Guide Complet

Parallel Streams en Java

Support de cours – 4ème année IIR – EMSI

10 novembre 2025

Table des matières

1 Définitions Fondamentales

1.1 Stream (Flux)

Définition : Stream

Une séquence d'éléments provenant d'une source de données (collection, tableau, etc.) qui supporte des opérations de traitement de données de manière déclarative.

Caractéristiques principales :

— **Immutable** : Ne modifie pas la source de données

— **Lazy (Paresseux)** : Les opérations intermédiaires ne s'exécutent que quand une opération terminale est appelée

— **Consommable** : Un stream ne peut être parcouru qu'une seule fois

1.2 Parallel Stream (Flux Parallèle)

Définition : Parallel Stream

Un stream qui divise automatiquement ses données en plusieurs segments et les traite simultanément sur plusieurs threads (cœurs processeur).

Comment ça marche :

```
Collection de 8 éléments : [1, 2, 3, 4, 5, 6, 7, 8]

Stream séquentiel :         Thread principal fait tout
Thread 1: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8

Parallel Stream :           Réparti sur 4 threads
Thread 1: 1 → 2
Thread 2: 3 → 4
Thread 3: 5 → 6
Thread 4: 7 → 8
```

1.3 ForkJoinPool

Définition : ForkJoinPool

Le moteur sous-jacent qui gère les threads pour les parallel streams. Par défaut, il utilise autant de threads que de cœurs disponibles sur le processeur.

1.4 Thread-Safety (Sécurité des Threads)

Définition : Thread-Safety

La capacité d'un code à fonctionner correctement quand plusieurs threads l'exécutent simultanément, sans corruption de données.

1.5 Race Condition (Condition de Course)

Définition : Race Condition

Problème qui survient quand plusieurs threads accèdent et modifient simultanément une ressource partagée, causant des résultats imprévisibles.

2 Exemple 1 : Bases du Parallel Stream

2.1 Code de Base

```
List<Integer> nombres = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

// Stream s quentiel classique
nombres.stream()
.map(n -> {
  System.out.println("Thread: " +
  Thread.currentThread().getName() + " - Traite: " + n);
  return n * 2;
})
.forEach(n -> {});

```

Explication ligne par ligne :

— `Arrays.asList(...)` : Crée une liste immuable de 10 entiers

— `.stream()` : Crée un **stream séquentiel** – un seul thread traite tous les éléments dans l'ordre

— `.map(n -> {...})` : Opération intermédiaire qui transforme chaque élément

— `Thread.currentThread().getName()` : Affiche le nom du thread qui traite l'élément

— Résultat : Vous verrez toujours le même thread (`main`)

2.2 Version Parallèle

```
nombres.parallelStream()
.map(n -> {
  System.out.println("Thread: " +
  Thread.currentThread().getName() + " - Traite: " + n);
  return n * 2;
})
.forEach(n -> {});

```

Différence clé :

— `.parallelStream()` : Crée un **parallel stream**

— Divise le travail entre plusieurs threads

— Résultat : Vous verrez différents threads (`ForkJoinPool.commonPool-worker-1`, `worker-2`, etc.)

Sortie Attendue

Stream séquentiel :

```
Thread: main - Traite: 1
Thread: main - Traite: 2
...
```

Stream parallèle :

```
Thread: ForkJoinPool.commonPool-worker-1 - Traite: 3
Thread: main - Traite: 1
Thread: ForkJoinPool.commonPool-worker-2 - Traite: 7
...
```

3 Exemple 2 : Comparaison de Performance

3.1 Création d'une Grande Collection

```
List<Integer> grandeCollection = IntStream.rangeClosed(1, 10
_000_000)
.boxed()
.collect(Collectors.toList());

```

Décomposition :

— `IntStream.rangeClosed(1, 10_000_000)` : Crée un flux de 1 à 10 millions

— `.boxed()` : Convertit les `int` primitifs en objets `Integer`

— `.collect(Collectors.toList())` : Collecte tous les éléments dans une List

Pourquoi 10 millions ?

Les petites collections ne bénéficient pas du parallélisme à cause de l'overhead (coût) de gestion des threads.

3.2 Test avec Stream Séquentiel

```
long debut1 = System.currentTimeMillis();
long somme1 = grandeCollection.stream()
.filter(n -> n % 2 == 0)
.mapToLong(n -> n * n)
.sum();
long fin1 = System.currentTimeMillis();

```

Pipeline expliqué :

1. `.stream()` : Stream séquentiel
2. `.filter(n -> n % 2 == 0)` : Garde uniquement les nombres pairs
   — `n % 2 == 0` : Le reste de la division par 2 est 0
3. `.mapToLong(n -> n * n)` : Transforme chaque nombre en son carré
   — `mapToLong` : Plus efficace que `map` pour les calculs numériques
4. `.sum()` : Opération terminale qui additionne tous les carrés

3.3 Test avec Parallel Stream

```
long debut2 = System.currentTimeMillis();
long somme2 = grandeCollection.parallelStream()
.filter(n -> n % 2 == 0)
.mapToLong(n -> n * n)
.sum();
long fin2 = System.currentTimeMillis();

```

Performance Typique

— **Séquentiel** : 400–600 ms
— **Parallèle** : 100–150 ms
— **Gain** : 3–4x plus rapide

4 Exemple 3 : Les Dangers du Parallel Stream

4.1 Danger 1 : Race Condition

Code Dangereux

```
1       List<Integer> resultats = new ArrayList<>();  // NON
    thread-safe!
2
3       nombres.parallelStream()
4       .map(n -> n * 2)
5       .forEach(resultats::add);  // DANGER!
6
```

Pourquoi c'est dangereux ?

```
Scénario de race condition :
Thread 1 lit la taille = 5
Thread 2 lit la taille = 5 (en même temps!)
Thread 1 écrit à index 5
Thread 2 écrit à index 5 (écrase Thread 1!)
→ Un élément est perdu!
```

ArrayList n'est pas synchronisée. Plusieurs threads qui ajoutent en même temps peuvent :

— Perdre des éléments

— Corrompre la structure interne

— Lever une `ConcurrentModificationException`

Solution Correcte

```
1       List<Integer> resultats = nombres.parallelStream()
2       .map(n -> n * 2)
3       .collect(Collectors.toList());  // Thread-safe!
4
```

4.2 Danger 2 : Ordre Non Garanti

```
1     nombres.parallelStream()
2     .limit(10)
3     .forEach(n -> System.out.print(n + " "));
4
```

Sortie possible : `5 1 8 3 2 9 4 7 6 10`

Pourquoi ? Chaque thread termine à son propre rythme. Le premier à finir affiche en premier.

Solution si l'ordre compte

```
1       .forEachOrdered(n -> System.out.print(n + " "));   //
    Respecte l'ordre
2
```

5 Exemple 4 : Bonnes Pratiques

5.1 Cas 1 : Traitement de Clients

```
List<String> clientsVIP = clients.parallelStream()
.filter(c -> c.getSolde() > 50000)
.filter(c -> c.getAnneeInscription() < 2020)
.map(c -> c.getNom().toUpperCase())
.sorted()
.collect(Collectors.toList());

```

Pipeline expliqué :

1. `.parallelStream()` : Divise la liste de clients entre threads
2. Premier `.filter()` : Garde clients avec solde > 50,000 MAD
3. Deuxième `.filter()` : Garde clients inscrits avant 2020
4. `.map()` : Convertit noms en majuscules
5. `.sorted()` : Trie alphabétiquement (attention : coûteux en parallèle !)
6. `.collect()` : Rassemble tous les résultats de manière thread-safe

Pourquoi c'est correct ?

— Pas de modification de variables externes
— `collect()` gère la synchronisation
— Chaque opération est indépendante

5.2 Cas 2 : Calculs Statistiques

```
double soldeMoyen = clients.parallelStream()
.filter(Client::isActif)
.mapToDouble(Client::getSolde)
.average()
.orElse(0.0);

```

Décomposition :

— `Client::isActif` : Référence de méthode (équivalent à `c -> c.isActif()`)
— `mapToDouble(Client::getSolde)` : Extrait les soldes comme primitives `double`
— `.average()` : Retourne un `OptionalDouble` (peut être vide si aucun client actif)
— `.orElse(0.0)` : Si pas de résultat, retourne 0.0

5.3 Cas 3 : Groupement Concurrent

```
Map<String, Long> parCategorie = clients.parallelStream()
.collect(Collectors.groupingByConcurrent(
Client::getCategorie,
Collectors.counting()
));

```

Explication :

— `groupingByConcurrent()` : Version thread-safe de `groupingBy()`

— `Client::getCategorie` : Fonction de classification (clé du Map)

— `Collectors.counting()` : Compte les éléments par catégorie

— Résultat : `{Standard=6543, Premium=2341, VIP=1116}`

Différence avec groupingBy() :

— `groupingBy()` : Crée un HashMap normal (problèmes en parallèle)

— `groupingByConcurrent()` : Crée un ConcurrentHashMap (thread-safe)

6 Quand Utiliser Parallel Stream ?

6.1 Situations Idéales

Utiliser Parallel Stream

```
        // 1. Grande collection + op ration co teuse
        List<Image> images = chargerImages(100_000);
        images.parallelStream()
        .map(img -> redimensionner(img))  // Calcul intensif
        .collect(Collectors.toList());

        // 2. Calculs ind pendants
        List<Double> prix = produits.parallelStream()
        .mapToDouble(p -> p.getPrix() * 1.20)  // Chaque calcul
    ind pendant
        .boxed()
        .collect(Collectors.toList());

        // 3. Filtrage sur grande  chelle
        logs.parallelStream()
        .filter(log -> log.niveau == ERROR)
        .filter(log -> log.timestamp.isAfter(hier))
        .collect(Collectors.toList());

```

6.2 Situations à Éviter

Éviter Parallel Stream

```
        // 1. Petite collection
        List<Integer> petiteListe = Arrays.asList(1, 2, 3, 4, 5)
    ;
        petiteListe.parallelStream()  // Overhead > gain
        .map(n -> n * 2)
        .collect(Collectors.toList());

        // 2. Modification de variables externes
        int compteur = 0;  // Race condition garantie!
        nombres.parallelStream()
        .forEach(n -> compteur++);  // ERREUR!

        // 3. Ordre important
        clients.parallelStream()
        .sorted()  // Tr s co teux en parall le
        .limit(10)  // Doit attendre le tri complet
        .collect(Collectors.toList());

```

7 Métriques de Performance

7.1 Facteurs qui Influencent le Gain

7.1.1 Taille de la collection

— $<$ 1,000 éléments : Pas de gain
— 1,000 – 10,000 : Gain possible
— $>$ 10,000 : Gain significatif

7.1.2 Coût de l'opération

— Simple (`n * 2`) : Peu de gain
— Moyenne (parsing, regex) : Gain moyen
— Complexe (I/O, calculs lourds) : Gain important

7.1.3 Nombre de cœurs CPU

— 2 cœurs : Gain ∼1.5x
— 4 cœurs : Gain ∼2–3x
— 8 cœurs : Gain ∼4–6x

7.2 Formule Simple

Estimation du Gain Potentiel

$$Gain potentiel = \min\left(\frac{nombre\_elements}{1000}, nombre\_coeurs \times 0.75\right)$$

8 Conseils Pratiques

8.1 Toujours Tester

```
    // Cr er une m thode de benchmark
    public static <T> long benchmark(Supplier<T> operation) {
      long debut = System.currentTimeMillis();
      operation.get();
      return System.currentTimeMillis() - debut;
    }

    // Utiliser
    long tempsSeq = benchmark(() ->
    data.stream().filter(...).collect(...)
    );
    long tempsPar = benchmark(() ->
    data.parallelStream().filter(...).collect(...)
    );

```

8.2 Éviter les Effets de Bord

```
    // Mauvais
    List<String> results = new ArrayList<>();
    data.parallelStream().forEach(results::add);

    // Bon
    List<String> results = data.parallelStream()
    .collect(Collectors.toList());

```

8.3 Utiliser les Collectors Appropriés

Pour parallel streams, préférez :

— `Collectors.groupingByConcurrent()` au lieu de `groupingBy()`

— `Collectors.toConcurrentMap()` au lieu de `toMap()`

9 Points Clés à Retenir

1. **Parallel Stream ≠ Toujours Plus Rapide**
   — Testez sur votre cas spécifique
2. **Pas de Variables Externes**
   — Utilisez toujours `collect()`
3. **Thread-Safety d'abord**
   — Collections concurrentes
   — Pas d'état partagé
4. **Mesurer, Mesurer, Mesurer**
   — Benchmarks réels
   — Profiling si nécessaire
5. **Simplicité avant Performance**
   — Code séquentiel d'abord
   — Optimiser si nécessaire

10 Conclusion

Les Parallel Streams sont un outil puissant de Java pour améliorer les performances sur de grandes collections avec des opérations coûteuses. Cependant, ils doivent être utilisés avec précaution :

— Toujours privilégier la **simplicité** et la **correction** avant la performance
— **Mesurer** l'impact réel avant d'optimiser
— Respecter les règles de **thread-safety**
— Éviter les **effets de bord**

Expérimentez, testez, et adaptez à vos besoins spécifiques !