Rapport Projet PSTL

Parallélisation automatique pour Java

Corentin Teillet - Alexis Capitanio

Lien GitHub : https://github.com/yanntm/AutoParallelJava

24/05/2021

Sorbonne Université - Faculté des Sciences et Ingénierie - M1 STL

Professeurs encadrants : Yann Thierry-Mieg, Cédric Besse et Jonathan Lejeune

Table des matières

Introduction

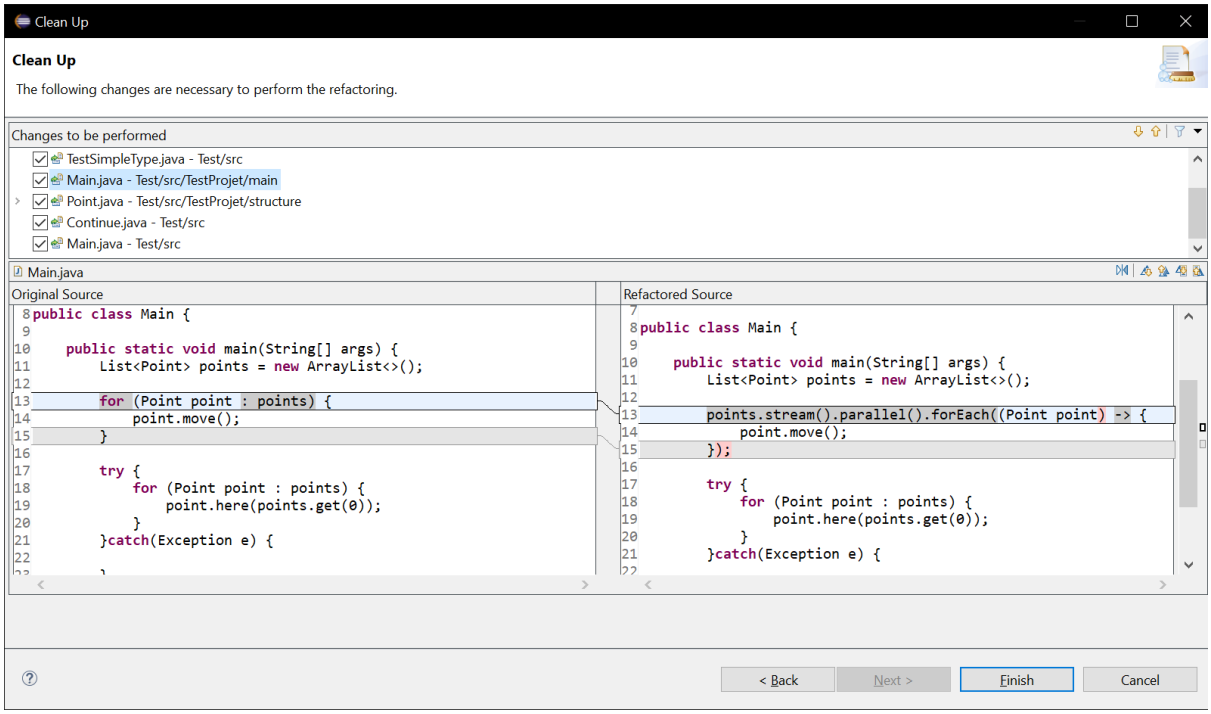
L'objectif de ce projet est de paralléliser des boucles faites sur des collections. Avec la sortie de Java 8 en 2014, une nouvelle interface permet de parcourir différemment les collections, notamment avec stream et parallel stream. La difficulté provient du fait qu'il faut d'abord transformer le corps de la boucle en une lambda expression. Cette Lambda Expression s'apparente à un style de programmation fonctionnelle où l'on réalise une composition de fonction pour obtenir le résultat voulu. L'utilisation de parallel stream permet de gagner du temps en utilisant des mécanismes de concurrence de façon transparente pour l'utilisateur. Les ordinateurs modernes sont équipés de processeurs multi-cores, pouvant donc faire plusieurs tâches en simultanées. Les mécanismes de parallélisation des boucles exploitant pleinement ce potentiel, permet donc de gagner du temps de calcul.

Le problème est donc de transformer des parcours de boucles utilisant l'ancienne méthode, avant Java 8, en un parcours concurrent qui sera donc plus rapide. Pour ce faire, nous allons développer un plug-in pour Eclipse, qui se chargera d'analyser le code source en parcourant l'arbre syntaxique abstrait du fichier qui est généré par Eclipse. Si une transformation est possible, alors le plug-in proposera une alternative utilisant un parcours parallèle.

Ce plug-in est utile pour tout développeur Java, car en une action la transformation est effectuée. De plus, il permet de remettre facilement au goût du jour d'anciens programmes pour les rendre plus efficaces sans devoir modifier manuellement le code.

Fonctionnement

Le principe de notre solution est de réécrire du code écris en Java vers du code Java en modifiant les parties que l'on a détectés comme transformable. Notre objectif est donc de trouver les boucles for améliorées et d'analyser leur corps de façon à ensuite les transformer en streams. Avant de valider la transformation, l'utilisateur peut voir les effets de la transformation pour ensuite décider de l'appliquer et peut aussi sélectionner seulement certaines transformations.



Contribution au problème

Pour pouvoir effectuer la transformation vers un stream parallèle, nous allons passer par deux étapes :

- la première étant la transformation de la boucle for améliorée vers un stream
- la deuxième la transformation du stream en un stream parallèle.

Pour pouvoir transformer une boucle for améliorée en stream, plusieurs conditions doivent être respectées :

(**P1**) Le corps de la boucle for ne doit lancer aucune exception vérifiée
(**P2**) Le corps de la boucle ne doit pas contenir de variable définie en dehors de la portée de la boucle qui sont non effectivement finale.
(**P3**) Le corps de la boucle ne doit pas contenir d'instruction "break".
(**P4**) Le corps de la boucle ne doit pas contenir d'instruction "return".
(**P5**) Le corps de la boucle ne doit pas contenir d'instruction "continue".

On a donc les mêmes conditions que l'article Lambdaficator([5]) sauf la précondition 1 de l'article. Nous n'appliquons pas la précondition 1 puisque nous pouvons récupérer un stream à partir d'un tableau Java grâce à la méthode Arrays.stream(). Nous pouvons donc transformer en stream un objet qui implémente l'interface Collections mais aussi un tableau d'objet, si toutes les autres conditions sont remplies.

Ensuite pour pouvoir passer d'un simple stream à un stream parallèle, il faut remplir les conditions nécessaires pour éviter les problèmes liés à la concurrence :

(**P6**) Modifier uniquement des valeurs de l'objet courant.
(**P7**) Si on modifie d'autres valeurs, il faut que ces valeurs soient protégées par des mécanismes de synchronisation de type synchronized ou Lock.
(**P8**) Il faut que l'objet soit présent seulement une fois dans la collection

Pour les conditions de nécessaire à la parallélisation on retrouve beaucoup des conditions énoncées dans [12] R. Khatchadourian, Y. Tang, M. Bagherzadeh, et S. Ahmed, « Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams » mais on ne se préoccupe pas de savoir si les structure sont ordonnées ou non.

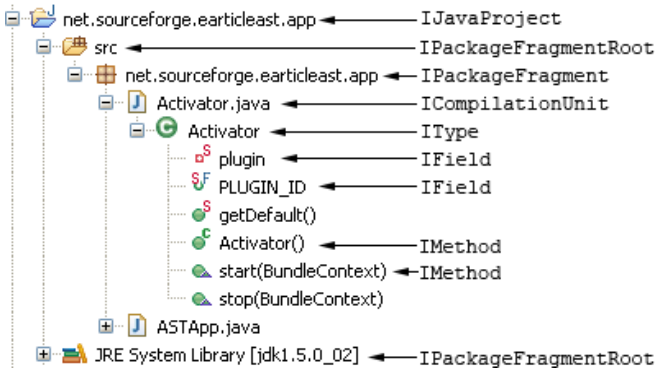
Les conditions 1-7 peuvent être vérifiées de façon statique, c'est-à-dire juste en vérifiant le code Java. Mais contrairement aux autres conditions, la condition 8 ne peut pas être vérifiés statiquement, il faudrait vérifier cela de façon dynamique donc à l'exécution. Malheureusement avec notre solution nous ne pouvons seulement faire des vérifications statiques car pour effectuer des vérifications dynamiques, il faudrait faire un outil qui tourne pendant l'exécution du programme.

Contribution fonctionnement

Pour atteindre notre objectif, nous avons créé un plug-in pour Eclipse, et nous allons nous greffer sur le point d'extension Clean-up d'Eclipse. Nous allons utiliser ce point d'extension car cela permet de montrer l'avant après à l'utilisateur. Nous aurions aussi pu nous greffer sur le point d'extension Refactor qui permet aussi de montrer les différences mais qui est moins accessible pour l'utilisateur puisqu'il propose plus d'interaction avec l'utilisateur, l'idéal pour nous est donc le clean-up puisque nous voulons quelque chose de très facile d'utilisation ainsi que de pouvoir visualiser les différences.

Pour effectuer des modifications sur les fichiers Java, nous avons utilisé la représentation qu'Eclipse fait d'un fichier Java, Eclipse représente chaque fichier par un arbre syntaxique abstrait (AST), avec chaque élément de l'arbre étant un ASTNode, par exemple une déclaration de méthode est représentée par objet de type MethodDeclaration et un appel de méthode est représenté par un objet de type MethodInvocation.

On peut voir un exemple de hiérarchie créée par Eclipse pour un projet sur cette image :



Eclipse donne comme paramètre au plugin pour effectuer la transformation, en argument le projet ainsi qu'une liste de CompilationUnit que l'utilisateur aura sélectionner au préalable pour effectuer les transformations.

Pour parcourir cet arbre, nous allons utiliser le design pattern visiteur qui permet de visiter chaque élément de l'arbre. Ce visiteur va commencer par visiter l'élément le plus haut et va aller le plus profond possible. Pour faire cela, on va instancier l'interface ASTVisitor qui va nous permettre de parcourir chaque élément de l'arbre, cette interface comporte deux méthodes pour chaque type d'éléments de l'arbre : la méthode endVisit et la méthode visit. La différence entre ces deux méthodes est que la première est appelée quand les fils de l'élément ont déjà été visité contrairement à la seconde qui est appelé avant la visite des fils. Grace à cette interface, nous allons pouvoir analyser chaque élément de l'arbre en vérifiant si un élément est présent ou s'il n'est jamais rencontré par exemple.

Nous avons créé 5 visiteurs :

- Le premier est la classe **MethodVisitor** qui nous sert à visiter les méthodes pour les classer en fonction de si elles sont parallélisables ou non, c'est elle qui vérifiera les conditions P6-8 pour les corps des méthodes.
- Le deuxième est la classe **ASTVisitorPreCond** qui sert à visiter l'objet sur lequel on veut effectuer des modifications, pour pouvoir vérificer si l'on peut transformer cet objet en stream. Il vérifie donc les conditions P1-P5 pour savoir si l'on peut transformer en stream le for amélioré, si on se rends compte qu'il n'est pas possible d'effectuer la transformation vers un stream alors on ne fera aucun traitement dessus.
- Le troisième est la classe **TraitementForBodyVisitor** qui va visiter tous les if du projets et génerer l'appel à des méthodes filter si les if sont des if englobants et qu'ils n'ont pas de else
- Le quatrième est la classe **TransformationMapVisitor qui** permet de vérifier si le corps du for amélioré peut–être transformé en une réduction ainsi que d'ajouter l'appel à la méthode mapTo s'il y en a besoin. Nous pouvons effectuer deux réductions : la réduction vers un appel à la méthode sum, si l'objectif de la boucle for est d'effectuer la somme de

valeur ; et la réduction vers la méthode collect(toList()) si l'objectif de la boucle est de remplir une liste.

Réécriture

Transformation vers méthode forEach

Le cas le plus général de réécriture est quand on prend le corps de la boucle forEach et qu'on le copie dans le stream.forEach()

```
1.  List<Point> points = …… ;
2.  for(Point p : points){
3.      p.move(10,10) ;
4.  }
```

Si on transforme cet exemple en stream, cela nous donne :

```
1.  List<Point> points = …… ;
2.  points
3.      .stream()
4.      .forEach(p -> p.move(10,10) ;
```

C'est donc le cas le plus général car aucun changement n'a pu être appliqué sur le corps de la boucle il a juste été intégré dans le lambda du forEach.

Transformation avec filter

Un traitement possible est d'ajouter un appel à la méthode filter entre l'appel à la méthode stream et l'appel à la méthode forEach. La méthode filter sera ajouté quand le corps de la boucle contient un if englobant sur tout le corps sans un else :

```
1.  List<Point> points = …… ;
2.  for (Point p : points) {
3.      if (p.x > 10) {
4.          p.move(10,10) ;
5.      }
6.  }
```

En appliquant la transformation, on obtiendra :

```
1.  List<Point> points = … ;
2.  points
3.      .stream()
4.      .filter(p -> p.x > 10)
5.      .forEach(p -> p.move(10,10) ;
```

Cette réécriture va donc contrairement à la précédente modifier le corps original puisqu'elle va supprimer le if pour ensuite mettre le corps du if dans la lambda du forEach.

De la même façon, si plusieurs if sont imbriqués sans avoir de else on peut enchainer les appels à la méthode filter :

```
1.  List<Point> points = …… ;
2.  for (Point p : points) {
3.      if (p.x > 10) {
4.          if (p.y< 0) {
5.              p.move(10,10) ;
```

```
6.          }
7.      }
8.  }
```

La récriture donnera :

```
1.  List<Point> points = … ;
2.  points
3.      .stream()
4.      .filter(p -> p.x > 10)
5.      .filter(p -> p.y < 0)
6.      .forEach(p -> p.move(10,10) ;
```

Transformation et réduction avec la méthode mapTo

MapTo est une méthode permettant de transformer un stream général vers un stream particulier comme un IntStream pour les int ou un DoubleStream pour le type double. Il existe un mapTo pour chaque type simple de Java : mapToInt, mapToDouble, mapToChar ….

Cet opérateur va nous permettre d'effectuer une réduction avec par exemple sum qui permet de faire la somme de tous les éléments présents dans le stream.

```
1.  somme = 0 ;
2.  List<Point> points = … ;
3.  for (Point p : points) {
4.      somme+=p.x;
5.  }
```

Ceci pourra se réécrire de cette façon :

```
1.  List<Point> points = … ;
2.  somme += points .stream()
3.                  .mapToInt(point -> point.x)
4.                  .sum() ;
```

Transformation vers addAll

Quand l'objectif de la boucle forEach est de remplir une liste, on peut utiliser la méthode de réduction collect qui permet de remplir la liste que l'on veut. Il nous faut donc pouvoir détecter le remplissage de cette liste.

```
1.  List<Integer> res = … ;
2.  List<Point> points = … ;
3.  for (Point p : points) {
4.      res.add(p.x);
5.  }
```

La transformation donne :

```
1.  List<Integer> res = …;
2.  List<Point> points = …;
3.  res.addAll( points.stream().map( point -> p.x).collect(Collectors.toList()) );
```

Transformation des boucles avec un paramètre de type simple

Le problème quand les paramètres de la boucle for améliorées sont de type simple est qu'il faut au préalable passer par un appel à la méthode mapTo pour transformer le stream d'un type Object vers un type simple, on est obligé de le faire car sinon cela peut casser des bouts de codes. Par exemple avec l'appel à une méthode remove sur une Collection si le paramètre est de type Integer, la méthode cherchera à supprimer l'élément donné en paramètre, alors que si l'on donne un objet de type int la méthode cherchera à supprimer l'élément d'index donné.

Si l'on a comme code de départ :

```
1.  List<Points> points = … ;
2.  List<Integer> vals = … ;
3.  for (int i : vals) {
4.      points.remove(i);
5.  }
```

La transformation donnera :

```
1.  vals.stream().mapToInt( i -> i).forEach( i -> points.remove(i));
```

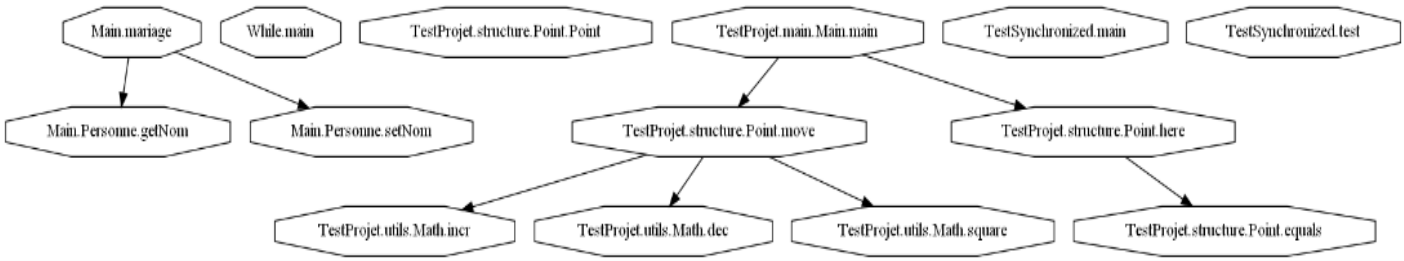
Parallélisation

Pour pouvoir paralléliser une boucle, il faut que la boucle ait déjà été transformer en stream. Ensuite il faut analyser le code du corps de la boucle pour pouvoir vérifier les conditions nécessaires. La plus grosse difficulté que l'on va rencontrer est celle de pouvoir savoir si les méthodes appelées ont un comportement compatible avec les conditions de parallélisation.

Analyse des méthodes

Pour traiter le problème des dépendances des méthodes on construit un graphe d'appel de méthode sur projet entier. Pour cela on parcourt la totalité de du projet en insérant au fur et à mesure les méthodes rencontrées et en créant un lien quand une méthode en appel une autre. On obtient ainsi un graph dirigé où chaque sommet représente une méthode et chaque lien de s1 vers s2 est un appel de fonction de la méthode s2 au sein de la méthode s1. Nous utilisons un des outils *Puck Graph*, implémenter par Yann Thierry-Mieg, Cédric Besse et Mikal Ziane (https://github.com/yanntm/Puck3), afin de créer ce graph.

Nous pouvons voir le graphe généré sur un petit projet exmple.



On s'appuiera sur un parcours DFS (Depth First Search) du graphe pour retrouver les cycles. La méthode suivante remplira la liste *res* avec tous les cycles impliquant le sommet d'origine (sommet *src*).

```
1.  public void dfs(List<List<Integer>> res, int src, int current, List<Integer> stack,
    boolean[] visited) {
2.      stack.add(current);
3.      visited[current] = true;
4.      for(int i : graph[current]) {
5.          if (current == src) {
6.              res.add(new ArrayList<>(stack));
7.          } else if (!visited[i]) {
8.              dfs(res, src, i, stack, visited);
9.          }
10.     }
11.     stack.remove(stack.size()-1);
12. }
```

On appel cette méthode sur tous les sommets du graph afin de récupérer tous les cycles du graphe.

Une fois le graphe construit et les cycles stockés, il temps de commencer le traitement du graphe. Cela consiste en une analyse du code des méthodes. Pour chaque méthode on attribuera une catégorie entre *NotParallelizable*, *ModifLocal*, *ReadOnly* et *ThreadSafe* en fonction de ce comportement et du comportement des méthodes appelées et on la rangera dans la liste correspondante.

- ***NotParallelizable:*** Les fonctions de cette catégorie ne sont pas parallélisables. Elles ne respectent pas l'une des conditions à la parallélisation. Une méthode appelant une autre méthode de la catégorie *NotParallelizable* se verra directement attribué cette dernière.
- ***ReadOnly :*** Ces méthodes ne réalise aucune opération d'écriture et sont parallélisables.
- ***ModifLocal :*** Ces méthodes réalise des opérations d'écriture local et sont parallélisables. Une méthode en appelant une autre de cette catégorie ne pourra plus être *ReadOnly et sera au minimum ModifLocal.*
- ***ThreadSafe :*** Les méthodes de cette catégorie peuvent avoir des comportements ne respectant pas les conditions de parallélisation. Mais, la présence de mécanisme de synchronisation pour protéger des accès concurrents indique que l'utilisateur est conscient de cela et essaye de s'en protéger. Le plug-in ne garantit pas un comportement concurrent viable et considère que l'utilisateur s'en est lui-même assurer. Une méthode en appelant une autre de cette catégorie ne pourra alors n'être que *ThreadSafe* ou *NotParallelizable.*

Notons qu'il se dégagent un certain ordre entre les catégories, où certaines en écrase d'autres. On obtient alors l'ordre suivant :

ReadOnly** écrasée par **ModifLocal** écrasée par **ThreadSafe** écrasée par **NotParallelizable

Pour commencer on extrait les sommets des cycles pour les traiter en premier. En effet, on ne peut pas les traiter de manière récursive sur les fils car cela engendrerait des appels récursif infinis. On traitera donc tous le cycle d'un seul coup, attribuant à toutes les méthodes du cycle la catégorie la plus "dominante". Une fois la catégorie attribuée tous les sommets du cycle seront réunie en un même sommet dans le graphe pour d'obtenir un arbre.

Nous pouvons à présent commencer le traitement principal des méthodes de l'arbre. Pour cela on va trier les sommets en fonction de la distance avec la feuille la plus éloignée dans leurs sous arbre. À tour de rôle et en commençant par les feuilles on va analyser la méthode associer à chaque sommet et lui attribuer une catégorie en fonction de son comportement de la catégorie de ses fils.

Pour cette seconde étape une alternative aurait été de simplement réaliser le traitement de manière récursive en appelant le traitement sur tous les fils puis en traitant le nœud courant. On aurait donc un traitement ressemblant à cela :

```
1.  algo parcours(sommet):
2.      if sommet alredyParsed
3.          return
4.      forAll s in sommet.subTree :
5.          parcours (s)
6.      parse(sommet)
7.
```

Transformations

Une fois l'attribution des méthodes réalisée, on analyse le corps de la boucle à transformer. Comme vus précédemment si les conditions nécessaires sont remplies, on transforme la boucle en stream avec des ajouts éventuels de méthode *mapTo, filter, sum* et *addAll.*

Enfin, on essaye de paralléliser le stream en verifant que les conditions de parallélisation sont vérifiées. Si le stream est suivie d'une opération *sum* ou *addAll,* alors les conditions sont forcément vérifiées et il sera possible de paralléliser directement ce dernier. Sinon, on analysera encore une fois le corps de lambda du forEach. Lors de cette analyse, on vérifiera que les catégories des méthodes appelées soient *ReadOnly*, *ModifLocal* ou *ThreadSafe* ainsi que les conditions de parallélisation soit bien respecté. Si tel est le cas on ajoutera ".parallel" après le stream pour le rendre parallel.

```
for(Personne p:pers) {
    if(p.age==0 && p.sexe) {
        p.anniversaire();
    }
}
```

```
51      pers.stream().parallel().filter((Personne p) -> p.age == 0 && p.sexe).forEach((Personne p) -> {
52          p.anniversaire();
53      });
54
55
```

Limites

Le problème majeur de cette méthode est le besoin d'avoir un projet complet avec toutes les fichiers source accessible à l'analyse. Il sera impossible pour le plug-in d'analyser une méthode compris dans un jar ou faisant partie de la bibliothèque standard java. En l'état beaucoup de méthodes se verront attribuer la catégorie *NotParallelizable* par manque d'information alors qu'elles auraient pu être exécuter dans un contexte parallèle.

Intérêt de la transformation en parallèle

Comme souligné en introduction la parallélisation de code permet d'exploité pleinement le potentiel des ordinateur moderne et permet généralement de gagner du temps par rapport à la programmations séquentiels. Pour autant cela ne signifie pas que le passage au parallèle n'a pas un coût. En effet le déploiement des mécanismes nécessaire à la parallélisation comme les Thread et pool de Thread, nécessite un temps création. Il sera donc plus intéressant de l'utiliser sur des grosses structures et/ou avec des opérations demandant un certain temps de calcule. Notons que selon l'article de J. Jurinova [9] sur la comparaison des performances en utilisant certaines features de Java 8, on peut voir que l'utilisation de stream ou de parallel stream, bien que relativement proche en termes de temps, reste favorable la plupart du temps.

Mesures

Pour évaluer notre travail, on peut parler de quelques chiffres :

- Le nombre de lignes total du projet est de 7423
- Le nombre de commit est de 85
- Le projet contient 80 fichiers dont 20 exclusivement pour la gestion du graphe de dépendances des méthodes

Nous avons testé notre plug-in sur un projet trouvé sur Github, ce projet est : https://github.com/coobird/thumbnailator

Nous avons choisi ce projet car c'est un projet écrit en Java sous eclipse, que ce projet est utilisé par beaucoup de personnes (17.2k) et que c'est un projet auquel on peut tous ce servir. Thumbnailator permet de réduire beaucoup d'image d'un coup. C'est donc un projet intéressant et qui peut servir à tout le monde voulant par exemple réduire des images pour envoyer sur les réseaux sociaux.

Pour effectuer les tests, nous avons effectué les tests avec un lot de 45 images et ensuite un lot de 130 images, nous avons d'abord fait les tests avec le programme initial puis nous avons appliqué notre plug-in.

Voici un bref récapitulatif des résultats obtenus :

nombre d'images	45	130
Avant application	3762 (ms)	36881 (ms)
Apres application	3030 (ms)	36378 (ms)

On peut donc voir que sur le plus petit lot, nous gagnons presque 1s ce qui représente 20% de gains de temps, alors que sur le plus gros lot, l'écart n'est pas signifiant.

On peut expliquer cet écart par le fait que nous n'appliquons pas le parallèle stream sur assez de boucles puisqu'il faudrait avoir les sources de toutes les méthodes utilisées, nous pouvons aussi perdre du temps sur la transformation en stream sur de faibles boucles.

Conclusion

Notre projet se compose de plusieurs visiteurs qui vont analyser à tour de rôle le code pour vérifier et/ou extraire certains éléments. Pour commencer, on analyse l'entièreté du projet en assignant une catégorie à chaque méthode. Dans un second temps, on cherche à retrouver les boucles for éligibles à la transformation en stream. La troisième étape est la recherche de if englobant à remplacer par des filter. Puis, on détecte les patterns compatibles avec mapTo, addAll et sum. Enfin, on analyse une dernière fois le code pour vérifier les conditions de parallélisation.

Dans l'état actuel de notre plug-in seules les boucles relativement simples ne faisant pas appel à des méthodes d'une bibliothèque peuvent être traiter ce qui réduit grandement le nombre des boucles capables d'être réécrite. De plus, bien que généralement plus rapide la différence de temps entre une exécution avant et après transformation varie grandement en fonction du code d'origine et peut parfois être négligeable.

Comme nous l'avons vu notre projet souffre d'un gros défaut, il doit avoir accès au code source de toutes les méthodes pour pouvoir les catégoriser et bien fonctionner. L'une de solution serait de s'intéressé byte code mais nous voulions rester sur de l'analyse syntaxique. Une autre solution, serait de faire d'avoir fait une analyse complète de la bibliothèque standard au préalable et de stocké le résultat. Ainsi, appelé une méthode dans la bibliothèque standard ne poserait plus de problème et permettrait de pouvoir traiter un nombre beaucoup plus important de boucle. Pour finir, la dernière solution que nous avons trouvée et qui peut être couplée à la précédente est d'utiliser un décompileur. Cela transformerait les fichiers class en fichier source que l'on pourrait alors analyser.

Bibliographie

[1] R. Khatchadourian, Y. Tang, M. Bagherzadeh, et S. Ahmed, « A Tool for Optimizing Java 8 Stream Software via Automated Refactoring », in *2018 Ieee 18th International Working Conference on Source Code Analysis and Manipulation (scam)*, New York: Ieee, 2018, p. 34-39.

[2] Eclipse, « Help - Eclipse Platform », *Help - Eclipse Platform*, 2007. https://help.eclipse.org/2020-12/index.jsp.

[3] Oracle, « java.util.stream (Java Platform SE 8) », *Javadoc Java 8*, janv. 07, 2021. https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html.

[4] L. Franklin, A. Gyori, J. Lahoda, et D. Dig, *LAMBDAFICATOR: From Imperative to Functional Programming through Automated Refactoring*. New York: Ieee, 2013, p. 1287-1290.

[5] L. Franklin, A. Gyori, J. Lahoda, et D. Dig, « LambdaFicator: From imperative to functional programming through automated refactoring », in *2013 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, USA, mai 2013, p. 1287-1290, doi: 10.1109/ICSE.2013.6606699.

[6] Oracle, « Lesson: Aggregate Operations (The Java™ Tutorials > Collections) », *Tutorials for Java 8*, 2020. https://docs.oracle.com/javase/tutorial/collections/streams/index.html.

[7] Eclipse, « Package org.eclipse.jdt.core.dom », *Documentation of the DOM eclipse*, 2017. https://www.ibm.com/support/knowledgecenter/SSZHNR_2.0.0/org.eclipse.jdt.doc.isv-3.14.100/reference/api/org/eclipse/jdt/core/dom/package-summary.html.

[8] Oracle, « Parallelism (The Java™ Tutorials > Collections > Aggregate Operations) », *Lessons for Java 8*, 2020. https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html.

[9] J. Jurinova, « Performance improvement of using lambda expressions with new features of Java 8 vs. other possible variants of iterating over ArrayList in Java », *J. Appl. Math. Stat. Inform.*, vol. 14, n° 1, p. 103-131, mai 2018, doi: 10.2478/jamsi-2018-0007.

[10] B. P. Lester, « Performance of Map-Reduce Using Java-8 Parallel Streams », in *Intelligent Computing, Vol 1*, vol. 858, K. Arai, S. Kapoor, et R. Bhatia, Éd. Cham: Springer International Publishing Ag, 2019, p. 723-736.

[11] Y. Tang, R. Khatchadourian, M. Bagherzadeh, et S. Ahmed, « Poster: Towards Safe Refactoring for Intelligent Parallelization of Java 8 Streams », in *Proceedings 2018 Ieee/Acm 40th International Conference on Software Engineering - Companion (icse-Companion*, New York: Ieee, 2018, p. 206-207.

[12] R. Khatchadourian, Y. Tang, M. Bagherzadeh, et S. Ahmed, « Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams », in *2019 Ieee/Acm 41st International Conference on Software Engineering (icse 2019)*, New York: Ieee, 2019, p. 619-630.