte. Les codes pour les deux versions équivalentes sont présentés dans les listings 2.19 (`PpFf`) et 2.20 (`Java`), lesquels utilisent des éléments présentés dans les listings 2.17 (`PpFf`), et 2.18 (`Java`).

— La première méthode utilisée permet d'extraire les lignes à partir d'un fichier. Le nom du fichier est passé en paramètre aux méthodes statiques `source` dans le cas de `PpFf` et `Files.lines` dans le cas de `Java`.

— Présentée dans l'exemple précèdent, la deuxième méthode, `parallel`, permet dans les deux cas de paralléliser le traitement de données.

— Les deux méthodes suivantes, `flatMap` est `map`, ont les mêmes noms. Tant pour `PpFf` que pour `Java`, les méthodes prennent comme argument une fonction qui s'applique sur chaque élément du flux. La méthode `flatMap` avec son argument décompose chaque ligne en mots individuels alors que la méthode `map` avec son argument transforme les lettres majuscules en minuscules.

— La dernière méthode regroupe les mots similaires ensemble et compte le nombre d'occurrences de chaque mot. Alors que `PpFf` réalise ces opérations à l'aide d'un objet `Reducer` (cf. listing 2.17) passé en argument à la méthode `reduceByKey`, notre version `Java` utilise un objet de classe `ReduceByKey` (cf. listing 2.18), classe définie pour faire en sorte que les deux versions soient aussi semblables que possible. Ici, la classe `ReduceByKey` met en œuvre l'interface `Consumer<String>`, donc définit une méthode `accept` — équivalente à la méthode `accumulator` de `Reducer` — utilisée comme deuxième argument de `collect`. Elle définit aussi une méthode `combine` — équivalente à la méthode `combiner` de `Reducer` — utilisée comme troisième argumentde `collect`, pour combiner les résultats intermédiaires. Finalement, pour obtenir le `Map` encapsulé par l'objet `ReduceByKey`, un *getter* simple est aussi défini.

Comme on le constate, les deux versions sont très semblables. Les principales différences sont qu'en `PpFf`, il faut spécifier certains types additionnels pour les *templates* génériques et spécifier de façon explicite le niveau de parallélisation désiré. Par contre, l'appel à la méthode `reduceByKey` avec un `Reducer` semble plus simple que la spécification d'une nouvelle classe.

CHAPITRE III

MISE EN ŒUVRE DE PPFF

Ce chapitre décrit la façon dont `PpFf` est mis en œuvre. De façon générale, la mise en œuvre utilise la bibliothèque `FastFlow`, et ce en héritant et étendant plusieurs de ses classes.

Ce chapitre est divisé en trois sections. La première section décrit les différents éléments composant la bibliothèque `PpFf`. La deuxième section examine comment la parallélisation du flux est réalisée. La dernière section présente quelques exemples décrivant comment un programme `PpFf` est compilé en un graphe `FastFlow` et exécuté.

3.1 Les éléments de PpFf

La bibliothèque `PpFf` est composée de plusieurs éléments qui permettent de gérer les flux de traitement de données. Une vue d'ensemble de ces éléments est illustrée dans la figure 3.1.

— Le point d'entrée de `PpFf` est la classe `Flow`, avec laquelle interagissent les développeurs pour créer des flux de traitement. Toutes les opérations de traitement d'un flux de `PpFf` sont liées aux méthodes exposées par cette classe.
— Le cœur de la mise en œuvre de `PpFf` est le module `Pipeline`. La création et l'exécution d'un flux sont gérées par celui-ci, qui construit tout d'abord une représentation intermédaire, et qui génère ensuite un graphe de nœuds `FastFlow`.
— Le module `Operators` regroupe tous les opérateurs définis dans `PpFf`. Exposés à l'utilisateur par le biais de l'`API` de `PpFf`, ces opérateurs permettent de traiter les données de diverses façons.

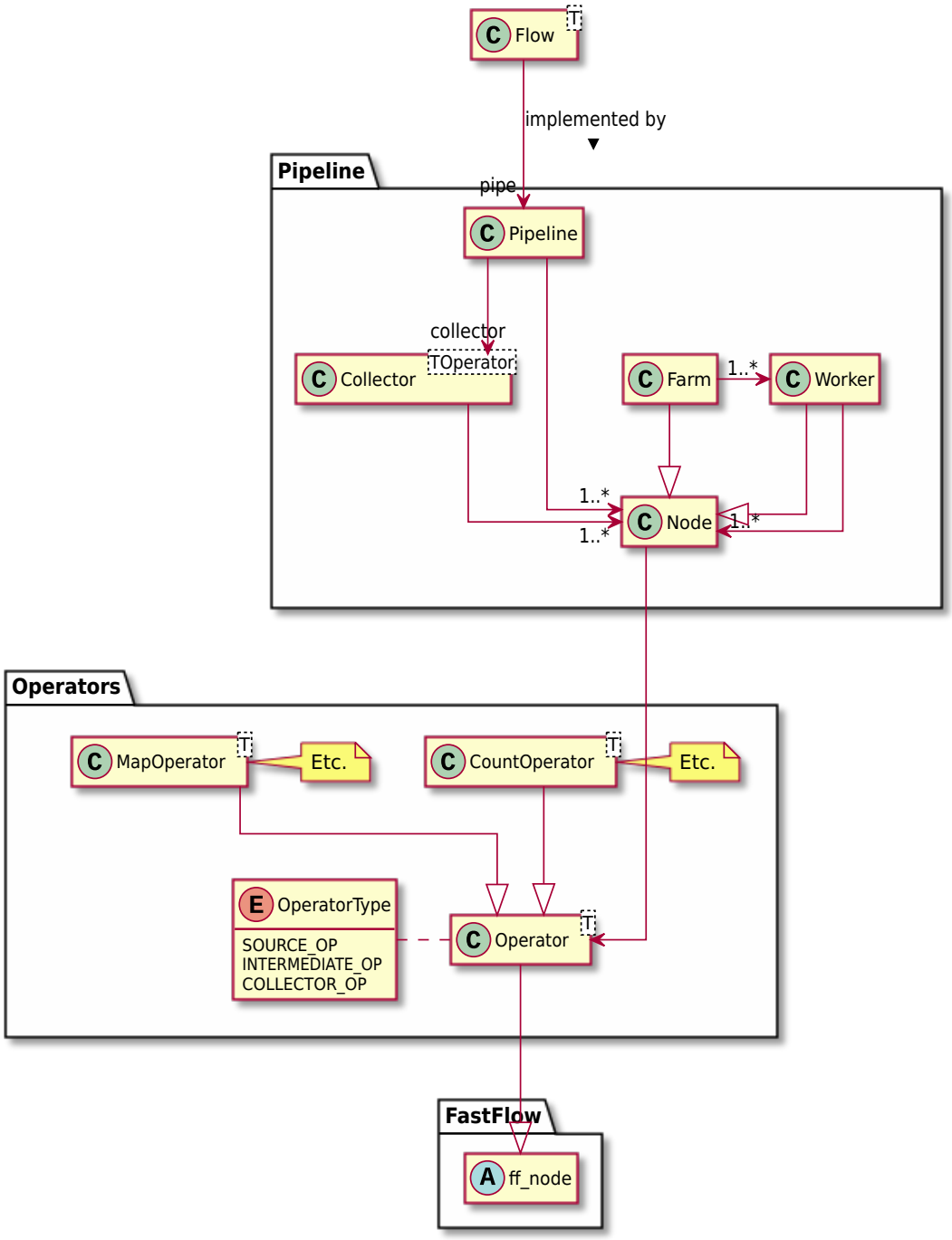

Figure 3.1: Les principaux éléments (classes et paquetages) de `PpFf`.

— Le dernier module, `FastFlow`, est la bibliothèque au–dessus de laquelle `PpFf` est implémenté.

3.1.1 Flow

La classe `Flow` est celle qui définit l'`API` de la bibliothèque `PpFf`, l'interface avec laquelle interagit le développeur. La figure 3.2 — qui reprend la figure du chapitre précédent (Fig. 2.1) — présente les diverses méthodes exportées par `PpFf`. Selon le type du résultat exporté par l'`API`, les méthodes sont divisées en plusieurs groupes : Source, Transformation, Agrégation (valeur simple ou collection) et Exécution (parallèle).

— Le premier groupe, Source, est le groupe de méthodes (statiques) qui permettent de créer un flux de données. Ce sont des méthodes qui fournissent les données initiales pour un flux. Sans un appel à une telle méthode, un flux ne peut pas exister.
— Le deuxième groupe, Transformation, est composé des méthodes qui retournent une référence vers un objet `Flow`. C'est ce mécanisme permet d'enchaîner les méthodes de l'`API`.
— Le troisième groupe, Agrégation, est divisé en deux sous-groupes, selon le type de résultat produit : valeur simple — les méthodes retournant une valeur simple, habituellement un scalaire (par ex., résultat booléen ou entier) — et collection — les méthodes retournant une collection.
— Le quatrième et dernier groupe, Exécution, comporte une seule méthode. Cette méthode modifie le comportement d'exécution du flux. Lorsqu'elle est ajoutée au `pipeline`, toutes les opérations suivant cette méthode seront exécutées en parallèle, en utilisant les instances d'un `farm`.

L'`API` de `PpFf` permet d'appliquer plusieurs opérations les unes à la suite des autres sur une collection de données. Ceci est possible en enchaînant les méthodes (*method chaining*). Les méthodes de l'`API` peuvent être enchainées tant qu'elles retournent une référence à `Flow`. Lorsqu'une méthode retourne une valeur, valeur simple ou collection, le traitement est alors exécuté.

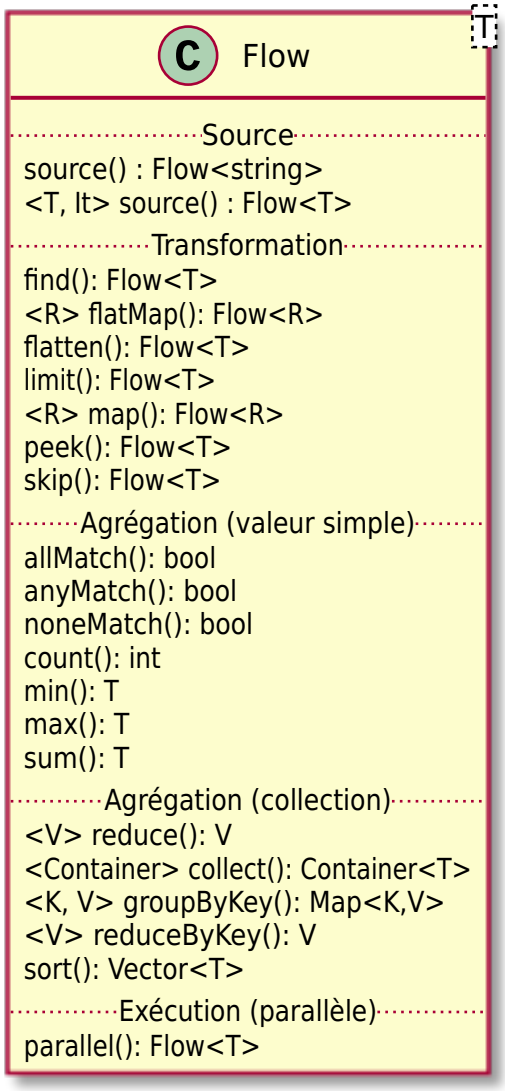

Figure 3.2: Les méthodes exportées par l'API de `PpFf`. Cette figure reprend la figure du chapitre précédent (Fig. 2.1) comme rappel et pour faciliter l'accés à la figure.

Listing 3.1: Des extraits (squelette) de la classe `Flow` avec la variable d'instance `pipe` et le code de la méthode statique `soure`.

```
class Flow {

public:
    static Flow& source(const std::string& path) {
        Flow* flow = new Flow();
        flow->pipe.addNodes<LinesFromFileOperator>(1, path);

        return *flow;
    }

    // ... Autres methodes....
    // ... Voir autres listings pour des exemples...

private:
    Pipeline pipe;
};
```

Listing 3.2: Le code de la méthode `map`, méthode qui fait partie du groupe `Transformation` de la classe `Flow`.

```
template < typename In, typename Out >
Flow& map(std::function<Out*(In*)> const& taskFunc) {
    pipe.addNodes<MapOperator<In, Out>>(pipe.nbWorkers(), taskFunc);

    return *this;
}
```

Listing 3.3: Le code de la méthode `count`, méthode qui fait partie du groupe `Agrégation` de la classe `Flow`.

```
unsigned int count() {
    pipe.addNodes<CountOperator<int>>(pipe.nbWorkers());
    pipe.run();

    return pipe.value<CountOperator<int>, int>();
}
```

Le cœur de la mise en œuvre de `PpFf` est le module `Pipeline`. Ce dernier s'occupe de la création et de l'exécution du flux. Les listings 3.1, 3.2 et 3.3 présentent le code de trois méthodes de l'`API` appartenant à trois groupes différents : `Source`, `Transformation` et `Agrégation`. L'attribut `pipe` utilisée dans chaque méthode est une instance de la classe `Pipeline`, objet qui crée et ajoute les opérateurs dans le pipeline associé au flux en appelant la méthode `addNodes` — voir ci-bas.

Une méthode statique `source`, comme dans le listing 3.1, est toujours la première méthode appelée, pour créer le flux de traitement. Cette méthode a un double rôle : elle crée un objet `Flow` puis fournit les données initiales du flux de données. Le listing 3.2 présente la méthode `map`, qui appartient au groupe `Transformation`, et qui ne fait qu'ajouter une série de nœuds au pipeline de traitement. Ces deux méthodes, `source` et `map`, retournent un objet `Flow`, ce qui permet le chainage de méthodes. La troisième méthode, `count`, qui appartient au groupe `Agrégation`, retourne une valeur entière — `int`, tel que spécifié dans le deuxième paramètre générique de la méthode `pipe.value()` — valeur qui indique le nombre d'éléments du flux. Plus exactement, cette valeur est obtenue (avec `value()`) après avoir exécuté le traitement associé au flux, ce qui se fait avec la méthode `pipe.run()`.

3.1.2 Pipeline

`PpFf` est implémenté au-dessus de la bibliothéque `FastFlow`. Plus précisément, `PpFf` construit tout d'abord une représentation intermédiaire, puis génère ensuite un graphe

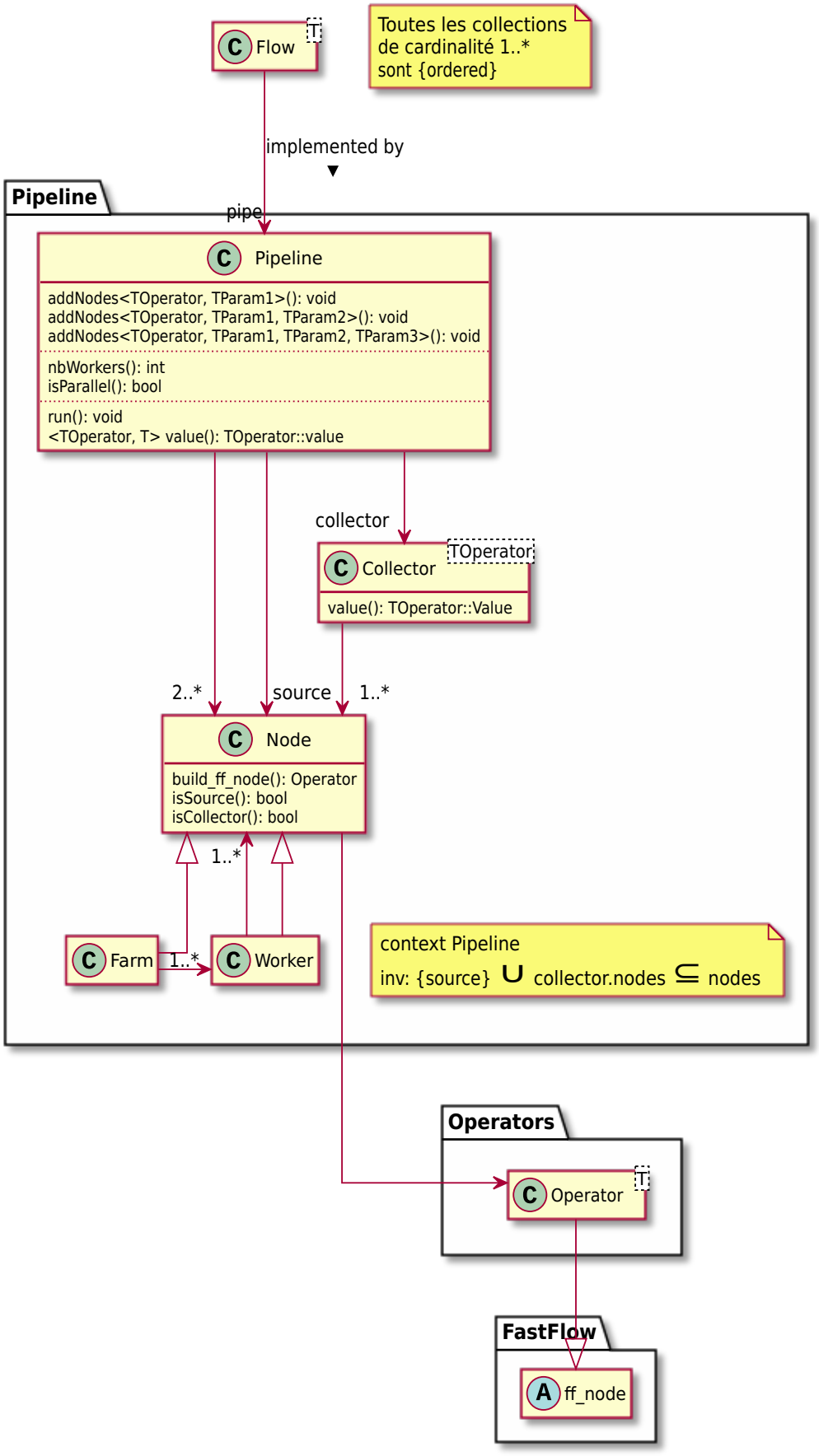

Figure 3.3: Un diagramme UML des éléments pour un `Pipeline` de `PpFf`.

de nœuds `FastFlow`. Toutes les classes nécessaires pour construire une telle structure intermédiaire sont groupées dans le module `Pipeline`, tel qu'illustré dans le diagramme de classes de la figure 3.3.

La classe principale de ce module, la classe avec le même nom que le module, est la classe `Pipeline`. Une instance de cette classe représente une chaîne de traitement. Plus précisément, `Pipeline` est l'implémentation du flux `Flow` décrit dans la sous-section 3.1.1. Un objet `Pipeline` est composé d'une ou plusieurs instances de la classe `Node`. Un `Node` peut être un `Pipeline`, un `Worker` ou un `Farm`. Les classes `Worker` et `Farm` sont utilisées pour créer les instances parallèles d'un `farm` — voir plus bas. Les instances `Node` sont ajoutées au flux en appelant la méthode `addNodes` de la classe `Pipeline`. En fonction du type de parallélisme (de flux, Sect. 3.2.1, ou de données, Sect. 3.2.2), un objet `Pipeline` peut être composé d'instances de `Node` ou de `Farm`. Lorsqu'une instance de `Farm` est ajoutée au pipeline, le flux est divisé en sous-flux — appelés les instances parallèles d'un `farm` — où chaque sous-flux est une instance de la classe `Worker`. Un objet `Worker` peut, à son tour, être un objet `Node` ou un objet `Pipeline`.

La structure ainsi créée sera parcourue lorsque le dernier `Node` ajouté au `pipeline` est un collecteur (identifié à l'aide de la méthode `isCollector` de cette classe). L'exécution du `pipeline` — fait par l'appel de la méthode `run` — implique des appels à la méthode `build_ff_node()` de chaque `Node` de la structure intermédiaire pour construire le graphe `FastFlow` approprié. Ce dernier, composé d'objets de type `ff_pipeline`, `ff_node` et `ff_farm` correspondant aux objets `Pipeline`, `Node` et `Farm` de `PpFf`, est exécuté directement à l'aide de la méthode appropriée d'un pipeline `FastFlow` (`run_and_wait_end()`).

Le résultat du traitement est retourné au `Pipeline` par l'intermédiare de la méthode `value()` de l'objet `Collector`. Si le traitement se fait en parallèle en utilisant les instances d'un `farm`, l'objet `Collector` du pipeline combine les résultats partiels obtenus de chacun des sous-flux.

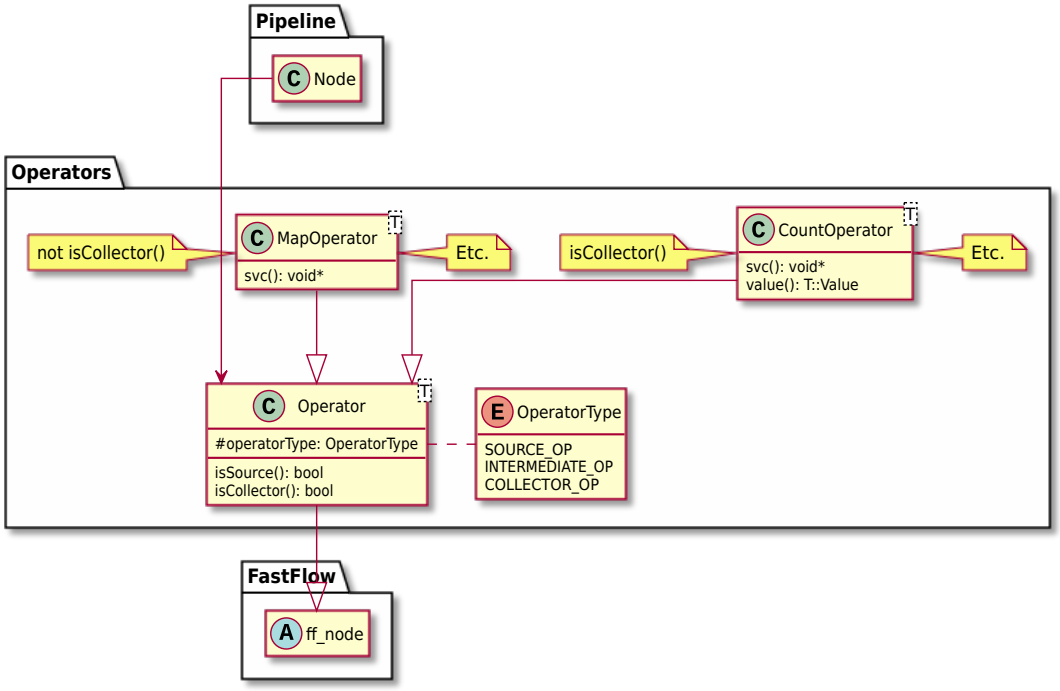

Figure 3.4: Un diagramme UML des `Operators` de `PpFf`.

3.1.3 Operators

Dans la terminologie de `PpFf`, chaque méthode exposée par l'interface de `PpFf` représente une opération et chaque opération est implémentée par un opérateur, un objet provenant d'une sous-classe de la classe `Operator`. Plus précisément, les opérateurs représentent les étapes dans la chaine de traitement de données. Tel qu'illustré dans la figure 3.4, les classes pour les divers opérateurs héritent de la classe de base, `Operator`. Notons que sur cette figure, pour alléger le diagramme, seules deux sous-classes, `MapOperator` et `CountOperator`, ont été indiquées, les autres étant implicites via les «Etc.».

Ces divers opérateurs peuvent être regroupés en trois catégories : *source*, *intermédiaire* et *collecteur*. Un objet `Pipeline` complet est composé d'un opérateur source, de zéro ou plusieurs opérateurs intermédiaires et d'un opérateur collecteur.

L'opérateur source est le premier opérateur ajouté dans le flux ; sans un tel opérateur, aucun traitement ne peut avoir lieu puisqu'il n'y aurait pas de données à traiter. Les deux opérateurs qui font partie de ce groupe sont `SourceOperator` et `LinesFromFileOperator`.

Les opérateurs intermédiaires, tels que `FindOperator`, `FlatMapOperator`, `FlatOperator`, `LimitOperator`, `MapOperator`, `PeekOperator` ou `SkipOperator`, sont des opérateurs qui traitent chaque élément du flux de façon indépendante. Autre point important : les opérateurs intermédiaires n'effectuent aucun traitement tant qu'un opérateur collecteur n'est pas ajouté au `pipeline`.

Le plus nombreux groupe d'opérateurs est celui des collecteurs, qui réunit les opérateurs suivants : `AllMatchOperator`, `AnyMatchOperator`, `NoneMatchOperator`, `GroupByKeyOperator`, `MaxOperator`, `MinOperator`, `ReduceOperator`, `CollectorOperator`, `SumOperator`, `CountOperator`, `ReduceByKeyOperator` et `SortOperator`. Un opérateur de type collecteur est toujours le dernier ajouté au `pipeline`. L'ajout de cet opérateur entraîne la création du graphe `FastFlow` et l'exécution du traitement spécifié par les opérateurs du pipeline. La valeur résultante du traitement est conservée en interne par chaque opérateur collecteur. C'est cette valeur qui va être retournée au `Pipeline` par un appel à `value()`.

La fonctionnalité spécifique à chaque opération du `pipeline` est implémentée par la méthode `svc` de chaque opérateur. Comme on peut le voir dans la figure 3.4, un `Operator` hérite de la classe `ff_node` de `FastFlow`. De cette façon, lorsque le graphe `FastFlow` est exécuté, c'est la méthode `svc` de chaque opérateur qui est appelée. C'est ce qui permet de spécifier la fonctionnalité spécifique à chaque opérateur

3.2 Éxécution parallèle avec parallélisme de flux ou de données

`PpFf` permet aux programmeurs de composer des morceaux de code séquentiel — des fonctions ou expressions lambdas — et d'exécuter le tout en parallèle, et ce à l'aide de deux formes de parallélisme : parallélisme de flux et parallélisme de données. L'objectif de cette section est de décrire la façon dont ces deux formes de parallélisme sont implémentés en `PpFf`.

3.2.1 Parallélisme de flux

Le parallélisme de flux consiste à exécuter plusieurs étapes d'un traitement séquentiel en parallèle en leur faisant traiter des données différentes. Les données se succèdent ainsi les unes aux autres dans les différentes étapes. Dans `PpFf`, une étape est créée lorsqu'une méthode de l'interface de `PpFf` est ajoutée au pipeline par le chaînage de méthodes.

À noter que ce type de parallélisme est appliqué par défaut dans notre bibliothèque. Ce fonctionnement permet de paralléliser des traitements avec des dépendances entre les données sans avoir recours à des synchronisations *explicites* par le programmeur.

Un flux composé de n étapes, où la $i^{\text{è}}$ étape exécute une opération O_i, peut être vu sous la forme d'une composition séquentielle d'opérations sur les éléments d'entrée comme suit, où $O(x)$ représente le traitement global d'un des éléments x du flux de données :

$$O(x) = O_n(\ldots(O_k(\ldots O_1(x))\ldots)\ldots));$$

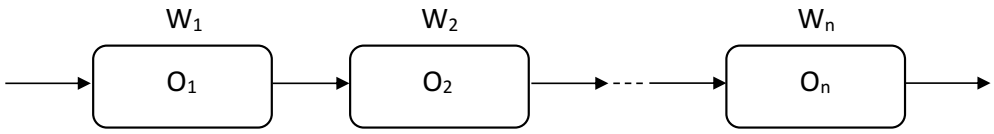

Figure 3.5: Une représentation graphique du parallélisme de flux pour une décomposition en n. Les opérations $O_1, O_2, \ldots, O_n$ sont exécutées de façon parallèle dans différents fils d'exécution représentés par $W_1, W_2, \ldots, W_n$ (W pour $Worker$).

Si on note par W_i le fil d'exécution (*thread*) associé à la $i^{è}$ étape, alors le parallélisme de flux du pipeline peut être représenté par un graphe linéaire de n travailleurs. Chaque travailleur correspond à une opération spécifique O_i. La figure 3.5 montre la représentation graphique du parallélisme de flux dans un tel pipeline. Signalons que cette approche n'accélère pas le calcul d'un élément donné du flux, qui doit parcourir toutes les étapes ; par contre, elle peut améliorer *le débit de sortie*.

3.2.2 Parallélisme de données

Le parallélisme de données implémenté dans `PpFf` consiste à répliquer une opération donnée (ou un pipeline d'opérations) sur un ensemble de travailleurs identiques. Autrement dit, il vise à effectuer un traitement identique sur un ensemble de données indépendantes les unes des autres. Dans notre implémentation de `PpFf`, le parallélisme de données est mis en œuvre avec les *Task Farm* de `FastFlow`.

Décrit à la section 1.4.3, un *Task Farm* est composée de trois entités : un *Emitter*, plusieurs instances de travailleurs (*workers*), et un *Collector*. Par défaut, dans `PpFf`, les éléments sont répartis par un *Emitter* aux divers travailleurs selon une politique *round robin*.[1] Les travailleurs reçoivent les éléments d'entrée et appliquent sur chacun l'opération spécifiée au préalable par l'utilisateur. Les résultats sont ensuite envoyés vers

1. Politique *round robin* : approche d'ordonnancement pour répartir la charge du travail entre des *threads*, avec une file d'attente circulaire. Cf. https://fr.wikipedia.org/wiki/Round-robin_(informatique).

le *Collector* chargé de les collecter — i.e., de les combiner — puis de les transmettre sur le flux de sortie.

Dans `FastFlow` un *Task Farm* est créé en instanciant la classe `ff_farm`. Cette classe correspond à la classe `Farm` du module `Pipeline`. Lorsqu'une instance de `Farm` est ajoutée au pipeline, le flux est divisé en sous-flux — appelés les instances parallèles d'un `farm`. Ceci est possible en utilisant la méthode `parallel` de l'API de `PpFf`. Le nombre de sous-flux ainsi créés dépend de la valeur spécifiée en paramètre pour cette méthode.

Dans `PpFf`, les éléments du flux sont traités dans l'ordre d'arrivée selon le principe *FIFO*. La collection de plusieurs résultats à l'aide de cette approche résulte dans un flux sans ordre sur ses éléments. En effet, le *Collector* transmet les éléments au flux de sortie dans l'ordre d'arrivée. Étant donné que plusieurs flux indépendants sont impliqués, cet ordre ne peut pas être prédéterminé. Si le respect d'un ordre spécifique est requis, l'utilisateur peut ajouter une opération qui produit cet ordre. Par exemple, la méthode `sort` de l'API (cf. l'annexe A), effectue le tri des éléments du flux selon l'ordre spécifié par la fonction `compare` envoyé en paramètre.

L'implémentation parallèle de ce modèle peut être exprimée sous la forme d'un ensemble de n travailleurs $W_1, W_2, \ldots, W_n$ qui appliquent une opération O sur les divers éléments $X = \{x_1, \ldots, x_k\}$ apparaissant dans le flux d'entrée pour produire en sortie un ensemble d'éléments $Y = \{y_1, \ldots, y_k\}$, où $y_i = O(x_i)$. Les divers éléments x_i $(i = 1, \ldots, k)$ sont répartis entre les travailleurs W_j $(j = 1, \ldots, n)$ d'une façon indéterminée, qui dépend notamment de la vitesse de traitement de chaque x_i.

La figure 3.6 montre une représentation graphique du parallélisme de données, où la même opération O est exécutée par les travailleurs $W_1, W_2, \ldots, W_n$.

Il est possible de spécifier en paramètre le nombre de travailleurs à utiliser pour l'exécution parallèle. Si ce paramètre n'est pas spécifié, un seul travailleur est activé. Permettre ainsi d'augmenter le nombre de travailleurs fait en sorte qu'on peut augmenter le débit du traitement des données si approprié, c'est-à-dire, augmenter le nombre de données qui peuvent être traitées en un temps donné si le nombre de cœurs ou processeurs le permet.

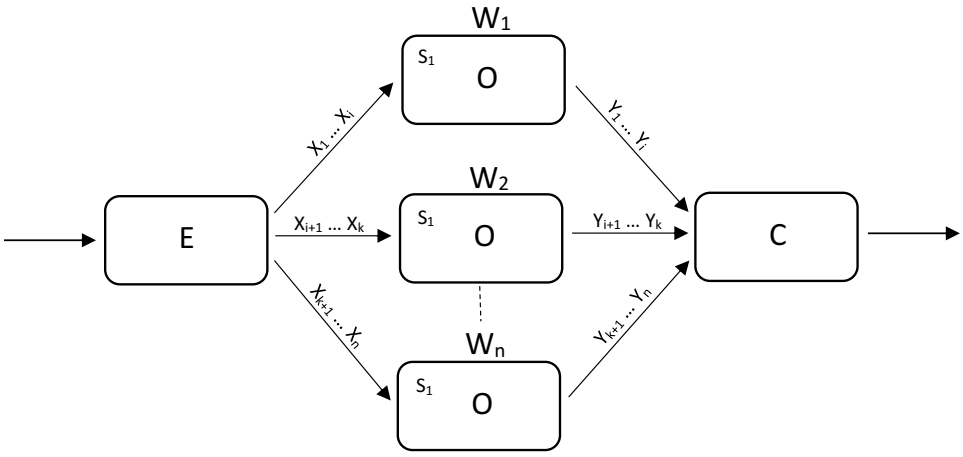

Figure 3.6: Une représentation graphique du parallélisme de données. La même opération O est exécutée par les n travailleurs $W_1, W_2, \ldots, W_n$ (W pour $Worker$). Les éléments de données sont répartis entre les travailleurs par l'émetteur E ($Emitter$) puis les divers résultats sont combinés par le collecteur C ($Collector$).

3.3 Implémentation de PpFf avec FastFlow : quelques exemples

Cette section décrit le modèle d'exécution de `PpFf` via `FastFlow` à l'aide du code source de trois petits exemples. Ces exemples illustrent, à l'aide de diagrammes de classes, la structure de la représentation intermédiaire et la relation entre cette représentation intermédiaire et le graphe `FastFlow` associé.

3.3.1 Exemple 1 : Un flux simple avec uniquement une source et un collecteur

Illustré dans le listing 3.4, le premier exemple analysé ne comporte que deux opérations :

— `source`, la méthode (statique) qui génère un flux séquentiel de nombres entiers à partir d'un conteneur (en `vector`) ;

— `max`, la méthode (d'instance) qui trouve l'élément maximum du flux.

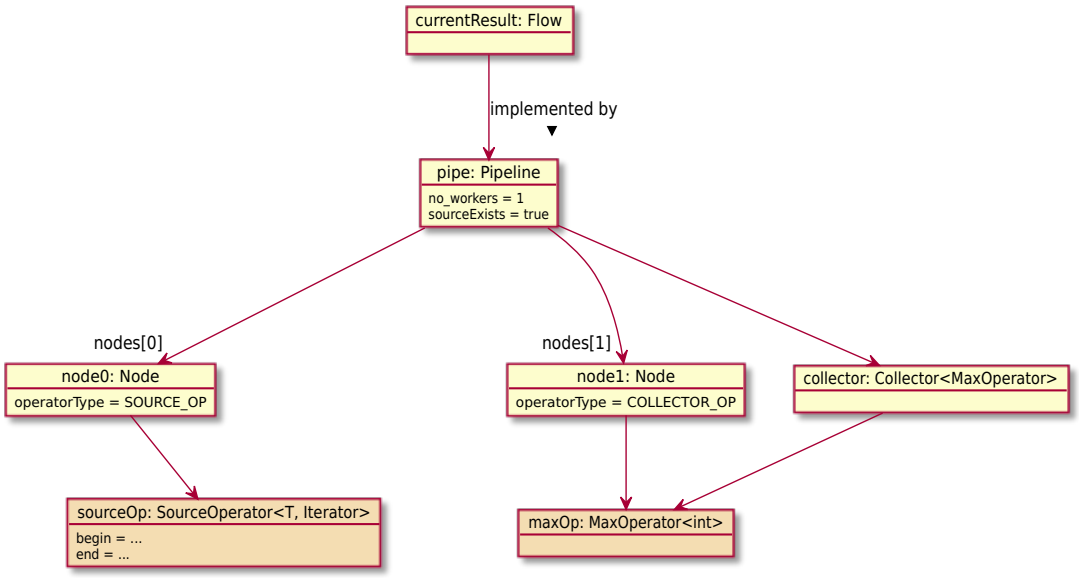

Figure 3.7: La structure intermédiaire `PpFf` pour l'exemple 1 :
`Flow currentResult = Flow::source<int>(...).max<int>();`.

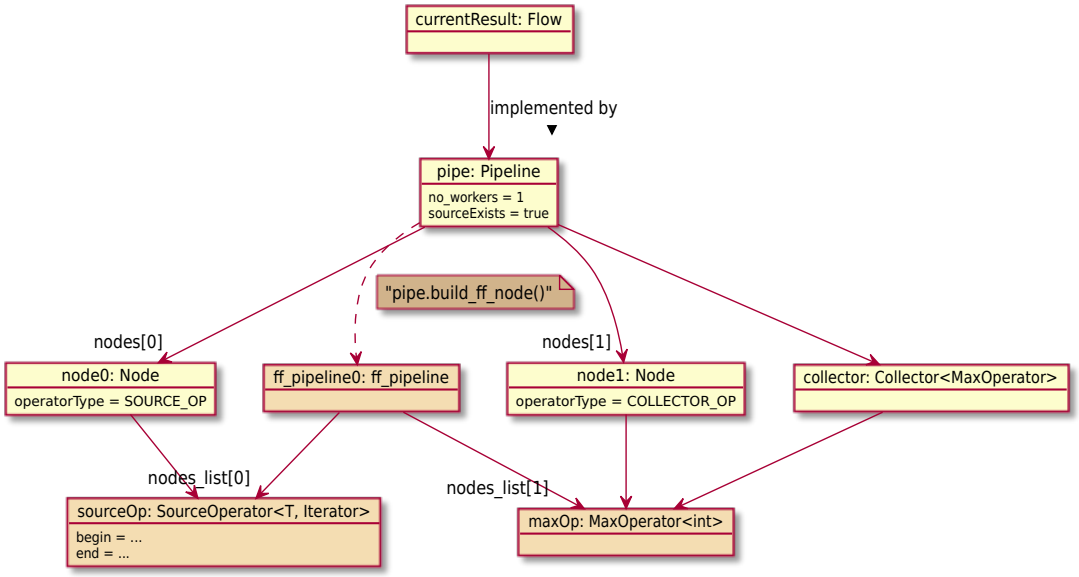

Figure 3.8: La structure intermédiaire `PpFf` et le graphe `FastFlow` associé pour l'exemple 1.

Listing 3.4: Le code source d'un flux pour trouver le maximum d'une collection de données.

```
std::vector<int> elems = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

int currentResult =
    Flow
    ::source<int>(elems.begin(), elems.end())
    .max<int>();
```

`PpFf` construit tout d'abord une représentation intermédiaire, puis génère ensuite un graphe de nœuds `FastFlow`. La représentation intermédiaire, illustrée dans la figure 3.7, est composée d'objets de type `Pipeline` et `Node`. Le premier objet créé est `pipe`, une instance de la classe `Pipeline`. Au fur et à mesure que les méthodes sont enchainées dans le `Flow`, `PpFf` construit la structure en ajoutant au `Pipeline` des instances de type `Node`. On peut observer que pour les deux méthodes enchainées dans `Flow`, deux instances du type `Node` sont ajoutées au `Pipeline` : `node0` et `node1`.

Selon l'opération indiquée, chaque `Node` garde une référence à une instance de la classe `Operator`. Dans notre cas, les deux objets `Operator` sont `sourceOp`, une instance de la classe `SourceOperator`, et `maxOp`, une instance de la classe `MaxOperator`.

Puisque le dernier opérateur du pipeline, `maxOp`, est un opérateur de type collecteur, cela lance la construction du graphe `FastFlow` associé via la méthode `build_ff_node()`. Illustrée dans la figure 3.8, la structure `FastFlow` est créée en parcourant chaque `Node` de la structure intermédiaire. Notons que les objets composant la structure intermédiaire se distinguent de ceux composant le graphe `FastFlow` par une couleur différente : pâle pour les objets `PpFf` et foncé pour les objets qui font partie du graphe `FastFlow` final. Soulignons aussi que cette même distinction au niveau des couleurs se retrouve aussi dans les autres exemples.

Toujours dans la figure 3.8, on peut noter que l'appel à `build_ff_node()` pour l'objet `pipe` génère un objet `ff_pipeline`, un objet `FastFlow`, et que les deux instances de

Listing 3.5: Le code source d'un flux pour trouver le maximum d'une collection de données en utilisant les instances parallèles d'un `farm`.

```
std::vector<int> elems = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

int currentResult =
    Flow
    ::source<int>(elems.begin(), elems.end())
    .parallel(2)
    .max<int>();
```

`Node` retournent les opérateurs associés, `sourceOp` et `maxOp`, qui sont des objets de type `ff_node`, la classe de base d'un graphe `FastFlow`. La structure ainsi créée sera exécutée en appelant la méthode `run_and_wait_end()` de `FastFlow`, et le résultat du traitement, dans ce cas l'élément maximum, sera retourné au `Pipeline` par l'intermédiaire de l'objet `collector`.

3.3.2 Exemple 2 : Un flux simple avec parallélisme de données

Le deuxième exemple illustre le fonctionnement d'un flux avec parallélisme de données, donc utilisant les instances parallèles d'un `farm`. Le code source pour cet exemple est présenté dans le listing 3.5. Par rapport au premier exemple, il y a une seule opération de plus, introduite dans le flux par le chaînage de la méthode `parallel(2)`, qui indique de répartir les éléments du flux entre deux (2) travailleurs (*threads*). Plus précisément, cette opération crée et ajoute une instance du type `Farm` au `Pipeline`. Ceci peut être observé dans la structure intermédiaire créée par `PpFf` et illustrée dans la figure 3.9.

On rappelle que lorsqu'une instance de `Farm` est ajoutée au pipeline, le flux est divisé en sous-flux — appelés les instances parallèles d'un `farm` — où chaque sous-flux est une instance de la classe `Worker`. Dans notre exemple, les deux sous-flux sont représentés dans la figure 3.9 par les instances `worker0` et `worker1`, où chacune de ces instances garde une référence à un objet du type `MaxOperator`.

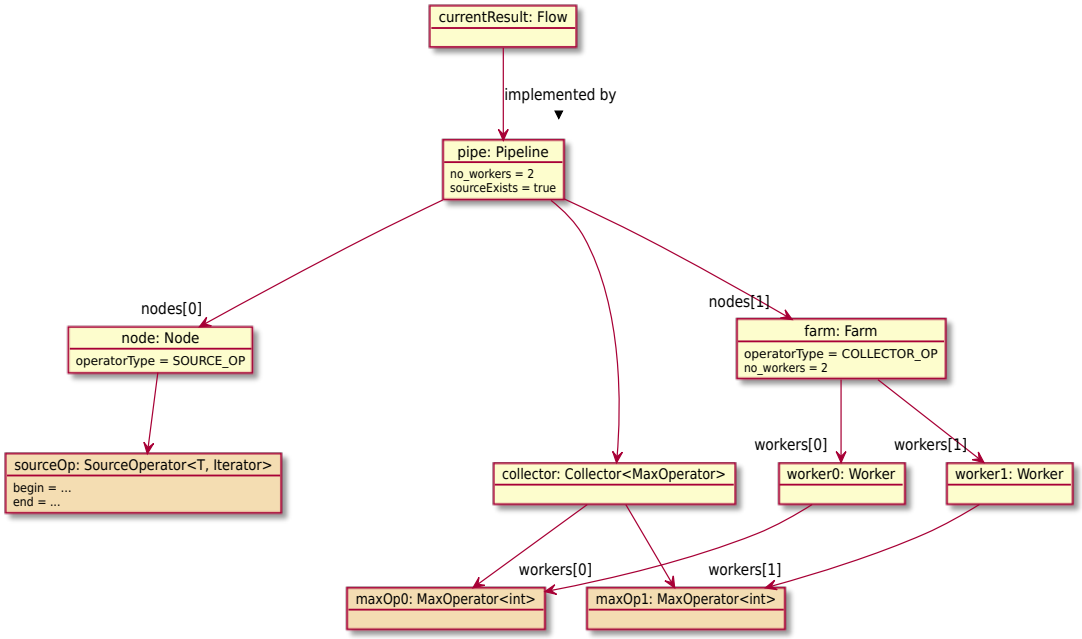

Figure 3.9: La structure intermédiaire `PpFf` pour l'exemple 2 :

`Flow currentResult = Flow::source<int>(...).parallel(2).max<int>();`.

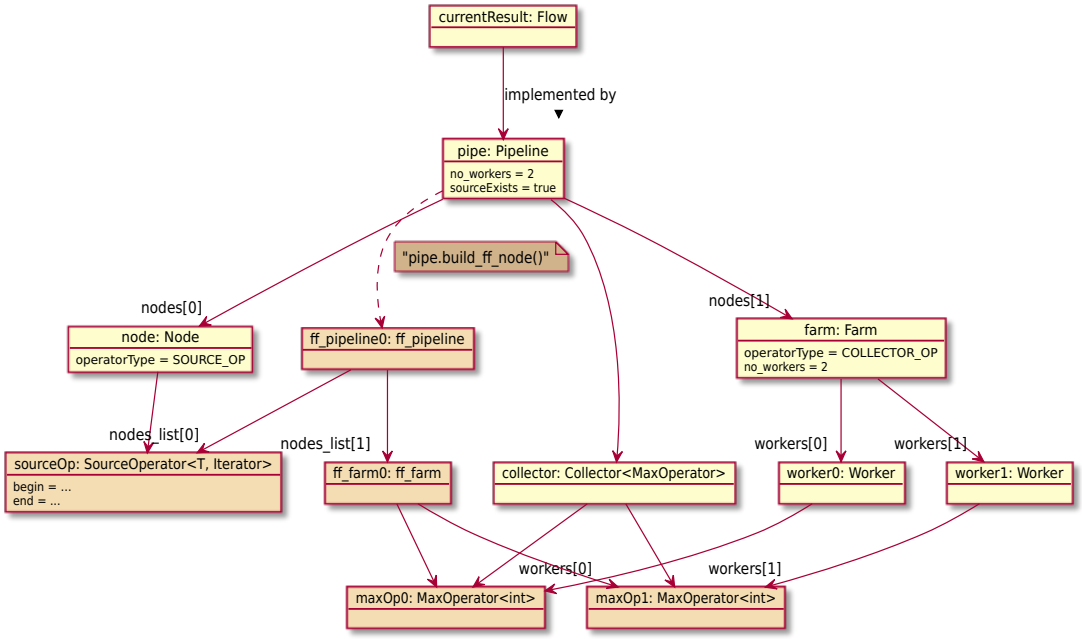

Figure 3.10: La structure intermédiaire `PpFf` et le graphe `FastFlow` associé pour l'exemple 2.

Listing 3.6: Le code source de la classe `Collector` qui combine les résultats partiels de chaque sous-flux via la méthode `value()`.

```
template< typename Operator >
class Collector {
public:
    Collector(std::vector<Node*> &nodes) : myNodes(nodes)
    {}

    Operator::Value value() {
        for (unsigned int i = 1; i < this->myNodes.size(); i++) {
            *((Operator*) myNodes[0]->op()) +=
            *((Operator*) myNodes[i]->op());
        }

        return ((Operator*) myNodes[0]->op())->value();
    }

private:
    std::vector<Node*>& myNodes;
};
```

On a vu dans l'exemple 1 que lorsqu'un opérateur de type collecteur, `maxOp` dans notre cas, est ajouté au pipeline, ceci lance la construction du graphe `FastFlow` associé via la méthode `build_ff_node()`. Illustrée dans la figure 3.10, le graphe ainsi généré est composé des objets `ff_pipeline`, `ff_node` et `ff_farm` correspondant aux instances de `PpFf pipe` (classe `Pipeline`), `sourceOp` (classe `Node`) et `farm` (classe `Farm`). La nouvelle structure créée sera exécutée en appelant la méthode `run_and_wait_end()` de `FastFlow`.

Une différence par rapport à l'exemple 1 est que l'instance de `Farm` divise le flux en deux sous-flux et les maximums produits par les sous-flux sont ensuite combinés par l'objet `collector`. Ceci est illustré dans la figure 3.10, où le `collector` garde une référence aux deux opérateurs, `maxOp1` et `maxOp2`, qui déterminent le maximum pour chaque sous-flux. La combinaison des résultats par le `collector` se fait par l'intermédiaire de l'opérateur `+=()`. Le code source pour la classe `Collector` est présenté dans le listing 3.6. Lorsque la méthode `value()` de cette classe est appelée, les valeurs associées aux opérateurs, une pour chaque sous-flux, sont combinées avec l'opérateur `+=()` associé à ce type d'opérateur, et le résultat final (conservé alors dans le nœud 0, `myNodes[0]`) est retourné.

3.3.3 Exemple 3 : Un flux avec une transformation et avec deux niveaux de parallélisme de données

Le dernier exemple illustre le fonctionnement du parallélisme de données en utilisant deux séries de `farm`. Le code source est présenté dans le listing 3.7. Les deux séries de `farm` sont introduites dans le flux en enchaînant des appels à la méthode `parallel()`, l'un avec l'argument 2, l'autre avec 4, indiquant que les éléments du flux doivent être répartis entre deux (2) et quatre (4) travailleurs (*threads*). Par rapport à l'exemple 2, une autre opération de transformation a aussi été ajoutée au flux, `map`, ici qui multiplie chaque élément du flux par la valeur 3.

La structure intermédiaire créée par `PpFf`, figure 3.9, contient deux instances du type `Farm`, `farm0` et `farm1`. On rappelle que toutes les opérations suivant la méthode `parallel`sont

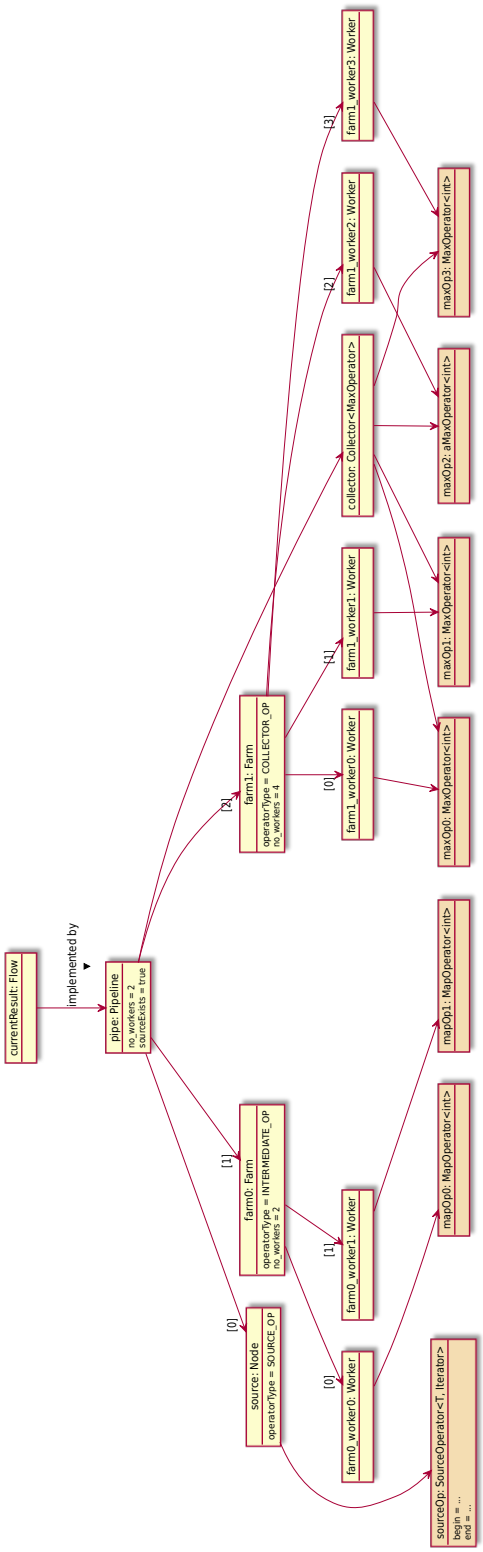

Figure 3.11: La structure intermédiaire `PpFf` pour l'exemple 3 :

`Flow::source<int>(...).parallel(2).map<int,int>(...).parallel(4).max<int>();`.

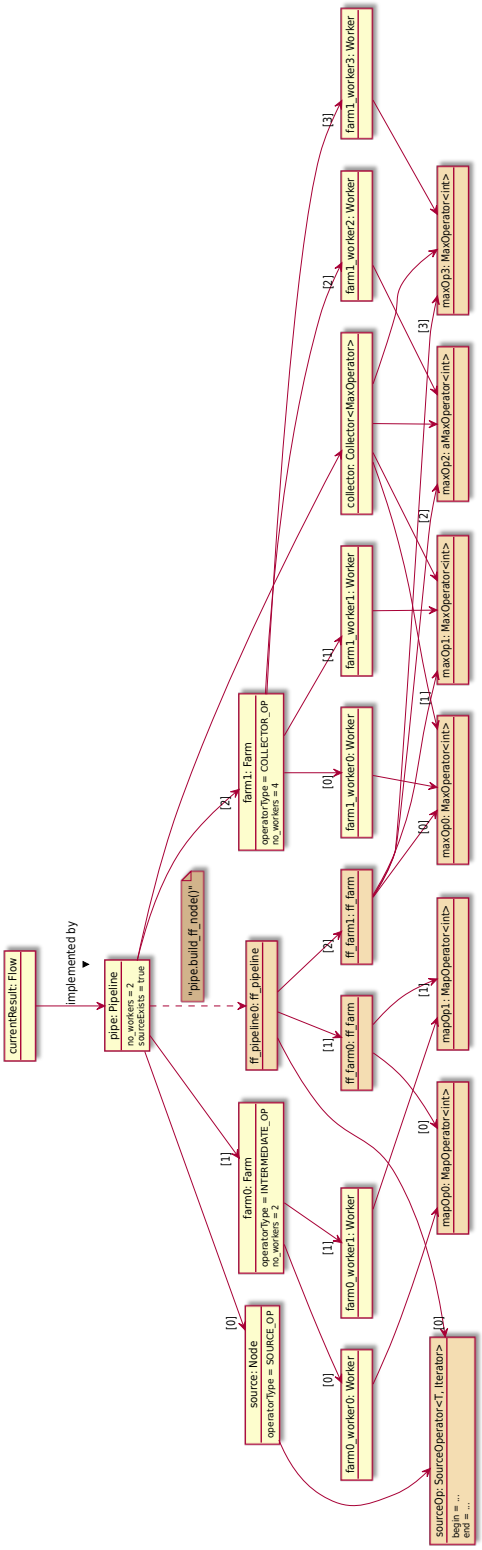

Figure 3.12: La structure intermédiaire `PpFf` et le graphe `FastFlow` associé pour l'exemple 3.

Listing 3.7: Le code source d'un flux pour trouver le maximum d'une collection de données en utilisant un nombre de travailleurs distincts.

```
std::vector<int> elems = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

int currentResult =
    Flow
    ::source<int>(elems.begin(), elems.end())
    .parallel(2)
    .map<int, int>(([](int *in){ *in *= 3; return in; }))
    .parallel(4)
    .max<int>();
```

exécutées en parallèle dans des sous-flux distincts. Dans notre cas, les opérations introduites par les méthodes `map` et `max` seront exécutées en parallèle sur différents sous-flux, par des instances du type `Worker`. Ceci est illustré dans la figure 3.11 où chaque `Worker` de l'objet `farm0` garde une référence à un objet de type `MapOperator` et chaque `Worker` de l'objet `farm1` garde une référence à un objet de type `MaxOperator`.

Comme dans les deux autres exemples, le graphe `FastFlow` est exécuté aprés qu'il est construit via la méthode `build_ff_node()`. Ce graphe est illustré dans la figure 3.12. Le résultat final, l'élément maximum du flux, est déterminé par l'objet `collector` en combinant les résultats partiels de chacun de quatre sous-flux de l'objet `farm1`.

CHAPITRE IV

ÉVALUATION DES PERFORMANCES : COMPARAISONS DE PPFF AVEC JAVA ET FASTFLOW

Ce chapitre présente une évaluation expérimentale de la bibliothèque `PpFf` afin de comparer ses performances avec d'autres approches d'exécution parallèle, plus spécifiquement avec `Java` et `FastFlow`. Dans les sections 4.2 et 4.3, nous présentons deux applications écrites avec `PpFf`. Ces applications ont été choisies non seulement pour montrer certaines fonctionnalités de notre bibliothèque, mais également pour leur pertinence dans des scénarios typiques. La section 4.2 présente une application permettant de calculer le nombre d'occurrences des mots dans un texte alors que la section 4.3 présente une application permettant de calculer des statistiques sur les prix d'indices boursiers — un exemple typique de traitement de flux en ligne (*online data processing*). Mais tout d'abord, nous présentons la façon dont nos expériences ont été effectuées.

4.1 Méthode utilisée pour les expériences

4.1.1 Caractéristiques des machines et compilateurs utilisés

Chaque système informatique a des caractéristiques propres. Quelques-uns des facteurs qui influencent les performances d'un tel système sont le type de processeur, le nombre de processeurs ou de cœurs, et la vitesse des processeurs.

Afin d'avoir des résultats plus représentatifs, nous avons conduit nos expériences sur trois machines différentes. Le tableau 4.1 montre les caractéristiques de ces machines.

	M1	M2	M3
OS	Ubuntu 20.04.1	CentOS 7.8.2003	CentOS 7.6.1810
Architecture	x86_64	x86_64	x86_64
Type de processeur	Intel(R) Xeon(R) Gold 5120	AuthenticAMD	Intel(R) Core(TM) i7-4790
Vitesse du processeur (GHz)	2.20	2.30	3.96
Mono/multi-usager	Multi	Multi	Mono
Nb. coeurs physiques	16	32	4
Nb. coeurs logiques	16	64	8
`java`	`openjdk version "14.0.1" 2020-04-14`	`java version "1.8.0_51"`	`openjdk version "11.0.7" 2020-04-14 LTS`
`g++ (GCC)`	9.3.0	8.3.1	8.3.0

Tableau 4.1: Les caractéristiques des machines utilisées dans les expériences. Le tableau décrit pour chaque machine le système d'exploitation utilisé, le type de la machine (mono- ou multi-usager), le nombre de coeurs (physiques vs. logiques) et les versions des compilateurs utilisés dans les expériences.

Les machines `M1` et `M2` sont des machines multi-usagers, tandis que la machine `M3` est une machine mono-usager sur laquelle les expériences ont éte roulées sans interférence. Soulignons que `M2` est peu utilisée et que nous étions toujours le seul usager lorsque les expériences ont été menées. Quant aux expériences sur `M1`, le serveur `labunix` le plus utilisé, elles ont été lancées durant la nuit — par une commande «`at 3:00AM ...`» — alors que peu d'usagers étaient connectés

4.1.2 Choix des programmes comparés

Le fonctionnement des programmes `Java`, `PpFf` et `FastFlow` n'est pas le même. Alors que `PpFf` et `FastFlow` permettent de varier le nombre de fils d'exécution (*threads*), `Java` ne le permet pas. Afin de montrer les meilleurs temps d'exécution et l'évolutivité de `PpFf`, des expériences préliminaires ont été effectuées, sur chaque machine, pour identifier les meilleures versions, et ce tant pour `PpFf` que pour `Java` et `FastFlow`. Ces expériences préliminaires ont été conduites avec un nombre de répétitions de 10 ou 20 et avec une quantité «moyenne» ou «grande» de données, selon les cas.

Dans le cas de `PpFf` et `FastFlow`, l'objectif était de déterminer le meilleur niveau de parallélisme à utiliser dans les *farm*s, c'est-à-dire, pour `PpFf`, la valeur pour la méthode `parallel()`, pour le parallélisme de données. Par contre, dans le cas de `Java`, l'objectif était de comparer diverses versions : avec ou sans JIT, séquentielle ou parallèle, avec ou sans *warmup* (voir ci-bas). C'est la version parallèle avec *warmup* et *JIT* qui a été utilisée. [1]

L'effet de préchauffage (*warmup*) est généralement dû au chargement des classes et à l'interprétation du *bytecode* au démarrage du programme plutôt qu'à l'exécution directe d'instructions machines. Lorsqu'une nouvelle application démarre, toutes les classes requises sont chargées en mémoire par le modèle de chargement paresseux (*lazy loading*).

---

1. Initialement, des expériences préliminaires ont aussi été effectuées en désactivant complètement le JIT. Toutefois, les temps d'exécutions étaient alors tellement longs que cette variante a rapidement été laissée de côté.

Un tel modèle est couramment utilisé pour reporter l'initialisation d'un objet jusqu'au moment où il est nécessaire. Il y a aussi l'effet de la compilation *JIT* (Cramer *et al.*, 1997) (*Just-In-Time compiler*), un composant de l'environnement d'exécution `Java` qui améliore les performances des applications en compilant le *bytecode* de la machine virtuelle en code machine *au moment de l'exécution*. Le *bytecode* est l'ensemble des instructions de la *JVM* (*Java Virtual Machine*) qui permet aux applications d'être exécutées sur plusieurs plates-formes. La conversion du *bytecode* en langage machine a un impact positif significatif sur la vitesse d'exécution une fois que le code machine s'exécute directement.

Dans toutes nos expériences, les programmes `Java` comparés avec `PpFf` ont donc été préchauffés en lançant une procédure de préchauffage avant de mesurer le temps pour la portion de code pertinente. Cette procédure de préchauffage a consisté à faire un traitement préalable d'une *fraction* des données, traitement utilisant toutes les méthodes utilisées par la suite.

4.1.3 Mesures des temps d'exécution

Il faut noter que le temps mesuré dans toutes les expériences *exclut le lancement du programme*. La mesure du temps se fait de l'intérieur du programme même, une fois celui-ci lancé. Lorsque le traitement est terminé, le temps est émis en sortie du programme sur `stdout`. Donc, le temps n'est pas mesuré avec la commande «`time`». Notamment, dans le cas de `Java`, le temps mesuré exclut donc le temps de lancement de la machine virtuelle. De plus, les quantités de données utilisées dans les expériences ont été choisies de sorte que les temps d'exécution soient d'au moins 200 millisecondes.

4.1.4 Exemples d'expériences préliminaires

Afin de montrer les étapes des diverses expériences qui ont conduit aux résultats finaux, l'annexe C présente un extrait d'un fichier de configuration `WordCount-bm-config.rb`, utilisé pour l'exécution des expériences, alors que les figures 4.1 et 4.2 présentent quelques-uns des résultats préliminaires obtenus.

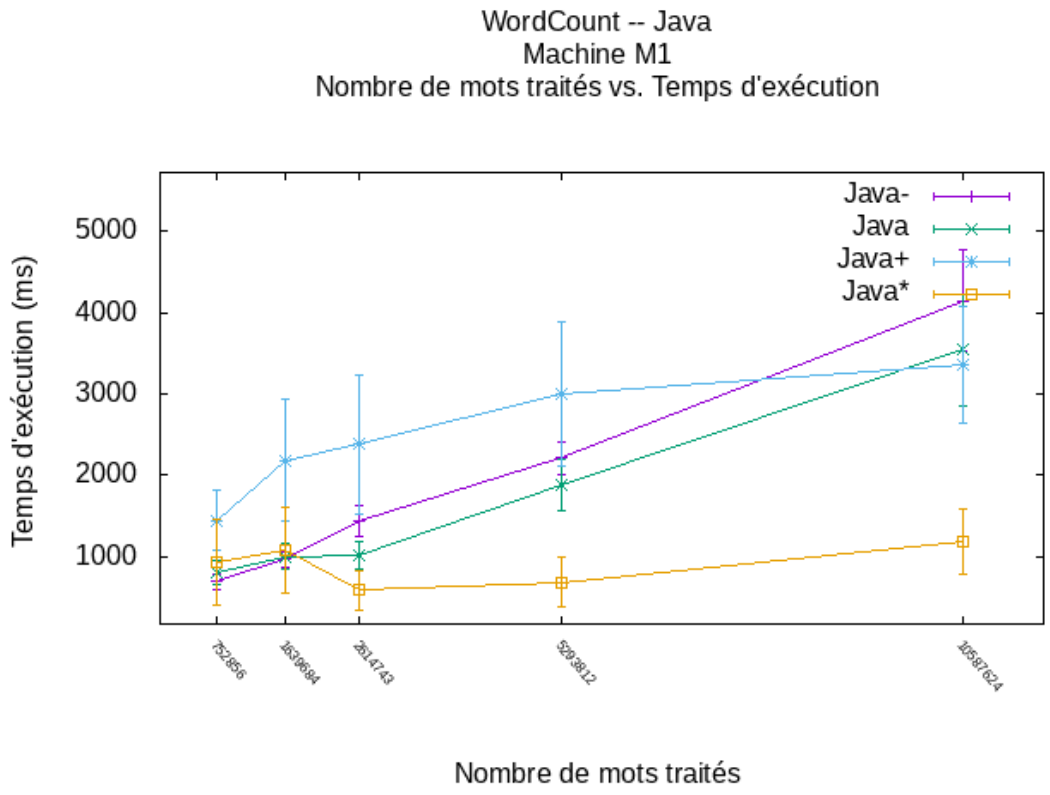

Figure 4.1: Les temps d'exécution pour `WordCount` sur la machine `M1`, pour Temps Java. Les temps sont en millisecondes (ms), obtenus en prenant la moyenne de 20 exécutions.

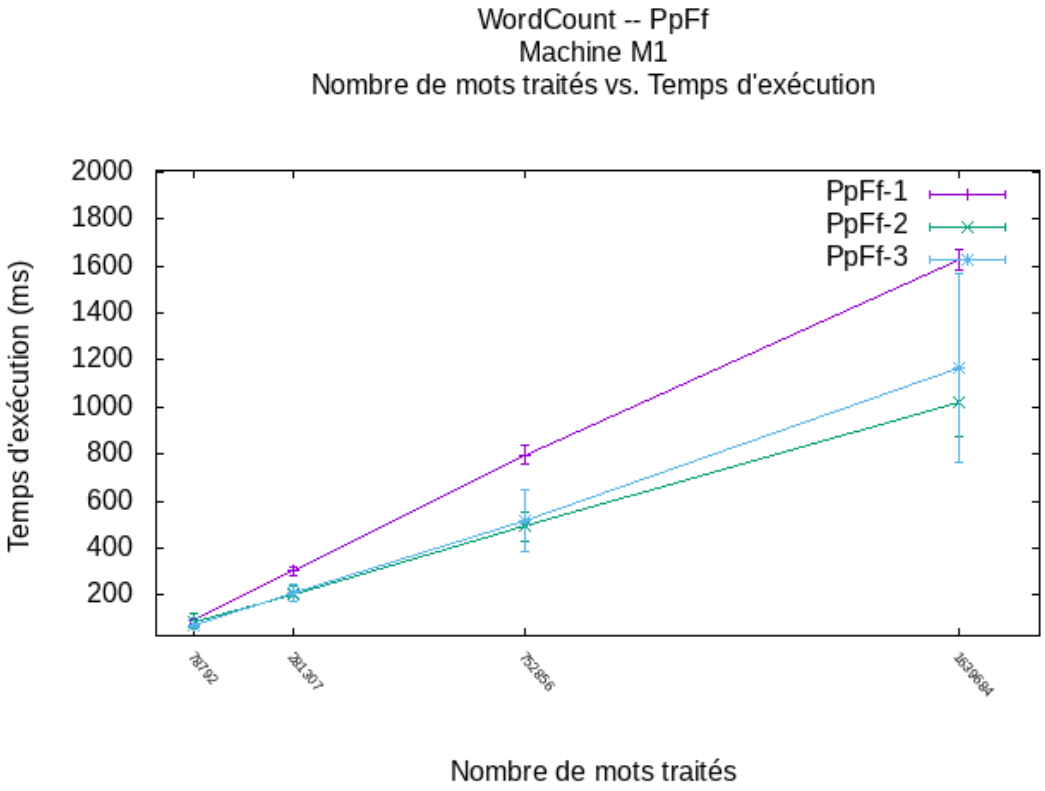

Figure 4.2: Les temps d'exécution pour `WordCount` sur la machine `M1`, pour Temps PpFf. Les temps sont en millisecondes (ms), obtenus en prenant la moyenne de 10 exécutions.

Créé par mon directeur de recherche, un script de configuration permet de décrire les expériences à exécuter pour une application donnée, dans notre exemple, pour `WordCount`. Dans ce fichier, pour chaque expérience, on peut spécifier tous les paramètres requis : la ou les machines sur laquelle est définie l'expérience, la quantité de données à traiter, le nombre de répétitions (pour le calcul de la moyenne), et les divers programmes à exécuter et comparer.

Les quantités de données à traiter sont regroupées en diverses catégories, les plus importantes étant `donnees_preliminaires`, `pas_mal_de_donnees` et `beaucoup_de_donnees`. Les deux premières catégories ont servi pour déterminer les meilleures versions à utiliser pour chacun des programmes (expériences préliminaires). Par exemple, les graphes des figures 4.1 et 4.2 montrent les temps d'exécution pour `WordCount` sur la machine `M1` pour le programme `WordCount.java` (expérience no. 11) et pour le programme `WordCount.cpp`, donc version `PpFf` (expériences no. 2). Les temps sont en millisecondes (ms), obtenus en prenant la moyenne de 20 ou 10 exécutions, selon le cas. Pour la version `Java`, quatre séries de mesures ont été effectuées : séquentielle sans préchauffage (`Java-`), séquentielle avec préchauffage (`Java`), parallèle sans préchauffage (`Java+`), et parallèle avec préchauffage (`Java*`) — pour les résultats, voir l'annexe D. Pour la version `PpFf`, trois séries de mesures ont été effectuées : `PpFf-1` avec une seule instance parallèle d'un *farm*, `PpFf-2` avec deux et `PpFf-3` avec trois.

L'objectif de ces mesures préliminaires était de déterminer la valeur qui semble la meilleure pour les temps d'exécution. On peut observer que le temps d'exécution pour `PpFf-3` est beaucoup plus grand que les deux autres (figure 4.2). (Pour mieux distinguer les performances entre deux ou plusieurs versions, une échelle logarithmique peut aussi être utilisée, lorsque nécessaire.) Pour la machine `M1`, `PpFf-2` est donc la version qui sera comparée aux autres programmes lors des expériences finales. Ces expériences finales comparent donc les meilleures versions entre elles, en utilisant de plus grandes quantités de données et avec un plus grand nombre de répétitions, soit 40.

Le nombre de répétitions indique combien de fois chaque programme est exécuté et son temps mesuré. On calcule ensuite la moyenne et l'écart-type. Dans un graphe comme

celui de la figure 4.1, les moyennes sont représentées par les valeurs qui composent la courbe sur le graphe et les dispersions par des petits barres verticales (par ex., voir les valeurs pour `Java+` de la figure 4.1). Plus précisément, la barre verticale indique un intervalle de deux (2) écart-types, donc un intervalle qui contient approximativement *95 % des temps mesurés.*[2]

Chaque série d'expériences finales inclut les versions séquentielles pour les programmes comparés des deux bibliothèques. Identifiées sur les graphes par `Seq` pour la version séquentielle de `PpFf` et par `Java` pour la version séquentielle avec *warm-up* de `Java`, ces versions utilisent les mêmes fonctions auxiliaires que les versions parallèles, mais s'exécutent de façon strictement séquentielle. Ces programmes séquentiels ont aussi été utilisé pour déterminer les accélérations. Le concept d'accélération détermine à quel point un programme parallèle est plus rapide qu'un programme séquentiel équivalent. On distingue deux types d'accélérations : relative ou absolue. Dans nos mesures, nous avons utilisé l'accélération *absolue*. L'accélération absolue compare le programme exécuté sur une machine multiprocesseurs avec un programme séquentiel, performant, qui résout le même problème.

Afin d'illustrer aussi la dispersion des accélérations, l'accélération moyenne a été calculée, ainsi que deux autres valeurs donnant un intervalle pour la valeur minimale et maximale de l'accélération. Représentées aussi par des petites barres verticales dans les graphes des sections 4.2 et 4.3, ces valeurs ainsi que l'accélération moyenne sont calculées comme suit :

— acc. moyenne = temps moyen séq. / temps moyen par.
— acc. min = temps min séq. / temps max par.
— acc. max = temps max séq. / temps min par.

---

2. En supposant une distribution normale des temps d'exécution.

4.2 Expériences avec l'application WordCount

4.2.1 Description de l'application

Dans la section 2.1, nous avons présenté `WordCount`, une application qui compte le nombre d'occurrences des divers mots dans un fichier texte. L'application prend en entrée un fichier texte et produit un conteneur de type `map<string, int>` où la clé représente un mot dans le fichier et la valeur type `int` associée représente le nombre d'occurrences du mot dans le fichier. Des extraits des programmes utilisés pour les expériences pour `WordCount` en `Java`, `C++` version `Sequentielle`, `PpFf` et `FastFlow` sont présentés dans l'annexe E.

4.2.2 Mesures obtenues et analyse des résultats

Dans cette section, nous évaluons l'application `WordCount` en examinant le temps d'exécution, le dédit et l'accélération sur les trois machines : `M1`, `M2` et `M3`. Tel que décrit dans la section 4.1, des expériences préliminaires ont été effectuées afin de choisir les meilleures versions. Dans le cas de `Java`, la meilleure version choisie est celle avec *warmup* et *JIT*. Elle est indiquée dans chaque graphe avec la notation `Java*`. Dans le cas de `PpFf` et `FastFlow`, les expériences ont été menées en variant les nombres d'instances parallèles d'un *farm*. Par exemple, pour `PpFf`, quatre instances parallèles d'un *farm* ont été utilisés sur la machine `M1`, neuf sur la machine `M2` et deux sur la machine `M3`. Le suffixe entier dans les indicateurs pour `PpFf` et `FastFlow` dans chaque graphe représente donc ce nombre d'instances parallèles d'un *farm*, par exemple `PpFf-4` utilise quatre instances parallèles d'un *farm*. Chaque expérience inclut aussi le programme séquentiel, indiqué sur chaque graphe par `Seq`, utilisant les mêmes fonctions auxiliaires que `PpFf` et `FastFlow`.

Les valeurs pour les unités de mesures — le temps d'exécution, le débit et l'accélération qui ont servi comme référence pour comparer les trois programmes — sont indiquées sur l'axe des y de chaque graphe, alors que le nombre de mots traités est indiqué sur l'axe des x. Afin de connaître l'impact de la quantité de données à traiter, les expériences ont

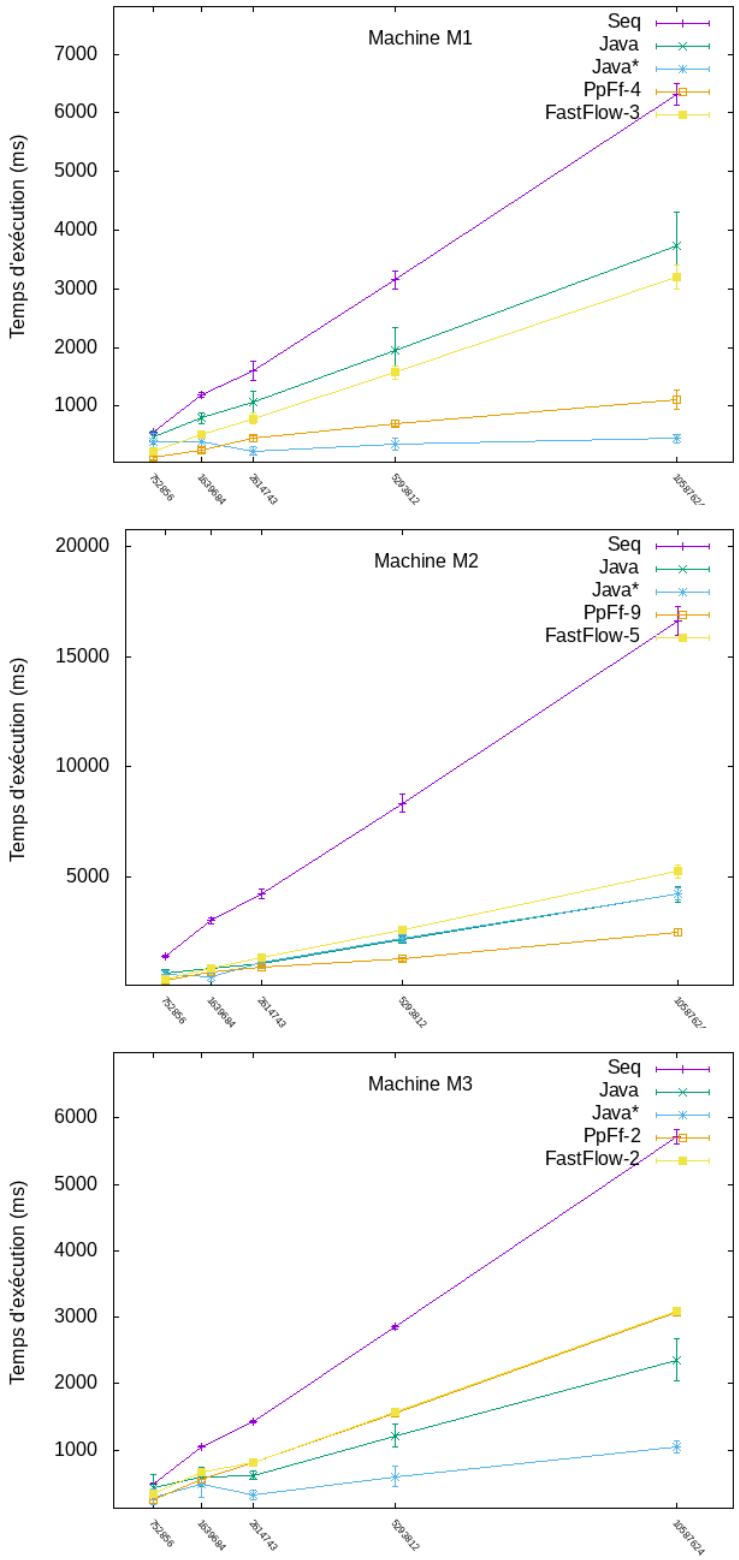

Figure 4.3: Les temps d'exécution des programmes pour `WordCount` sur les machines `M1`, `M2` et `M3`. L'axe des x indique le nombre de mots traités. L'axe des y indique le temps d'exécution, en millisecondes.

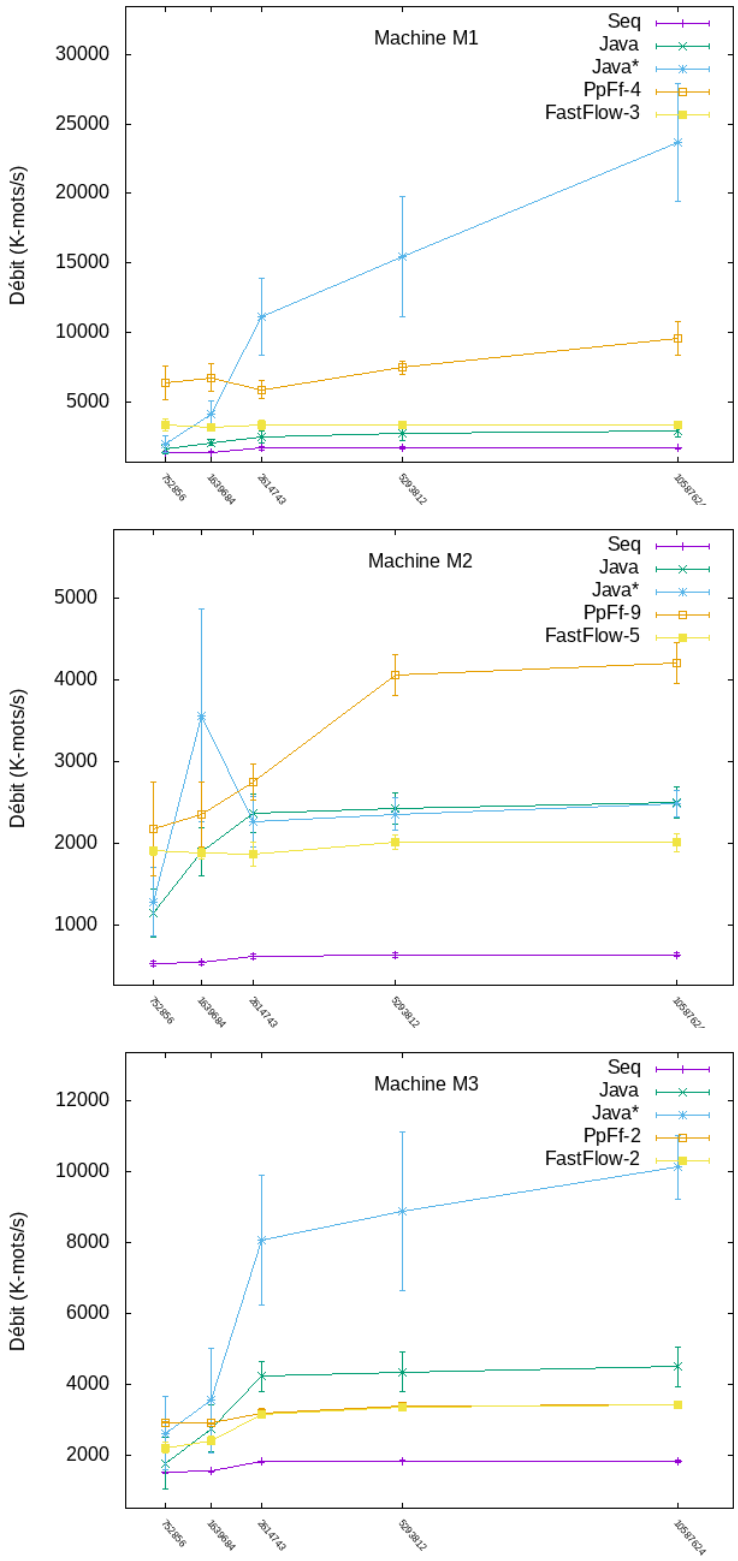

Figure 4.4: Les débits des programmes pour `WordCount` sur les machines `M1`, `M2` et `M3`. L'axe des x indique le nombre de mots traités. L'axe des y indique le nombre de milliers de mots par seconde (K-mots/s).

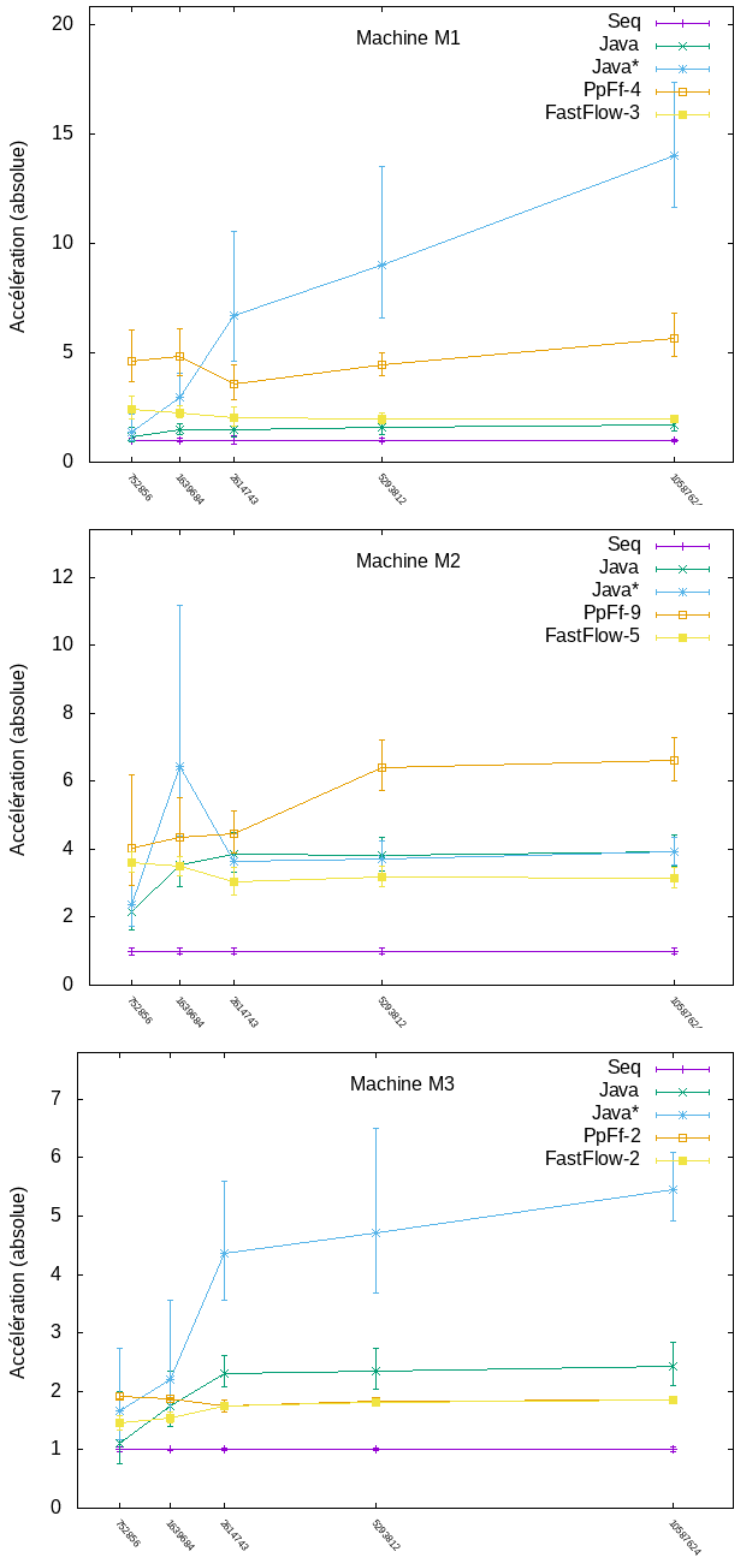

Figure 4.5: Les accélérations des programmes pour `WordCount` sur les machines `M1`, `M2` et `M3`. L'axe des x indique le nombre de mots traités. L'axe des y indique l'accélération absolue par rapport à `WordCountSeq.cpp` (`Seq`).

été menées en utilisant plusieurs ensembles de données. Chaque ensemble de données — un fichier sur disque — contient un nombre croissant de mots. Ces nombres de mots varient de 752 856 à 10 185 035. Les résultats finaux sont des moyennes pour 40 répétitions. Ils sont présentés comme suit :

— Figure 4.3 : les temps d'exécution sur les machines `M1`, `M2` et `M3`.
— Figure 4.4 : les débits sur `M1`, `M2` et `M3`.
— Figure 4.5 : les accélérations par rapport à la version `Seq`entielle sur `M1`, `M2` et `M3`.

Il faut préciser que pour `WordCount`, le résultat n'est pas trié. Du point de vue de la parallélisation, le tri est plutôt lie á l'algorithme de tri et non à la parallélisation. Pour toutes les versions, c'est un *dictionnaire* qui est produit — `unordered_map` en `C++`, `HashMap` en `Java` — et la mesure du temps d'exécution se termine une fois le *dictionnaire* construit.

En comparant les temps d'exécution entre `PpFf` et `Java`, on constate que, pour les machines `M1` et `M3`, `Java` est plus performant que `PpFf`. La gestion de *threads* entre les deux programmes diffère. `PpFf` exécute chaque opération d'un `pipeline` sur un *thread* différent. Par exemple, les cinq opérations de `WordCount` — `source`, `flatMap`, `map`, `find` et `reduceByKey` — s'exécutent sur cinq *threads* différents. Par contre, `Java` gère la création de *threads* par l'intermédiaire du *framework* `fork/join`. Décrit dans le chapitre 1, le *framework* divise une tâche en plus petites sous-tâches indépendantes, et ce récursivement jusqu'à ce qu'elles soient assez simples pour être exécutées, et ce de façon efficace grâce à une approche de *work stealing* (Frigo *et al.*, 1998). Ce mécanisme permet à `Java` d'être plus efficace que `PpFf` sur les machines `M1` et `M3`. `PpFf` est plus rapide que `Java` sur la machine `M2`. La figure 4.4, machine `M2`, montre cet aspect. Par rapport aux machines `M1` et `M3`, `M2` dispose d'un plus grand nombre de processeurs. Cela a permis d'augmenter le nombre d'instances parallèles d'un *farm* dans le programme `PpFf` et en conséquence `PpFf` est plus performant que `Java`.

En comparant les temps d'exécution entre les programmes `PpFf` et `FastFlow`, on constate que `PpFf` obtient presque le même temps d'exécution que `FastFlow` sur la machine

M3 et qu'il performe mieux que FastFlow sur les machines M1 et M2. On rappelle que PpFf est implémenté au–dessus de la bibliothèque FastFlow, et donc il semble que PpFf n'introduise pas de surcoûts par rapport à FastFlow.

Afin de mieux comparer les deux programmes, PpFf et Java, nous avons aussi calculé, à partir des mêmes séries d'expériences, le débit, soit le nombre de mots traité par seconde. Tel qu'on peut le voir dans la figure 4.4, ces débits ont été calculés pour les trois machines. L'axe des x indique le nombre de mots traités et l'axe des y indique le nombre de milliers de mots par seconde (K-mots/s). Un point intéressant, qui peut être observé dans les diagrammes de débits, est que le débit est relativement constant. En prenant comme exemple le diagramme pour PpFf-4 pour M1 de la figure 4.4, on peut noter que le débit reste relativement stable peu importe la taille du fichier. Cela démontre que PpFf est efficace non seulement pour un petit volume de travail, mais aussi pour de grands traitements de données.

La dernière série de résultats tirée de nos expériences vise à comparer les accélérations. Illustrées sur la figure 4.5, les accélérations indiquent à quel point un programme parallèle est plus rapide qu'un programme séquentiel équivalent. En comparant les accélérations du programme PpFf avec celles de FastFlow, on peut observer que PpFf a de meilleures accélérations sur les machines M1 et M2 et une accélération presque identique sur la machine M3. Cela montre que PpFf n'introduit pas des surcoûts par rapport à FastFlow. Par contre, Java a de meilleures accélérations pour M1 et M3, mais pas pour M2. Comme on peut le constater dans le tableau 4.1, la machine M2 dispose d'un plus grand nombre de processeurs par rapport à M1 et M3. Cet avantage est exploité par les instances de *farm*, permettant à PpFf d'obtenir une meilleure accélération que Java sur cette machine.

4.3 Expériences avec l'application StockPrice

Dans le monde informatique actuel, les institutions financières produisent d'énormes quantités d'informations, par ex., des informations sur les marchés boursiers. Un problème important qu'elles rencontrent consiste à trouver des moyens efficaces pour résumer et

visualiser les données afin de produire des informations utiles sur le comportement du marché, notamment pour prendre des décisions d'investissement. Cette section présente une application qui calcule le prix maximum pour diverses actions d'un marché boursier. Des extraits des programmes utilisés pour les expériences pour `StockPrice` en `Java`, `C++` version `Sequentielle`, `PpFf` et `FastFlow` sont présentés dans l'annexe F.

4.3.1 Description de l'application

L'application `StockPrice` calcule le prix d'une action en utilisant le modèle *Black-Scholes* (MacBeth et Merville, 1979). Ce modèle d'évaluation est utilisé pour déterminer le prix juste ou la valeur théorique d'une option d'achat ou de vente, et ce en fonction de six variables telles que la valeur de l'action sous-jacente, le prix d'exercice, le taux d'intérêt sans risque, la volatilité du prix de l'action, la durée et le type d'option. Les calculs associés à cette évaluation sont assez simples en termes algorithmiques, puisqu'il s'agit essentiellement d'une séquence linéaire de calculs points-flottants.

L'application `StockPrice` est composée de cinq opérations principales, comme on peut le voir dans le listing 4.1 qui présente un extrait de la version `PpFf`.

— Une opération `Flow::source` qui définit la source du flux de données. Ici, la source est constituée par les lignes contenues dans un fichier. Le listing 4.2 montre un exemple avec quelques enregistrements tirés d'un des fichiers de données.

  Un enregistrement est identifié par les informations suivantes : le nom de l'action, la valeur actuelle de l'action sous-jacente, le prix d'exercice, le taux d'intérêt sans risque, le taux de dividende, la volatilité du prix de l'action, le temps qu'il reste à l'option avant son échéance (exprimé en années), le type d'option (`C=CALL` : prix pour une option d'achat ; `P=PUT` : prix pour une option de vente), la valeur de dividende et la valeur de référence `DerivaGem`. Les valeurs `DerivaGem`, la valeur et le taux de dividende ne sont pas utilisés dans `StockPrice` pour calculer le prix d'une action.

Listing 4.1: Un extrait du code de `StockPrice.cpp` (version `PpFf`).

```
Reducer<StockAndPrice, double>
   reducer(0.0,
            [](double maxPrice, StockAndPrice sp) {
               return std::max(maxPrice, sp.StockPrice);
            },
            [](double max, double workerResult) {
               return std::max(max, workerResult);
            });
std::unordered_map<std::string, double> currentResult =
    Flow
    ::source(inputFile)
    .parallel(nbFarmWorkers)
    .map<std::string, OptionData>(getOptionData)
    .map<OptionData, StockAndPrice>(calculateStockPrice)
    .reduceByKey<StockAndPrice, std::string, double>(
            reducer,
            [](StockAndPrice* sp) { return &(sp->StockName); } );
```

Listing 4.2: Un exemple illustrant l'information sur des actions contenues dans le fichier.

```
SNY 100.00 90.00 0.1000 0.00 0.10 1.00 C 0.00 18.6308591206674982
JCI 100.00 100.00 0.1000 0.00 0.10 0.50 C 0.00 5.8502736042849798
DSX 100.00 100.00 0.1000 0.00 0.10 1.00 C 0.00 10.3081472436668
LILA 100.00 110.00 0.1000 0.00 0.10 0.10 C 0.00 0.003523074865
NVS 100.00 110.00 0.1000 0.00 0.10 0.50 C 0.00 1.1407228438274099
FLML 100.00 110.00 0.1000 0.00 0.10 1.00 C 0.00 4.216747020308
TEF 100.00 90.00 0.1000 0.00 0.25 0.10 C 0.00 11.1352446183467002
DXB 100.00 90.00 0.1000 0.00 0.25 0.50 C 0.00 16.0926388440922991
HSEA 100.00 90.00 0.1000 0.00 0.25 1.00 C 0.00 21.16345465848
LENS 100.00 100.00 0.1000 0.00 0.25 0.10 C 0.00 3.65996266031
```

— Une opération `parallel` qui répartit les éléments du flux entre divers *threads* de parallélisme de données. Toutes les étapes qui suivent cette opération seront donc exécutées en parallèle, via une *farm*.
— Une opération `map`, qui permet d'extraire le nom et les options de chaque action.
— Une autre opération `map` qui calcule le prix de chaque action. L'algorithme utilisé est celui de *Black-Scholes* (MacBeth et Merville, 1979).
— Une dernière opération, `reduceByKey`, qui extrait le prix maximum pour chaque action.

4.3.2 Mesures obtenues et analyse des résultats

Dans cette section, nous évaluons l'application `StockPrice` en examinant le temps d'exécution, le débit et l'accélération sur les trois machines : `M1`, `M2` et `M3`.

Comme dans le cas de `WordCount`, des expériences préliminaires ont été effectuées afin de choisir les meilleures versions. Dans le cas de `Java`, la meilleure version choisie est celle avec *warmup* et *JIT*. Elle est indiquée dans chaque graphe avec la notation `Java*`. Dans le cas de `PpFf` et `FastFlow`, les expériences ont été menées en variant les nombres d'instances parallèles d'un *farm*. Les meilleures versions choisies pour `PpFf` (resp. `FastFlow`) sont celles utilisant cinq et quatre (resp. quatre et quatre) instances parallèles d'un *farm* sur les machines `M1` et `M2` et deux sur la machine `M3`. Comme pour `WordCount`, le suffixe entier dans les indicateurs pour `PpFf` et `FastFlow` dans chaque graphe représente donc ce nombre d'instances parallèles d'un *farm*. Chaque expérience inclut aussi le programme séquentiel, indiqué sur chaque graphe par `Seq`. Chaque programme calcule le prix maximum pour plusieurs actions. Les temps d'exécution et les débit résultants sont représentés sur l'axe des y de chaque graphe, alors que les nombres de transactions traitées sont représentés sur l'axe des x. Les résultats de expériences finaux sont des moyennes pour 40 répétitions. Ils sont présentés comme suit :

— Figure 4.6 : les temps d'exécution sur les machines `M1`, `M2` et `M3`.
— Figure 4.7 : les débits sur les machines `M1`, `M2` et `M3`.

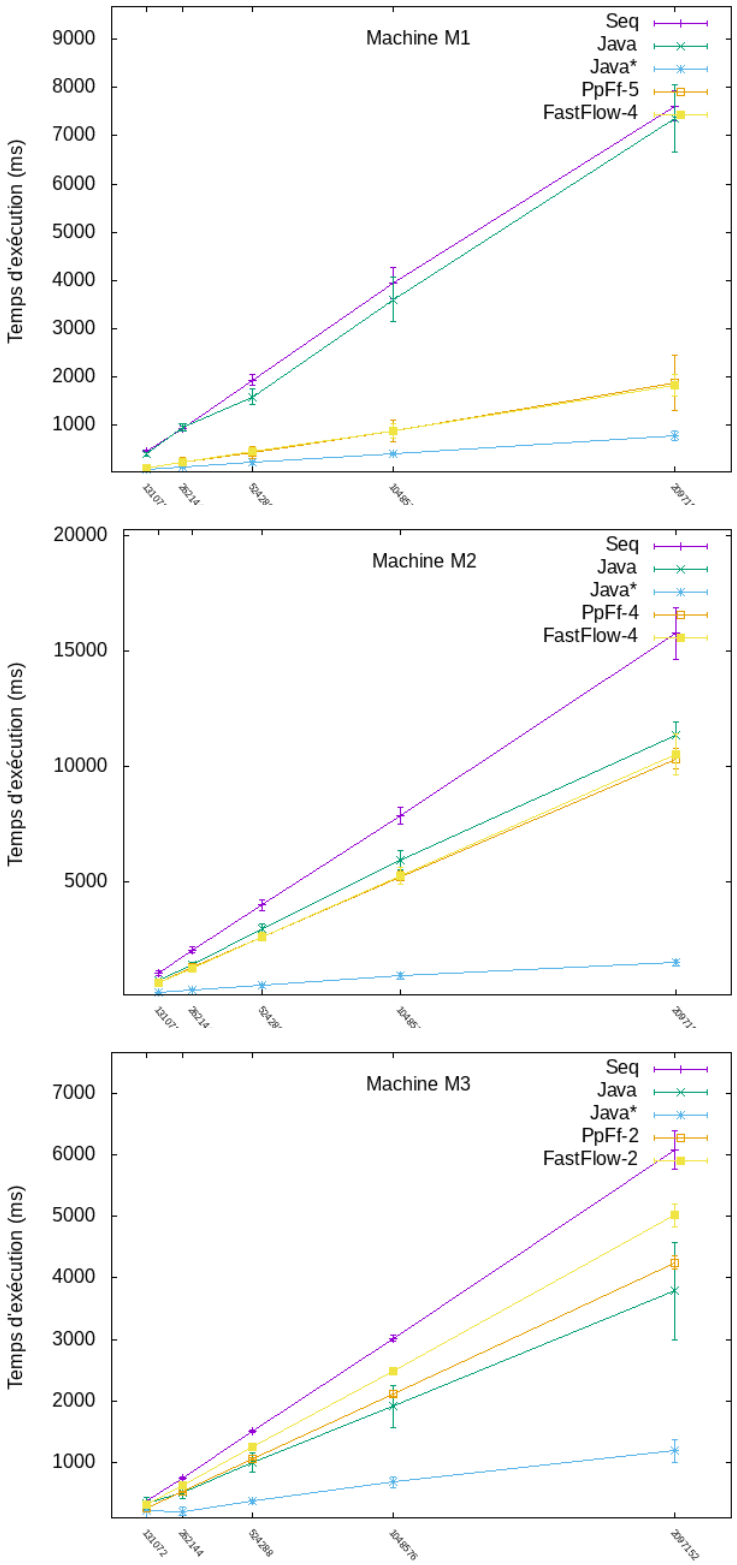

Figure 4.6: Les temps d'exécution des programmes pour `StockPrice` sur les machines `M1`, `M2` et `M3`. L'axe des x indique le nombre de transactions traitées. L'axe des y indique le temps d'exécution, en millisecondes.

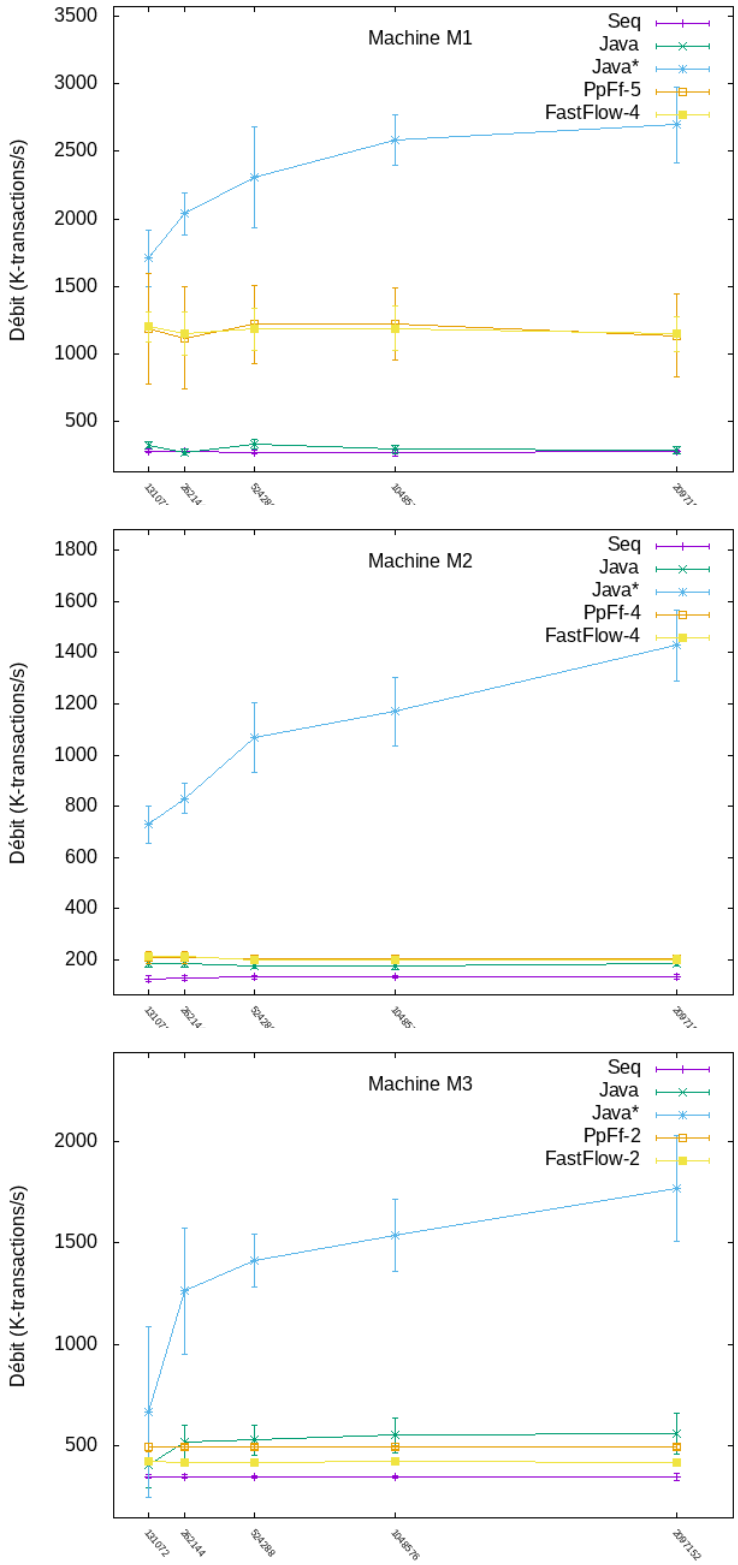

Figure 4.7: Les débits des programmes pour `StockPrice` sur les machines `M1`, `M2` et `M3`. L'axe des x indique le nombre de transactions traitées. L'axe des y indique le nombre de milliers de transactions par seconde (K-transactions/s).

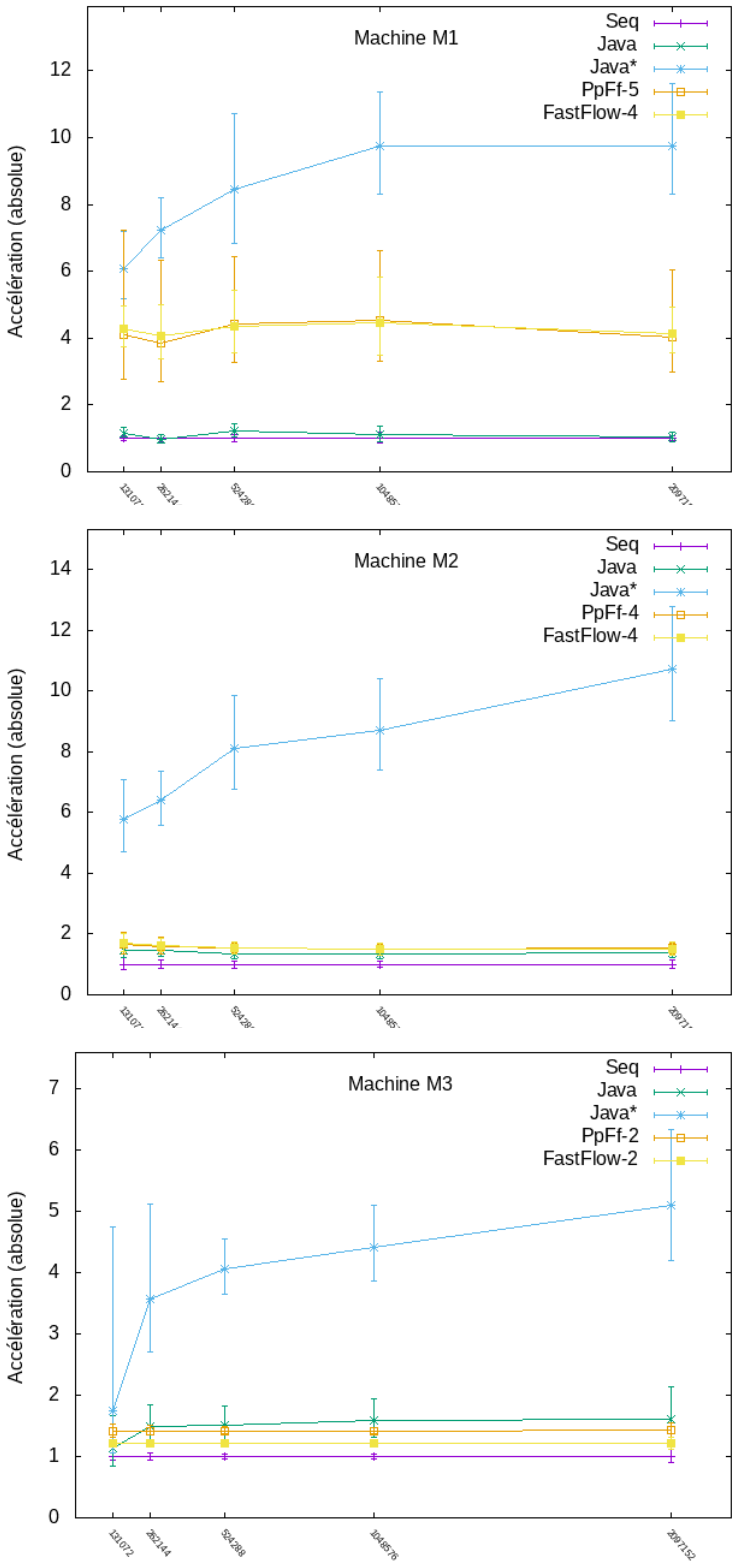

Figure 4.8: Les accélérations des programmes pour `StockPrice` sur les machines `M1`, `M2` et `M3`. L'axe des x indique le nombre de mots traités. L'axe des y indique l'accélération absolue par rapport à `StockPriceSeq.cpp` (`Seq`).

— Figure 4.8 : les accélérations sur les machines `M1`, `M2` et `M3`.

En examinant les temps d'exécution sur les trois machines, on peut constater que `Java` est plus rapide que `PpFf`. On verra dans la section 4.5 pourquoi `Java` est plus performant. Par contre, si on compare `PpFf` et `FastFlow`, les différences des temps d'exécution sont faibles. Cela démontre que les surcoûts introduits par `PpFf` par rapport à `FastFlow` sont négligeables.

À partir des mêmes séries d'expériences, nous avons aussi calculé les débits, soit le nombre de transactions traitées par seconde. Dans la figure 4.7, on trouve les débits moyens, indiqués par les valeurs qui composent la courbe sur le graphe, ainsi que des petites barres verticales qui indiquent la dispersion de débits mesurés ; comme pour `WordCount`, cet intervalle indique la moyenne ± deux fois l'écart-type. Un point intéressant, qui peut être observé dans les graphes de débits, est que la dispersion du débit pour `Java` est plus grande que celle de `PpFf`. Nous avons aussi calculé les accélérations, présentées dans la figure 4.8 : on constate que les accélérations sur `M2` et `M3` ne sont pas très bonnes.

4.4 Autres expériences avec l'application WordCount : Utilisation de `parallel` et nombre de *threads*

On a vu dans la plupart des expériences présentées dans les sections précédentes que notre bibliothèque `PpFf` est généralement moins performante que `Java`. Cela est dû à la gestion de *threads* qui diffère entre les deux programmes. Dans cette section, nous analysons plus en détail le fonctionnement et la gestion de *threads* de `PpFf`.

La création d'un *thread* est une opération coûteuse. La surcharge varie selon les plates-formes, mais la création d'un *thread* requiert toujours l'exécution d'un grand nombre d'instructions additionnelles. Si le traitement effectué par un *thread* est trop petit, il devient inutile d'avoir ce *thread* puisque les surcoûts peuvent être aussi grands que le traitement lui-même. Donc, afin d'amortir les coûts liés à la création d'un *thread*, il faut avoir suffisamment de travail à faire faire par le *thread* pour que cela en vaille la peine.

Les expériences présentées dans cette section vont démontrer que la charge de travail associée à un opérateur influence fortement le temps d'exécution, et que s'il est trop petit, les performances parallèles diminuent.

4.4.1 Description de trois variantes de WordCount

L'application choisie pour ces expériences est l'application `WordCount`, décrite dans la section 2.1. Afin de montrer les coûts associés à la gestion des *threads*, trois versions de `WordCount` vont être comparées : `WordCountSplitted` avec un pipeline comptant cinq opérations, `WordCount` avec quatre opérations, et `WordCountMerged` avec seulement trois opérations. On doit préciser que le résultat obtenu en exécutant les trois versions de `WordCount` est identique et que l'appel à `parallel()` (voir ci-bas) n'est pas compté dans les opérations. Des extraits du code utilisé pour la définition de trois versions sont présentés dans l'annexe G.

Les cinq opérations du pipeline pour `WordCountSplitted` sont les suivantes :

1. `source`, pour extraire et retourner un flux avec les lignes du fichier spécifié en argument.

   Note : Dans tous les cas, l'appel à `source` est immédiatement suivi d'un appel à `parallel`, la méthode qui permet de répartir les éléments du flux entre divers sous-flux et *threads*.

2. `map`, pour décomposer chaque ligne en une collection de mots.
3. `flatten`, pour décomposer une collection de mots en mots individuels.
4. `map`, pour transformer chacun des mots en remplaçant les lettres majuscules en minuscules.
5. `reduceByKey`, pour regrouper les mots similaires ensemble et compter le nombre d'occurrences de chaque mot.

La version `WordCount` est créée à partir de la version `WordCountSplitted` en *fusionnant* les appels aux méthodes `map` et `flatten`, qui sont remplacés par un appel à `flatMap`.

Finalement, la version `WordCountMerged` est créée à partir de `WordCount` en fusionnant les appels aux méthodes `flatMap` et `map`. Plus précisément, ces deux étapes sont fusionnées en combinant en une seule fonction l'argument fourni à `flatMap` et celui fourni à `map`. Donc, comme on peut le noter dans l'annexe G.4, les fonctions `splitInWords` et

`toLowercaseLetters` (passées en argument à `flatMap` et `map`) sont remplacées par la fonction `splitInLowerCaseWords` (passée en argument au `flatMap` de `WordCountMerged`).

Donc, en résumé :

`WordCountSplitted`	=	`source` →	`map` →	`flatten` →	`map` →	`reduceByKey`
`WordCount`	=	`source` →	`flatMap`	→	`map` →	`reduceByKey`
`WordCountMerged`	=	`source` →	`flatMap'`	→		`reduceByKey`

4.4.2 Mesures obtenues et analyse des résultats

Nous allons maintenant évaluer les trois versions de `WordCount` en examinant les temps d'exécution sur la machine `M3`. Deux séries d'expériences ont été menées en variant les nombres d'instances parallèles d'un *farm*. Dans la première série, nous avons utilisé deux instances parallèles d'un *farm* et dans la deuxième seulement une instance. Les résultats sont présentés dans la figure 4.9. Les trois versions `WordCountSplitted`, `WordCount` et `WordCountMerged` sont identifiées sur les graphes par $\texttt{PpFf}_\texttt{s}$, `PpFf` et $\texttt{PpFf}_\texttt{m}$ suivi d'un suffixe entier, qui représente le nombre d'instances parallèles d'un *farm*. Par exemple, $\texttt{PpFf}_\texttt{s}$`-2` représente la version du programme `WordCountSplitted` qui utilise deux instances parallèles d'un *farm* (`parallel(2)`).

On rappelle que, dans `PpFf`, chaque opération ajoutée dans le pipeline de traitement d'un flux s'exécute sur un *thread* différent (parallélisme de flux). Le traitement peut être davantage parallélisé si on utilise aussi les instances parallèles d'un *farm* (parallélisme de données), instances qui sont créées en enchaînant la méthode `parallel`. La valeur entière passée en argument de cette méthode spécifie le nombre d'instances parallèles d'un *farm* entre lesquelles sont répartis les éléments du flux à traiter. Par exemple, si la valeur de l'argument est 1, les versions `WordCountSplitted`, `WordCount` et `WordCountMerged` sont exécutées en utilisant repsectivement cinq, quatre et trois *threads*, correspondant aux opérations (étages) de leur pipeline de traitement. Par contre, si la valeur passée en argument à `parallel` est 2, le flux est divisé en deux sous-flux et chaque opération du pipeline après la méthode `parallel` est exécutée sur un *thread* différent pour

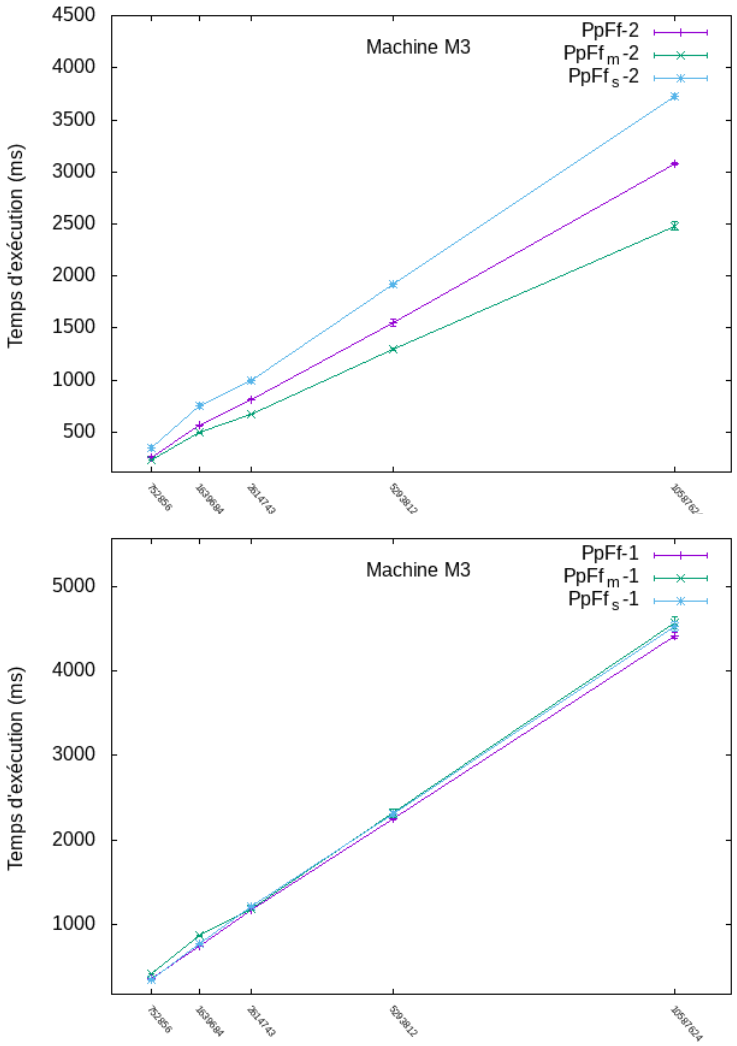

Figure 4.9: Les temps d'exécution des programmes pour `WordCount` sur la machine `M3`. L'axe des x indique le nombre de mots traités. L'axe des y indique le temps d'exécution, en millisecondes.

chaque sous-flux. Par exemple, `WordCountSplitted` avec `parallel(2)` est exécutée en utilisant 9 *threads*, un *thread* correspondant à la méthode `source` et les autres 8 *threads* correspondant aux 4 opérations qui suivent, qui sont toutes exécutées sur deux instances parallèles d'un *farm*.

En comparant les temps d'exécution pour les trois versions de `WordCount` pour la première série d'expériences, avec `parallel(2)`, on constate que `WordCountMerged` est plus performant que les deux autres. Pour le même travail à exécuter, `WordCountMerged` utilise seulement trois opérations par rapport aux versions `WordCount` et `WordCountSplitted` qui utilisent quatre et cinq opérations. Comme chaque opération est exécutée sur un *thread* différent *pour chaque sous-flux créé*, `WordCountMerged` utilise moins de *threads* pour la même tache à accomplir, et donc les coûts introduits par la création des *threads* sont moins importants. Cela peut être observé sur le graphe où, pour une valeur maximale de mots à traiter, `WordCountMerged` obtient un meilleur temps d'exécution que `WordCount` de 0.5s et de plus de 1s que `WordCountSplitted`.

Dans la deuxième série d'expériences, avec l'utilisation de `parallel(1)`, les différences de temps d'exécution entre les trois versions de programmes sont beaucoup plus petites. L'utilisation d'une seule instance parallèle d'un *farm* entraîne l'utilisation d'un seul *thread* supplémentaire entre les versions de programmes. Par exemple, la version `WordCount` exécutée en utilisant quatre *threads* est légèrement plus rapide (4419.6 ms) que la version `WordCountSplitted` qui utilise cinq *threads* (4526.6 ms).

Malgré les coûts qui lui sont associés, le parallélisme de données dans `PpFf` reste important. Cela peut être constaté si on compare les temps moyens d'exécution pour la même série d'expériences, `parallel(1)`, entre la version `WordCountMerged` et les deux autres versions. En utilisant trois *threads*, la version `WordCountMerged` avec un temps d'exécution de 4569.6 ms est moins performante que les versions `WordCount` (4 *threads*) et `WordCountSplitted` (5 *threads*). Cet effet est plus évident si on compare les temps d'exécution entre les deux séries d'expériences. Les versions avec `parallel(2)` — $\texttt{PpFf}_\texttt{s}\texttt{-2}$ (3730.5 ms), `PpFf-2` (3079.8 ms) et $\texttt{PpFf}_\texttt{m}\texttt{-2}$ (2479.9 ms) — sont plus rapides que les versions avec `parallel(1)` — $\texttt{PpFf}_\texttt{s}\texttt{-1}$ (4526.6 ms), `PpFf-1` (4419.6 ms) et $\texttt{PpFf}_\texttt{m}\texttt{-1}$ (4569.6 ms).

4.5 Discussion des résultats et limites de PpFf

Dans ce chapitre, plusieurs expériences ont été présentées dans le but d'évaluer les performances de `PpFf`.

Tout d'abord, deux applications utilisant `PpFf` ont été comparées avec des programmes `Java` équivalents utilisant les `Streams`. Diverses unités de mesures — temps d'exécution, débit et accélération — ont servi comme référence pour comparer ces programmes. Les résultats de ces expériences ont montré que `Java` se comporte généralement mieux, ais pas toujours, que `PpFf`. Ces mêmes expériences ont aussi permis de comparer `PpFf` avec `FastFlow`. Les résultats ont montré que les surcoûts introduits par `PpFf` sont peu élevés.

Ensuite, d'autres expériences ont été menées pour comparer différentes versions du programme `WordCount`, versions créées entièrement à l'aide de la bibliothèque `PpFf`. Ces expériences nous ont montré que lorsqu'un trop grand nombre de *threads* sont créés, le programme devient moins efficace.

Un programme créé en utilisant la bibliothèque `PpFf` peut être décomposé en différents étages (étapes), étages qui représentent les opérations à effectuer dans la chaîne de traitement d'un flux. Conçu dans un style fonctionnel, `PpFf` permet de décomposer un programme en fines tâches. Or, la taille d'une telle tâche — sa granularité — qui représente le travail dans un étage, influence fortement les temps d'exécution du programme. Les résultats des expériences de la section 4.4 ont montré que si la granularité est trop fine et qu'il y a trop de *threads*, le programme résultant sera moins performant.

Donc, la décomposition en étages est importante et utile, mais il faut s'assurer que le travail fait par un étage soit assez gros, sinon les surcoûts du parallélisme deviennent trop grands. Nous reviendrons sur cette question dans la conclusion, dans la partie sur les travaux futurs.

CONCLUSION

Dans ce mémoire, nous avons présenté `PpFf`, une bibliothèque `C++` qui peut être utilisée pour créer des applications de traitement de flux de données. Implémentée par l'intermédiaire de la bibliothèque `FastFlow` et `C++ 17`, la bibliothèque exploite les fonctionnalités modernes du `C++` et les concepts de programmation générique.

Conçue au-dessus de la bibliothèque `FastFlow`, `PpFf` fournit une abstraction de programmation souple et expressive qui facilite le développement d'applications parallèles, masquant ainsi non seulement la complexité des mécanismes de traitement concurrent, mais aussi le modèle et le traitement de données utilisés. L'abstraction dans `PpFf` vise à simplifier la manière dont une séquence de données est visualisée et traitée, permettant ainsi de réutiliser les mêmes opérateurs dans différents contextes et de réutiliser les mêmes algorithmes sur différents modèles de données (par exemple `vector`, `list`, `set` etc.). Ces aspects différencient notre bibliothèque d'autres bibliothèques qui exposent différents types de données à utiliser dans la même interface, ce qui oblige le développeur à réévaluer ses opérations lorsque le contexte change.

Avec une `API` simple, `PpFf` a été conçu pour permettre l'utilisation d'un style fonctionnel. Il expose à l'utilisateur un ensemble de méthodes qui peuvent être chaînées afin de transformer les données dans un style purement fonctionnel à l'aide d'une série de transformations.

Les résultats d'expériences montrent que `PpFf` peut fournir des applications de traitement de flux de données à haut débit. Les performances obtenues par `PpFf` sont dûes en partie au fait qu'il bénéficie directement de la conception optimisée de `FastFlow` pour le traitement de flux.

En exécutants différents programmes, `PpFf` a démontré la facilité avec laquelle son `API` permet de résoudre certains problèmes classiques de traitement de flux de données. Sa

conception permet d'utiliser de nombreux concepts de programmation fonctionnelle, par exemple, les expressions *lambdas* ou l'évaluation paresseuse.

Dans le chapitre sur les expériences, nous avons montré que, dans certains cas, notre bibliothèque pouvait obtenir de meilleurs temps d'exécution qu'un programme `Java` équivalent, avec les `Streams` introduits en Java 8.0, l'un des outils de traitement de données le plus performant de nos jours.

Avec `PpFf`, nous croyons que `C++` a une chance de rivaliser avec `Java` qui est considéré souvent comme plus convivial que `C++`.

Perspectives

Les travaux présentés dans ce mémoire pourraient être étendus dans plusieurs directions, dont certaines sont décrites ci-dessous.

Ajouter de nouvelles méthodes parallèles Afin de construire des requêtes complètes sur une séquence de données, nous pourrions étendre `PpFf` pour prendre en charge davantage de méthodes parallèles de traitement de flux et de données, par exemple, `distinct`, `findAny`, `findFirst`, `forEach`, `of`, etc. Ceci donnerait à l'utilisateur une plus grande flexibilité pour la manipulation de données.

Optimiser la génération du graphe FastFlow L'une des caractéristiques de `PpFf` est que chaque opération dans la chaîne de traitement s'exécute sur un *thread* différent. Comme on l'a vu dans le chapitre sur l'évaluation des performances, cela peut affecter les performances d'un programme si le travail fait par une opération n'est pas assez important pour compenser les surcoûts introduits par la création de *threads*. Afin d'optimiser le graphe généré, il serait intéressant d'implémenter un module qui fusionne des opérations lorsque le travail fait par une opération semble trop court, trop simple. Pour ce faire, il faudrait aussi définir un modèle de coûts, pour estimer cette quantité de travail.

Implémenter un système distribué La version actuelle de `PpFf` est conçue pour fonctionner uniquement en mémoire partagée. Afin d'utiliser la bibliothèque dans un

environnement de données massives (*Big Data*), il faudrait implémenter un module qui gère la distribution des flux. Le module devrait gérer les problèmes liés aux systèmes distribués tels que la coordination des nœuds, la reprise après une panne, l'allocation des tâches, etc. Ce module serait relativement facile à implémenter dans `PpFf` grâce à `FastFlow`, qui prend déjà en charge l'exécution sur des systèmes distribués.

ANNEXE A

MÉTHODES PUBLIQUES DE L'API DE PPFF

Méthode	**Type**	**Description**
Source — Méthodes **statiques** pour créer un flux à partir d'une source		
`Flow::source(string& path)`	`Flow&`	Retourne un flux avec les lignes contenues dans le fichier indiqué par `path`.
`template<T, Iterator>` `Flow::source(Iterator begin, Iterator end)`	`Flow&`	Convertit un conteneur de type STL en un flux.

Suite page suivante

Tableau A.1 Les méthodes publiques de l'API (*suite*)

Méthode	**Type**	**Description**
Transformation — Méthodes pour produire un flux à partir d'un flux existant		
`template<In>` `find(Func<bool(In*)> const& predicate)`	`Flow&`	Retourne les éléments du flux qui satisfont `predicate`.
`template<In, Container, Out>` `flatMap(Func<Container*(In*)> const& taskFunc)`	`Flow&`	Applique la fonction fournie en argument à chaque élément du flux et concatène ces éléments lorsque plusieurs sont produits par la fonction.
`template<In, Out, Container=In>` `flatten()`	`Flow&`	Aplanit un flux multi-niveaux en créant un flux unique à partir du contenu des divers flux/conteneurs.
`template<T>` `limit(int n)`	`Flow&`	Retourne un flux composé des `n` premiers éléments du flux d'entrée.
`template<In, Out>` `map(Func<Out*(In*)> const& taskFunc)`	`Flow&`	Retourne un flux composé de l'application de `taskFunc` à chacun des éléments du flux.
`template<In>` `peek(Func<void(In*)> const& taskFunc)`	`Flow&`	Applique la fonction `taskFunc` à chaque élément du flux et réémet l'élément (sans le modifier) sur le flux de sortie. Note : Utile pour le débogage.

Suite page suivante

Tableau A.1 Les méthodes publiques de l'API (*suite*)

Méthode	**Type**	**Description**
`template<T>` `skip(int n)`	`Flow&`	Retourne un flux composé des éléments du flux d'entrée, mais en omettant les `n` premiers éléments.

Suite page suivante

Tableau A.1 Les méthodes publiques de l'API (*suite*)

Méthode	**Type**	**Description**
Agrégation — Méthodes qui produisent une valeur (scalaire) à partir des éléments d'un flux		
`template<T>` `allMatch(Func<bool(T*)> predicate)`	`bool`	Retourne `true` si tous les éléments du flux satisfont `predicate`, sinon `false`.
`template<T>` `anyMatch(Func<bool(T*)> predicate)`	`bool`	Retourne `true` si au moins un élément du flux satisfait `predicate`, sinon `false`.
`count()`	`unsigned int`	Retourne le nombre d'éléments du flux.
`template<T>` `max(Func<void(T*, T*)> compare)`	`T`	Retourne l'élément maximum du flux en fonction du comparateur.
`template<T>` `min(Func<void(T*, T*)> compare)`	`T`	Retourne l'élément minimum du flux en fonction du comparateur.
`template<T>` `noneMatch(Func<bool(T*)> predicate)`	`bool`	Retourne `true` si aucun des éléments du flux ne satisfait `predicate`, sinon `false`.
`template<In, Out=In>` `reduce(Reducer<In, Out> const& reducer)`	`Out`	Effectue une réduction sur les éléments du flux. Voir la notion de `Reducer`, Section 2.5.3.

Suite page suivante

Tableau A.1 Les méthodes publiques de l'API (*suite*)

Méthode	**Type**	**Description**
`template<In, Out=In>` `reduce(Out init, Func<Out(In, Out)> acc)`	`Out`	Effectue une réduction des éléments du flux, en utilisant `init` comme valeur initiale et `acc` comme fonction d'accumulation.
`template<T>` `sum()`	`T`	Retourne la somme des éléments du flux.

Suite page suivante

Tableau A.1 Les méthodes publiques de l'API (*suite*)

Méthode	**Type**	**Description**
Agrégation — Méthodes qui produisent une collection à partir d'un flux		
`template<T, Container<T>>` `collect()`	`Container<T>`	Retourne un conteneur STL avec tous les éléments du flux.
`template<In, K=In, V=In, MapType>` `groupByKey(Func<K*(In*)> fk, Func<V*(In*)> fv)`	`MapType`	Retourne un dictionnaire (*map*) avec les éléments du flux regroupés par clé.
`template<In, K=In, V=In, MapType>` `reduceByKey(Reducer<In, V> r, Func<K*(In*)> fk)`	`MapType`	Effectue une réduction sur les valeurs de chaque clé à l'aide d'un `Reducer`. Voir la notion de `Reducer`, Section. 2.5.3.
`template<T>` `sort(Func<bool(T, T)> const& compare)`	`Collection<T, Container>`	Effectue le tri des éléments du flux selon l'ordre spécifié par `compare`. Note : Le premier élément du flux de sortie n'est émis *qu'après que la fin de flux ait été rencontrée.*

Suite page suivante

Tableau A.1 Les méthodes publiques de l'API (*suite*)

Méthode	**Type**	**Description**
Execution — Méthode pour l'exécution parallèle du flux		
`parallel(int workers = 1)`	`Flow&`	Spécifie le nombre de travailleurs à utiliser pour traiter les éléments du flux.

ANNEXE B

SPÉCIFICATIONS SEMI-FORMELLES DU TYPE REDUCER ET DE LA MÉTHODE GROUPBYKEY

Les descriptions (semi-)formelles présentées dans cette annexe ont été formulées par mon directeur de recherche, le Prof. Guy Tremblay, et elles sont présentées par souci de complétude.

B.1 Description (semi-)formelle d'un `Reducer`

Soit les éléments suivants :

- T et R des types de données ;
- $S = [s_0, s_1, \ldots, s_k]$, un flux de données, où $s_i \in T$ $(i = 0, \ldots, k)$;
  Soit $S_0, \ldots, S_n$ une partition de S ;
- Soit $v_0 \in R$;
- Soit $acc : R \times T \to R$;
- Soit $comb : R \times R \to R$;

Soit red un `Reducer` avec les attributs $v_0, acc, comb$.

Alors S`.reduce(`red`)`

```
S.reduce( Reducer v0 acc comb )
  = foldr comb r0 [r1, ..., rk-1]
    where
      [S0, S1, ..., Sk-1] = splitIntoSubstreams S k
      [r0, r1, ..., rk-1] = [foldr acc v0 Si | i <- [0, ..., k-1]]
```

```
foldr f a []     = a
foldr f a (x:xs) = f x (foldr f a xs)
```

B.2 Description (semi-)formelle de `GroupByKey`

Dénotons comme suit un *map* M qui associe la clé c_i à la valeur v_i, pour $i = 0, \ldots, n$:[1]

— $M = \{c_i \mapsto v_i \mid 0 \leq i \leq n\}$

Soit alors les éléments suivants :

— T, un type de données ;

— $S = [s_0, s_1, \ldots, s_k]$, un flux de données, où $s_i \in T$ $(i = 0, \ldots, k)$;

— $fc : T \to C$, une fonction sur T qui produit une «clé» de type C ;

— $fv : T \to V$, une fonction sur T qui retourne une «valeur» de type V ;

Le résultat d'un appel pour le flux S de la méthode `groupByKey` avec deux arguments peut alors être décrit comme suit :

— $S.\texttt{groupByKey}(fc, fv) = \{c \mapsto vals(c, S) \mid c \in cles(S)\}$

Où

— $cles(S) = \{fc(s_i) \mid s_i \in S\}$

— $vals(c, S) = \{fv(s_i) \mid s_i \in S \wedge fc(s_i) = c\}$

Quant à la méthode `groupByKey` avec un seul argument, elle est équivalente à celle avec deux arguments, mais où le deuxième est simplement la fonction *id*entité :

— $S.\texttt{groupByKey}(fc) = S.\texttt{groupByKey}(fc, id)$

Où $id(x) = x$

1. Dans du code `C++`, un tel *map* serait représenté comme suit : $\{\{c_0, v_0\}, \ldots, \{c_n, v_n\}\}$

ANNEXE C

EXTRAIT DU SCRIPT DE CONFIGURATION DES EXPÉRIENCES POUR WORDCOUNT

Cette section présente un **extrait** du fichier de configuration pour les expériences pour les programmes `WordCount`. Ce fichier de configuration permet de spécifier les paramètres nécessaires pour l'exécution des expériences : le nom de la machine, les quantités de données à utiliser, le nombre de répétitions (pour le calcul de la moyenne et de l'écart-type), et les programmes à exécuter.

Listing C.1: Un extrait d'un fichier de configuration pour les expériences pour les programmes `WordCount*.*` : fichier `WordCount-bm-config.rb` .

```
#####################################################################
# Les divers programmmes et commandes pour les executer.
#####################################################################
Program.define( 'Seq', "./#{PGM}Seq" )

Program.define( 'Java-', "java -cp . #{PGM} 10" )  # Seq. sans warmup
Program.define( 'Java',  "java -cp . #{PGM} 11" )  # Seq. avec warmup
Program.define( 'Java+', "java -cp . #{PGM} 20" )  # Par. sans warmup
Program.define( 'Java*', "java -cp . #{PGM} 21" )  # Par. avec warmup

[*1..8].each do |k|
  Program.define( "PpFf-#{k}", "./#{PGM} #{k}" )
end

[*1..8].each do |k|
  Program.define( "FastFlow-#{k}", "./#{PGM}FastFlow #{k}" )
end

#####################################################################
# Diverses quantites de donnees et la fonction pour les obtenir.
#####################################################################
def fichier_de_donnees_pour(nb_items)
  "testdata/#{nb_items}Words.txt"
end

peu_de_donnees = [3805, 7610]

donnees_preliminaires =  [78792, 281307, 752856, 1639684]

pas_mal_de_donnees = [78792, 281307, 752856, 1639684, 2137758]

beaucoup_de_donnees = [752856, 1639684, 2614743, 5293812, 10587624]
#
```

```
#####################################################################
# Les experiences.
#####################################################################

#####################################################
# Experiences faites sur les machines indiquees pour identifier les
# programmes de chaque groupe (Java, PpFf et FastFlow) qui sont les
# plus performants.
#####################################################
Experience.define( 11,
                   nb_items: beaucoup_de_donnees,
                   nb_repetitions: 20,
                   programs: ['Java-', 'Java', 'Java+', 'Java*']
                 )

Experience.define( 2,
                   machines: ['c34581', 'java'],
                   nb_items: donnees_preliminaires,
                   nb_repetitions: 10,
                   programs: ['PpFf-1', 'PpFf-2', 'PpFf-3']
                 )

Experience.define( 3,
                   machines: ['c34581', 'java'],
                   nb_items: donnees_preliminaires,
                   nb_repetitions: 10,
                   programs: ['FastFlow-1', 'FastFlow-2', 'FastFlow-3']
                 )

Experience.define( 30,
                   machines: ['japet'],
                   nb_items: pas_mal_de_donnees,
                   nb_repetitions: 10,
                   programs: ['FastFlow-2', 'FastFlow-4', 'FastFlow-6', 'FastFlow-8']
                 )
```

```
# [...]

NB_REPETITIONS_IC = 40
DONNEES_IC = beaucoup_de_donnees

Experience.define( 3001,
                   machines: ['java'],
                   nb_items: DONNEES_IC,
                   nb_repetitions: NB_REPETITIONS_IC,
                   programs: ['Seq', 'Java', 'Java*', 'PpFf-2', 'FastFlow-2']
                 )

Experience.define( 3002,
                   machines: ['japet'],
                   nb_items: DONNEES_IC,
                   nb_repetitions: NB_REPETITIONS_IC,
                   programs: ['Seq', 'Java', 'Java*', 'PpFf-8', 'FastFlow-5']
                 )

Experience.define( 3003,
                   machines: ['MacOS', 'c34581'],
                   nb_items: DONNEES_IC,
                   nb_repetitions: NB_REPETITIONS_IC,
                   programs: ['Seq', 'Java', 'Java*', 'PpFf-1', 'FastFlow-1']
                 )
```

ANNEXE D

RÉSULTATS DES EXPÉRIENCES PRÉLIMINAIRES POUR ÉVALUER LES PARAMÈTRES D'EXÉCUTION SUR LES DIVERSES MACHINES POUR LES PROGRAMMES WORDCOUNT.JAVA ET STOCKPRICE.JAVA

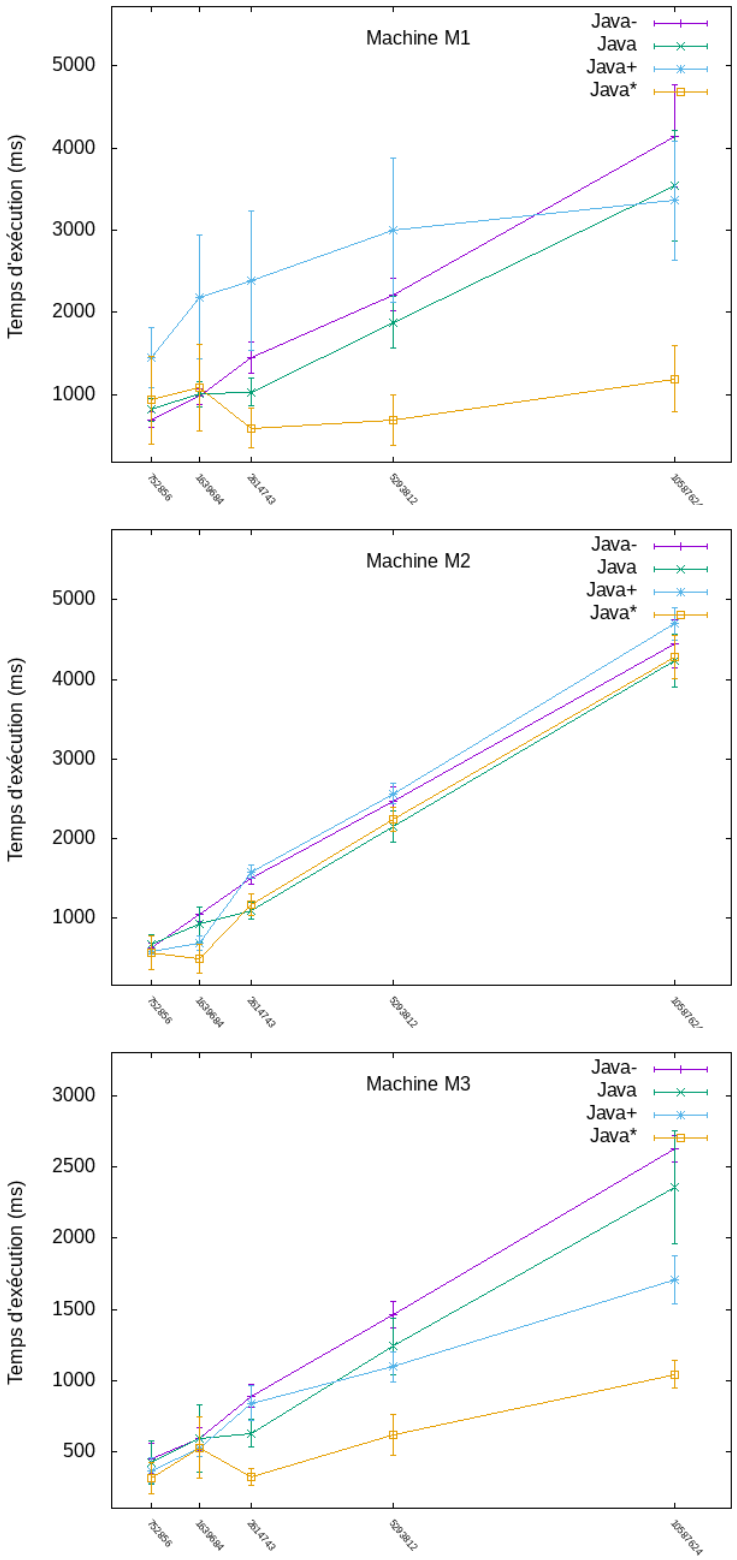

Figure D.1: Les temps pour les diverses variantes de `WordCount.java` sur les machines `M1`, `M2` et `M3`.

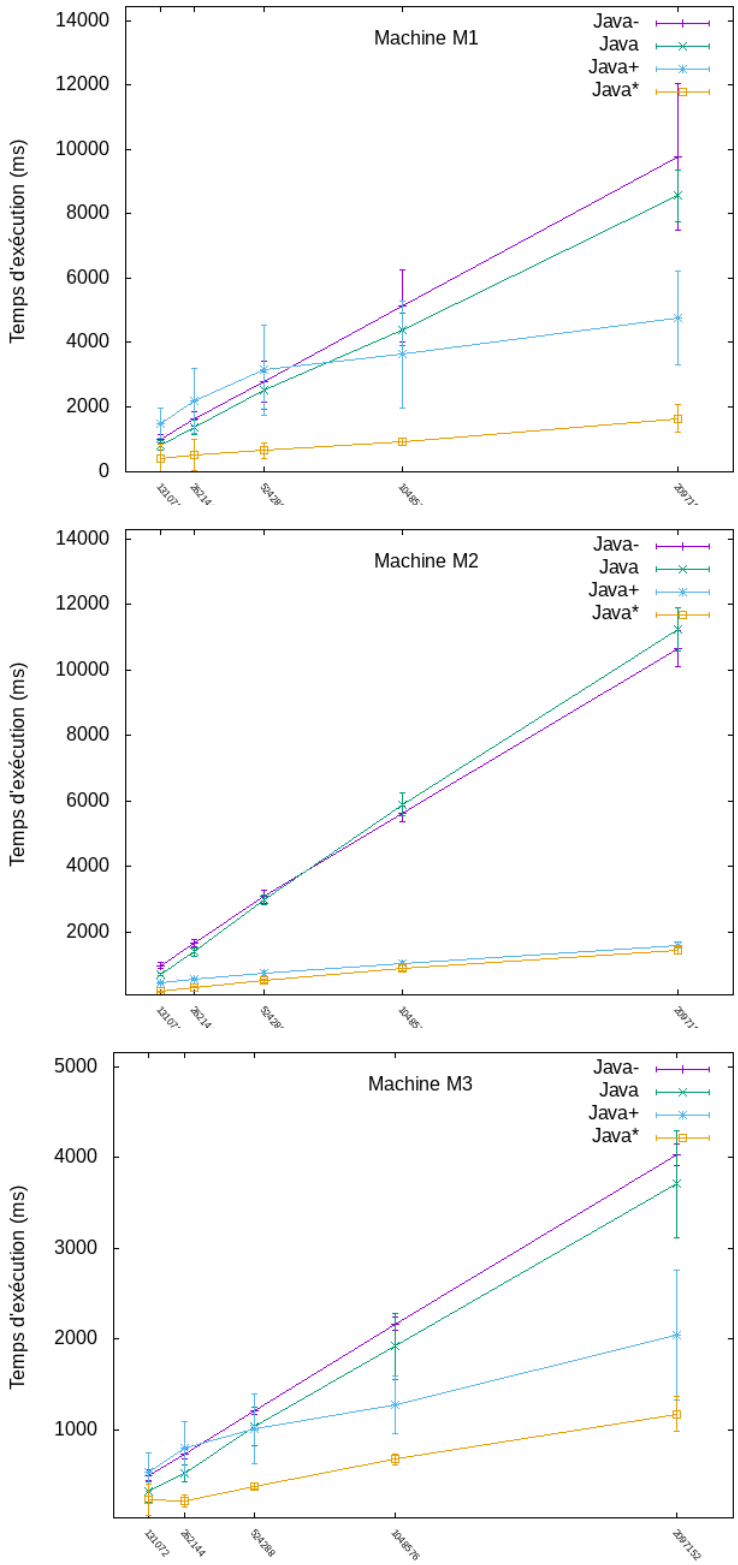

Figure D.2: Les temps pour les diverses variantes de `StockPrice.java` sur les machines `M1`, `M2` et `M3`.

ANNEXE E

EXTRAITS DES PROGRAMMES UTILISÉS POUR LES EXPÉRIENCES AVEC WORDCOUNT

E.1 Programme WordCount.java

```
class ReduceByKey implements Consumer<String> {
    @SuppressWarnings("unchecked")
    private Map<String,Integer> map = (Map) new HashMap();

    public void accept( String s ) {
        map.put(s, map.getOrDefault(s, 0) + 1);
    }

    public void combine( ReduceByKey other ) {
        for ( Map.Entry<String,Integer> entry : other.map.entrySet() ) {
            Integer value = map.getOrDefault(entry.getKey(), 0);
            map.put( entry.getKey(), value + entry.getValue() );
        }
    }

    public Map<String,Integer> toMap() {
        return map;
    }
}
```

```
public class WordCount {
    public static Stream<String> splitInWords(String line) {
        return Arrays
            .stream( line.trim().split("[^a-zA-Z]") )
            .filter( word -> word.length() > 0 );
    }

    public static String toLowerCaseLetters(String word) {
        return word.toLowerCase();
    }

    public static void main(String[] args) throws IOException {
        // ...
        long startTime = System.nanoTime();

        Map<String,Integer> wordsCount =
            Files.lines( Paths.get(inputFile) )
            .parallel()
            .flatMap( WordCount::splitInWords )
            .map( WordCount::toLowerCaseLetters )
            .collect( ReduceByKey::new,
                      ReduceByKey::accept,
                      ReduceByKey::combine )
            .toMap();

        long duration = (System.nanoTime() - startTime);
        double milliseconds = (double) duration / 1000000;
        // ...
    }
}
```

E.2 Fonctions auxiliaires communes aux programmes WordCountSeq.cpp, WordCount.cpp et WordCountFastFlow.cpp

```
Words* splitInWords(std::string* line) {
    Words* words = new Words();
    size_t start = 0, next_letter;
    while ( start < line->length() ) {
        for ( next_letter = start;
              next_letter < line->length() && !isalpha(line->at(next_letter));
              next_letter++ )
            {};
        if ( next_letter >= line->length() ) break;
        for ( start = next_letter + 1;
              start < line->length() && isalpha(line->at(start)); start++ )
            {};
        std::string word = line->substr(next_letter, start - next_letter);
        words->push_back( word );
    };

    return words;
}

std::string* toLowercaseLetters(std::string* data) {
    std::for_each( data->begin(),
                   data->end(),
                   [](char& c) { c = ::tolower(c); } );

    return data;
}
```

E.3 Programme WordCountSeq.cpp (version séquentielle)

```
// ...
auto begin = std::chrono::high_resolution_clock::now();

std::unordered_map<std::string, size_t> currentResult;

std::ifstream file(inputFile);
std::string* line = new std::string;
while (std::getline(file, *line)) {
    Words* words = splitInWords(line);
    for (auto word = words->begin(); word != words->end(); word++) {
        std::string* wordLC = toLowercaseLetters(&(*word));
        currentResult[*wordLC] += 1;
    }
    line = new std::string;
}

auto end = std::chrono::high_resolution_clock::now();
long duration_ms =
   std::chrono::duration_cast<std::chrono::milliseconds>(end-begin).count();
// ...
```

E.4 Programme WordCount.cpp (version `PpFf`)

```
// ...
auto begin = std::chrono::high_resolution_clock::now();

Reducer<std::string, int> reducer(
          0,
          [](int count, std::string _) { return count + 1; },
          std::plus<int>{}
        );

std::unordered_map<std::string, int> currentResult =
    Flow
    ::source(inputFile)
    .parallel(farmParallelism)
    .flatMap<std::string, Words, std::string>(splitInWords)
    .map<std::string, std::string>(toLowercaseLetters)
    .reduceByKey<std::string, std::string, int>(reducer);

auto end = std::chrono::high_resolution_clock::now();
long duration_ms =
    std::chrono::duration_cast<std::chrono::milliseconds>(end-begin).count();
// ...
```

E.5 Programme WordCountFastFlow.cpp

```
struct linesFromFileStage : ff_node {
    std::string const &path;
    linesFromFileStage(std::string const &path) : path(path){}

    void* svc(void*) {
        std::ifstream file(path);

        std::string* line = new std::string;
        while (std::getline(file, *line)) {
            ff_send_out(line);
            line = new std::string;
        }
        return EOS;
    }
};

struct splitInWordsStage : ff_node_t<std::string, Words> {
    Words* svc(std::string* task) {
        return splitInWords(task);
    }
};

struct flatStage : ff_node_t<Words,std::string> {
    std::string* svc(Words* task) {
        for (auto &elem: *task) {
            ff_send_out(&elem);
        }
        return GO_ON;
    }
};
```

```
struct toLowercaseLettersStage : ff_node_t<std::string> {
    std::string* svc(std::string* task) {
        return toLowercaseLetters(task);
    }
};

struct groupByKeyStage : ff_node_t<std::string,void> {
    typedef std::unordered_map<std::string, int> CONTAINER;
    CONTAINER& container;

    groupByKeyStage(CONTAINER& container) : container(container){}

    void* svc(std::string* task) {
        container[*task] += 1;
        return GO_ON;
    }
};
```

```
int main(int argc, char *argv[]) {
    // ...
    auto begin = std::chrono::high_resolution_clock::now();

    std::vector<ff_node*> workers;
    for ( uint32_t i = 0; i < nbFarmWorkers; i++ ) {
        workers.push_back( new ff_Pipe<>( new splitInWordsStage,
                                          new flatStage,
                                          new toLowercaseLettersStage ) );
    }

    std::unordered_map<std::string, int> result;
    ff_Pipe<> ffp( new linesFromFileStage(inputFile),
                   new ff_farm(workers),
                   new groupByKeyStage(result) );

    if (ffp.run_and_wait_end() < 0)
        error("running pipe");

    auto end = std::chrono::high_resolution_clock::now();
    long duration_ms =
      std::chrono::duration_cast<std::chrono::milliseconds>(end - begin).count(
    // ...
}
```

ANNEXE F

EXTRAITS DES PROGRAMMES UTILISÉS POUR LES EXPÉRIENCES AVEC STOCKPRICE

F.1 Programme StockPrice.java

```
Map<String, Double> stockPrice =
  Files.lines( Paths.get(inputFile) )
       .parallel()
       .map( StockPrice::getOptionData )
       .map( StockPrice::calculateStockPrice )
       .collect( GroupByKey::new, GroupByKey::accept, GroupByKey::combine )
       .toMap();
```

F.2 Programme StockPriceSeq.cpp (version séquentielle)

```
std::unordered_map<std::string, double> currentResult;
std::ifstream file(inputFile);
std::string* line = new std::string;
while (std::getline(file, *line)) {
    OptionData* od = getOptionData(line);
    StockAndPrice* sp = calculateStockPrice(od);

    auto mapIt = currentResult.find(sp->StockName);
    if (mapIt != currentResult.end()) {
        mapIt->second = std::max(sp->StockPrice, mapIt->second);
    } else {
        currentResult[sp->StockName] = sp->StockPrice;
    }

    line = new std::string;
}
```

F.3 Programme StockPrice.cpp (version PpFf)

```
Reducer<StockAndPrice, double>
   reducer(0.0,
           [](double maxPrice, StockAndPrice sp) {
              return std::max(maxPrice, sp.StockPrice);
           },
           [](double max, double workerResult) {
              return std::max(max, workerResult);
           });
std::unordered_map<std::string, double> currentResult =
    Flow
    ::source(inputFile)
    .parallel(nbFarmWorkers)
    .map<std::string, OptionData>(getOptionData)
    .map<OptionData, StockAndPrice>(calculateStockPrice)
    .reduceByKey<StockAndPrice, std::string, double>(
           reducer,
           [](StockAndPrice* sp) { return &(sp->StockName); } );
```

F.4 Programme StockPriceFastFlow.cpp

```
std::vector<ff_node*> workers;
for (uint32_t i = 0; i < nbFarmWorkers; i++) {
    workers.push_back(  new ff_Pipe<>( new GetOptionDataStage(),
                                       new CalculateStockPriceStage() ) );
}

std::unordered_map<std::string, double> result;
ff_Pipe<> ffp( new LinesFromFileStage(inputFile),
               new ff_farm(workers),
               new ReduceByKeyStage(result) );

if (ffp.run_and_wait_end()<0)
    error("running pipe");
```

ANNEXE G

EXTRAITS DES PROGRAMMES UTILISÉS POUR LES EXPÉRIENCES AVEC LES TROIS VERSIONS DE WORDCOUNT

G.1 Fonctions auxiliaires utilisées par les programmes

```
Words* splitInWords(std::string* line) {
    Words* words = new Words();
    size_t start = 0, next_letter;
    while ( start < line->length() ) {
        for ( next_letter = start;
              next_letter < line->length()
                && !isalpha(line->at(next_letter));
              next_letter++ )
            {};
        if ( next_letter >= line->length() ) break;
        for ( start = next_letter + 1;
              start < line->length()
                && isalpha(line->at(start)); start++ )
            {};
        std::string word = line->substr(next_letter, start - next_letter);
        words->push_back( word );
    };

    return words;
}
```

```
std::string* toLowercaseLetters(std::string* data) {
    std::for_each( data->begin(),
                   data->end(),
                   [](char& c) { c = ::tolower(c); } );

    return data;
}

Words* splitInLowerCaseWords(std::string* line) {
    Words* words = new Words();
    size_t start = 0, next_letter;
    while ( start < line->length() ) {
        for ( next_letter = start;
              next_letter < line->length() && !isalpha(line->at(next_letter));
              next_letter++ )
            {};
        if ( next_letter >= line->length() ) break;
        for ( start = next_letter + 1;
              start < line->length() && isalpha(line->at(start)); start++ )
            {};
        words->emplace_back( *line, next_letter, start - next_letter );
        toLowercaseLetters(&words->back());
    };

    return words;
}
```

G.2 Programme WordCountSplitted.cpp

```
Reducer<std::string, int> reducer(0,
                                  [](int count, std::string _)
                                    { return count + 1; },
                                  std::plus<int>{} );

std::unordered_map<std::string, int> currentResult =
    Flow
    ::source(inputFile)
    .parallel(farmParallelism)
    .map<std::string, Words>(splitInWords)
    .flatten<Words, std::string>()
    .map<std::string, std::string>(toLowercaseLetters)
    .reduceByKey<std::string, std::string, int>(reducer);
```

G.3 Programme WordCount.cpp

```
Reducer<std::string, int> reducer(0,
                                  [](int count, std::string _)
                                    { return count + 1; },
                                  std::plus<int>{} );

std::unordered_map<std::string, int> currentResult =
    Flow
    ::source(inputFile)
    .parallel(farmParallelism)
    .flatMap<std::string, Words, std::string>(splitInWords)
    .map<std::string, std::string>(toLowercaseLetters)
    .reduceByKey<std::string, std::string, int>(reducer);
```

G.4 Programme WordCountMerged.cpp

```
Reducer<std::string, int> reducer(0,
                                  [](int count, std::string _)
                                    { return count + 1; },
                                  std::plus<int>{} );

std::unordered_map<std::string, int> currentResult =
    Flow
    ::source(inputFile)
    .parallel(farmParallelism)
    .flatMap<std::string, Words, std::string>(splitInLowerCaseWords)
    .reduceByKey<std::string, std::string, int>(reducer);
```

BIBLIOGRAPHIE

Aldinucci, M., Danelutto, M., Kilpatrick, P. et Torquati, M. (2014). FastFlow : High-level and efficient streaming on multi-core. In S. Pllana et F. Xhafa (dir.), *Programming Multi-core and Many-core Computing Systems*. Wiley.

Aldinucci, M., Meneghin, M. et Torquati, M. (2010). Efficient Smith-Waterman on multi-core with FastFlow. Dans *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, 195–199. IEEE.

Amarasinghe, S., Hall, M., Lethin, R., Pingali, K., Quinlan, D., Sarkar, V., Shalf, J., Lucas, R. et Yelick, K. (2011). ASCR programming challenges for exascale computing. Dans *Report of the 2011 Workshop on Exascale Programming Challenges*.

Apache (2012). Apache Spark. Lightning-fast unified analytics engine. http://spark.apache.org/. Accessed on 12.05.2019.

Apache (2014). Flink. https://flink.apache.org. Accessed on 07.21.2018.

Aumage, O., D., B., Counilh, M.-C., Furmento, N., Namyst, R., S., T. et Wacrenier, P.-A. (2018a). StarPU. A Unified Runtime System for Heterogeneous Multicore Architectures. http://starpu.gforge.inria.fr/. Accessed on 06.14.2020.

Aumage, O., Furmento, N. et Thibault, S. (2018b). The StarPU runtime system. http://starpu.gforge.inria.fr/tutorials/2016-06-PATC/slides/02_mastering_starpu.pdf. Accessed on 06.14.2020.

Beard, J. (2016). RaftLib. Accessing data from within a kernel. https://https://github.com/RaftLib/RaftLib/wiki/Accessing-data-from-within-a-kernel. Accessed on 07.06.2020.

Beard, J. C., Li, P. et Chamberlain, R. D. (2017). RaftLib : a C++ template library for high performance stream parallel processing. *The International Journal of High Performance Computing Applications*, *31*(5), 391–404.

Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J. et Menon, R. (2001). *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers.

Cramer, T., Friedman, R., Miller, T., Seberger, D., Wilson, R. et Wolczko, M. (1997). Compiling Java just in time. *IEEE Micro*, *17*(3), 36–43.

Dean, J. et Ghemawat, S. (2004). MapReduce : Simplified data processing on large clusters. Dans *OSDI'04 : Sixth Symposium on Operating System Design and Implementation*, 137–149., San Francisco, CA.

Ernstsson, A. et Kessler, C. (2015). SkePU. Autotunable Multi-Backend Skeleton Programming Framework for Multicore CPU and Multi-GPU Systems. https://www.ida.liu.se/labs/pelab/skepu/skepu1/. Accessed on 06.21.2020.

Farber, R. (2016). *Parallel programming with OpenACC*. Newnes.

Frampton, M. (2015). *Mastering Apache Spark*. Packt Publishing Ltd.

Frigo, M., Leiserson, C. et Randall, K. (1998). The implementation of the Cilk-5 multithreaded language. Dans *PLDI '98 Proc. of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, 212–223. ACM.

Haque, W. (2006). Concurrent deadlock detection in parallel programs. *International Journal of Computers and Applications*, *28*(1), 19–25.

Hudak, P. et Wadler, P. (1990). *Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.0)*. Rapport technique YALEU/DCS/RR777, Yale University, Department of Computer Science.

Hutton, G. (2016). *Programming in Haskell*. Cambridge University Press.

Josuttis, N. M. (2012). *The C++ standard library : a tutorial and reference*. Addison-Wesley.

Leiserson, C. et Plaat, A. (1998). Programming parallel applications in Cilk. *SINEWS : SIAM News*, *31*(4), 6–7.

MacBeth, J. D. et Merville, L. J. (1979). An empirical examination of the Black-Scholes call option pricing model. *The journal of finance*, *34*(5), 1173–1186.

Oracle (2017). Oracle. Lesson : Aggregate Operations. https://docs.oracle.com/javase/tutorial/collections/streams/. Accessed on 08.12.2018.

Oracle-Documentation (2018). Oracle. Java Stream API. https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html. Accessed on 09.12.2018.

Reinders, J. (2007). *Intel Threading Building Blocks : Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media.

Salloum, S., Dautov, R., Chen, X., Peng, P. X. et Huang, J. Z. (2016). Big data analytics on Apache Spark. *International Journal of Data Science and Analytics*, *1*(3), 145–164.

Steele, G. (1984). *Common LISP : The Language*. Digital Press.

Torquati, M. (2015). *Parallel Programming Using FastFlow*. Computer Science Department, University of Pisa.

Urma, R.-G. et al. (2014). *Java 8 in action*. Manning publications.

Warburton, R. (2014). *Java 8 Lambdas : Pragmatic Functional Programming*. O'Reilly Media, Inc.

Wu, Z., Lu, K., Wang, X., Zhou, X. et Chen, C. (2015). Detecting harmful data races through parallel verification. *The Journal of Supercomputing*, *71*(8), 2922–2943.

Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S. et Stoica, I. (2013). Discretized streams : Fault-tolerant streaming computation at scale. Dans *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, 423–438. ACM.