Verrous, barrières et loquets

Master Informatique — Semestre 2 — UE obligatoire de 3 crédits

Année 2017-2018

Version du 28 janvier 2018

Le paquetage `java.util.concurrent`

L'API de **`java.util.concurrent`** propose des outils clefs en main pour **simplifier la gestion** du multitâche en JAVA et améliorer les **performances**.

Le programmeur n'a plus à **réinventer la roue** pour des fonctionnalités standards telles que les *exécutions asynchrones*, les gestions de *collections* accédées par plusieurs threads (telles que les files d'attentes), les *verrous* en lectures/écritures, etc.

Il doit ≪ juste ≫ utiliser les bons outils, correctement.

☛ *De nouveaux types de verrous*

Les limites de `synchronized`

L'usage de verrous intrinsèques s'effectue via le mot-clef **`synchronized`** dont l'un des atouts est de **forcer le relâchement du verrou dans chaque méthode où il est saisi.**

C'est aussi une limite parfois embarrassante.

- Il est impossible en effet de *prendre un verrou intrinsèque dans une méthode et de le relâcher dans une autre.*
- Il est impossible également de *tenter de prendre un verrou* intrinsèque : ou bien le thread obtient le verrou, ou bien il est bloqué.
- Il est impossible de tenter de prendre un verrou intrinsèque *pendant un laps de temps limité*, c'est-à-dire d'abandonner après un délai.
- Les verrous intrinsèques ne sont pas *équitables* (bien au contraire).
- Il est impossible de lâcher un verrou intrinsèque lors du traitement d'une *interruption* : il faut quitter le bloc **`synchronized`**.
- etc.

Le paquetage `java.util.concurrent.locks`

L'interface **`Lock`** offre de nouveaux objets « verrous » munis de méthodes équivalentes à l'acquisition et au relâchement du verrou intrinsèque d'un objet. Cependant l'acquisition et le relâchement ne sont plus liés à un bloc.

Deux implémentations de cette interface sont proposées :

① **`ReentrantLock`** : un seul thread possède le verrou ; on peut *tenter* de le prendre ou l'attendre un *temps limité*. Voire le relâcher lors d'une *interruption*.

② **`ReentrantReadWriteLock`** : plusieurs threads sont acceptés en « lecture », mais un seul en « écriture ».

Ce n'est pas véritablement un verrou !

Ce type de verrou permet de mettre en oeuvre des *lectures en parallèle* et des *écritures en exclusion mutuelle* afin d'augmenter les performances.

Interface Lock

Méthodes permettant d'acquérir ou de relâcher un verrou :

- **`lock()`** prend le verrou s'il est disponible et sinon endort le thread.
- **`unlock()`** relâche le verrou. L'appel à **`unlock()`** doit se faire quoi qu'il arrive, par exemple dans un bloc **`finally`**.

- **`lockInterruptibly()`** lève **`InterruptedException`** si le statut d'interruption est positionné lors de l'appel de cette méthode ou si le thread est interrompu *pendant qu'il attend le verrou*.
  - ⇝ Le traitement de cette exception pourra relâcher le verrou !

Blanche-Neige

```
class BlancheNeige {
  private ReentrantLock verrou = new ReentrantLock();
  public void requerir() {
    System.out.println(Thread.currentThread().getName()
                       + "␣veut␣la␣ressource");
  }
  public void acceder() {
    verrou.lock();
    System.out.println(Thread.currentThread().getName()
                       + "␣accède␣à␣la␣ressource.");
  }
  public void relacher() {
    System.out.println(Thread.currentThread().getName()
                       + "␣relâche␣la␣ressource.");
    verrou.unlock();
  }
}
```

La méthode `tryLock()`

Il est possible d'**essayer** d'acquérir un verrou et si le verrou est déjà pris par un autre thread,

① *de ne pas être mis en attente*

```
boolean tryLock()
```

② ou *d'être mis en attente mais seulement pendant un temps maximal déterminé*

```
boolean tryLock(long time, TimeUnit unit)
```

Cette méthode pourra lever une **`InterruptedException`**.

Verrou équitable (fair)

L'implémentation des verrous réentrants propose deux traitements différents de la liste des threads en attente : **équitable** (fair) ou non.

Il faut choisir lors de la construction du verrou :

```
ReentrantLock(boolean fairness)
```

Si l'équité est choisie, le thread qui attend depuis le plus longtemps est ≪ favorisé ≫ pour le réveil.

Mais ceci ne garantit pas un ordonnancement FIFO de manière absolue.

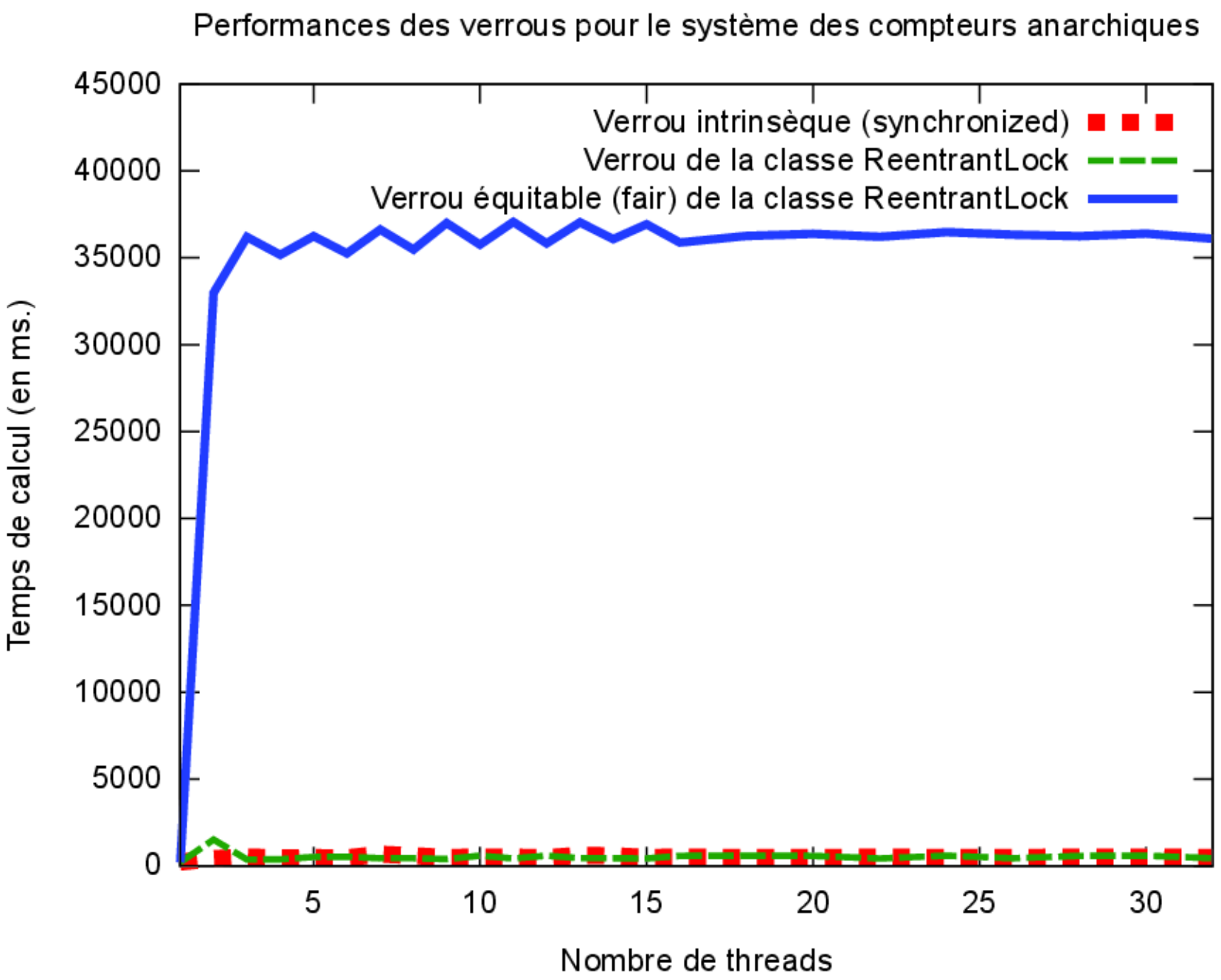
En effet, **`tryLock()`** ne respecte pas l'équité d'un verrou : si le verrou est disponible, la méthode retournera **`true`** même si d'autres threads sont déjà en attente !

Trois codages pour corriger le code des compteurs anarchiques

```
static private int valeur = 0;
static private Object verrou = new Object();
static private int part = 20_000 / nbThreads;
...
  for (int i = 1; i <= part; i++){
    synchronized(verrou){ valeur++ ; }
  }
```

```
static private int valeur = 0;
static private ReentrantLock verrou = new ReentrantLock();
// static private ReentrantLock verrou = new ReentrantLock(true);
...
  for (int i = 1; i <= part; i++){
    verrou.lock();
    try { valeur++ ; }
    finally { verrou.unlock(); }
  }
```

Comparaison de trois verrous (1/2)



Comparaison de trois verrous : interprétations

L'usage d'un verrou équitable a un coût.

Comment l'expliquer ?

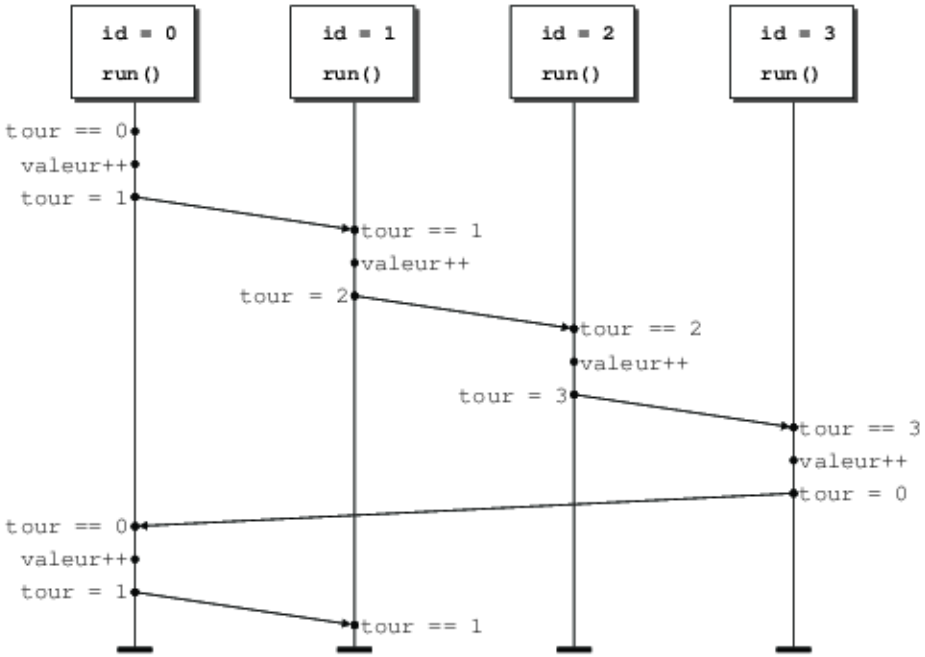
— Les appels à un verrou équitable prennent plus de temps, car il faut gérer l'équité.

— L'équité empêche la machine virtuelle de réaffecter immédiatement le verrou au thread qui vient de le relâcher, qui le réclame à nouveau et qui, envoyé dans l'état BLOCKED, risque de perdre l'accès au processeur.

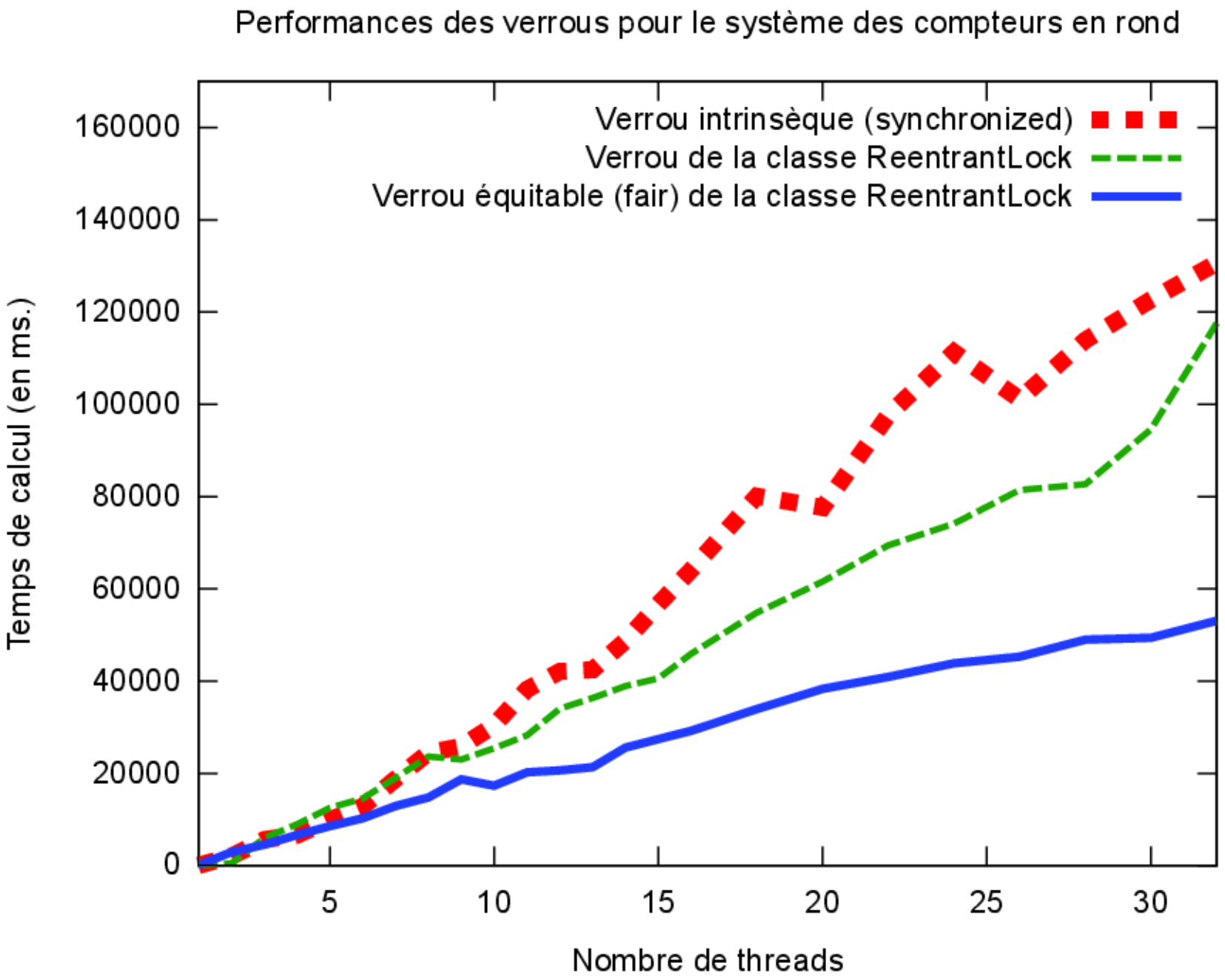
— Autre chose ?

Cas des compteurs en rond

```
public void run() {
   int i = 1;
   while ( i <= part ) {
     synchronized(verrou){
       if ( tour == id) {                    // Si c'est mon tour,
         valeur++ ;                          // j'incrémente valeur,
         tour = (tour+1) % nbActeurs ;       // je passe au voisin
         i++ ;                               // et j'ai alors réussi une
       }                                     // de mes incrémentations.
     }
   }
}
```



Comparaison de trois verrous (2/2)



- ✔ *De nouveaux types de verrous*
- ☛ *Les verrous de Lecture/Écriture*

Les verrous de lecture-écriture : `ReentrantReadWriteLock`

Placer un simple verrou sur un objet (ou une collection d'objets) assure que tout accès à cet objet s'effectuera en exclusion mutuelle.

C'est bien pour bien commencer !

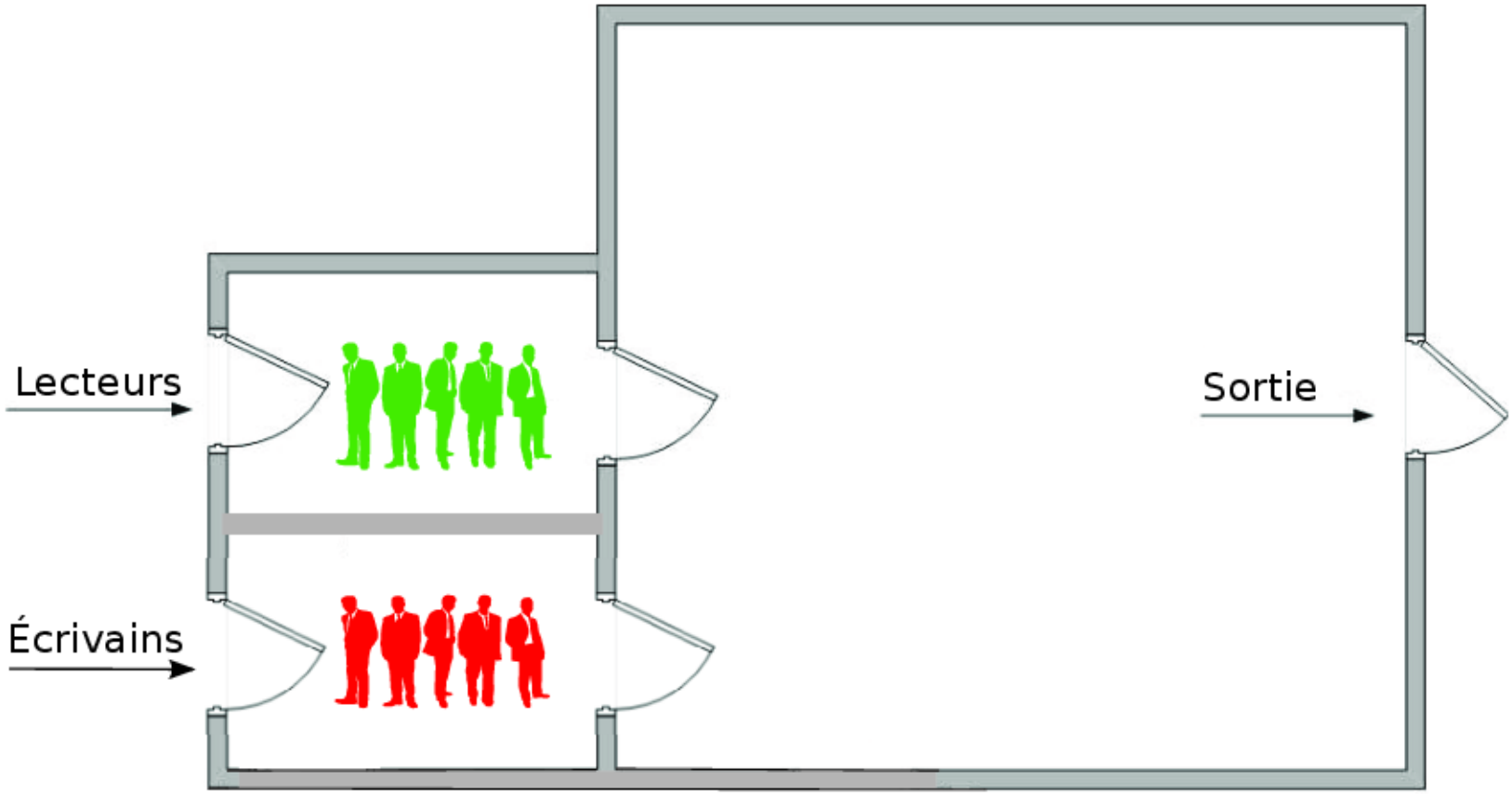
Mais ceci interdit d'effectuer *des lectures en parallèle*, qui pourtant ne posent a priori aucun problème (ni en terme d'atomicité, ni sur le plan matériel).

Afin d'améliorer les performances, il est donc naturel de vouloir

- *autoriser les lectures de l'objet en parallèle* ;
- *interdire les lectures de l'objet lors d'une écriture*.

La classe **`ReentrantReadWriteLock`** permet justement de séparer les actions de lectures (autorisées en parallèle) et les actions d'écriture (en exclusion mutuelle).
Ses objets sont formés deux verrous distincts, mais associés : un **verrou en lecture** (appelé « read-lock ») et un **verrou en écriture** (appelé « write-lock »).

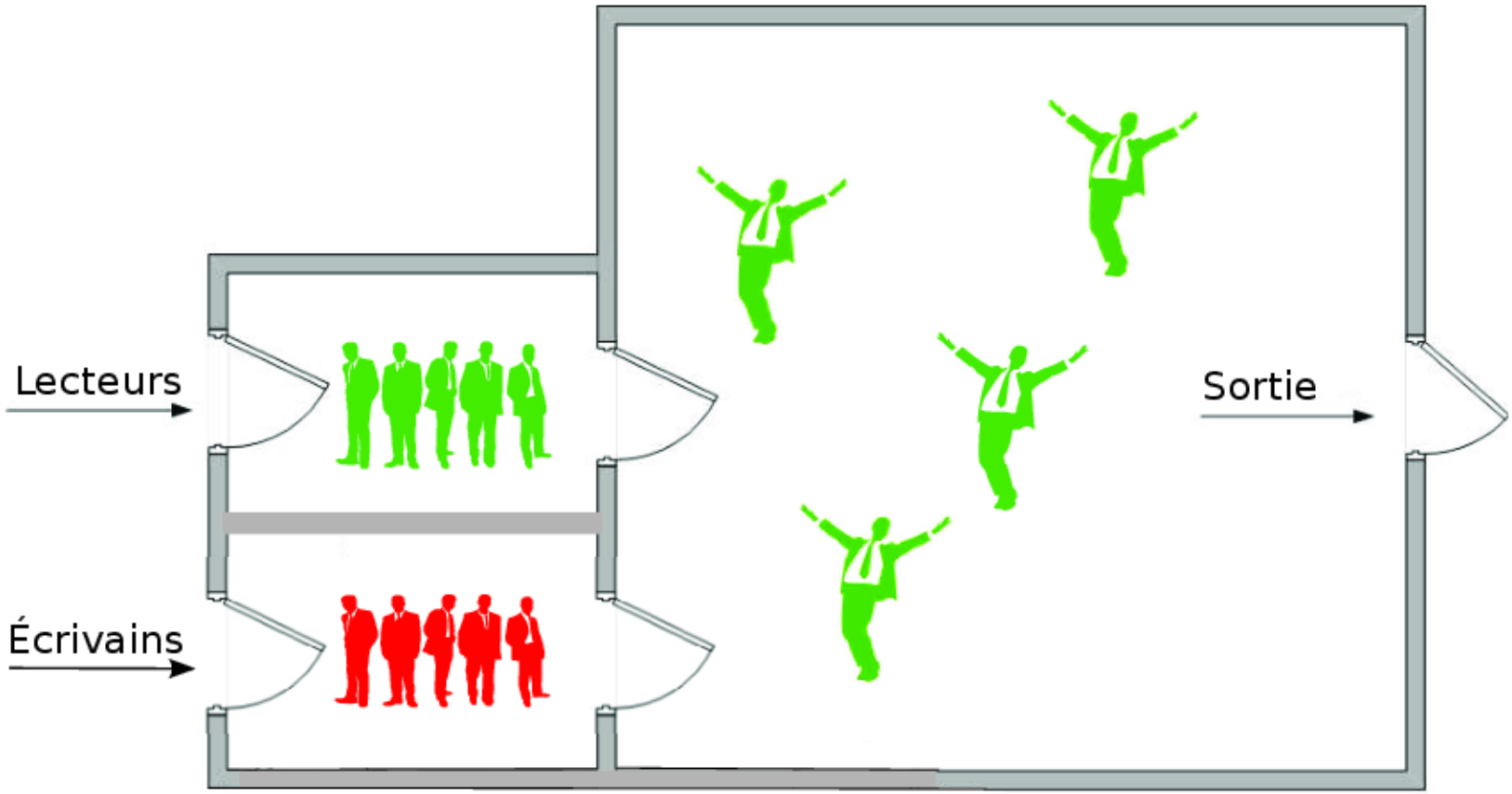
Structure d'un verrou de Lecture/Ecriture



Il y a deux files d'attentes associées à ce verrou de lecture/écriture :

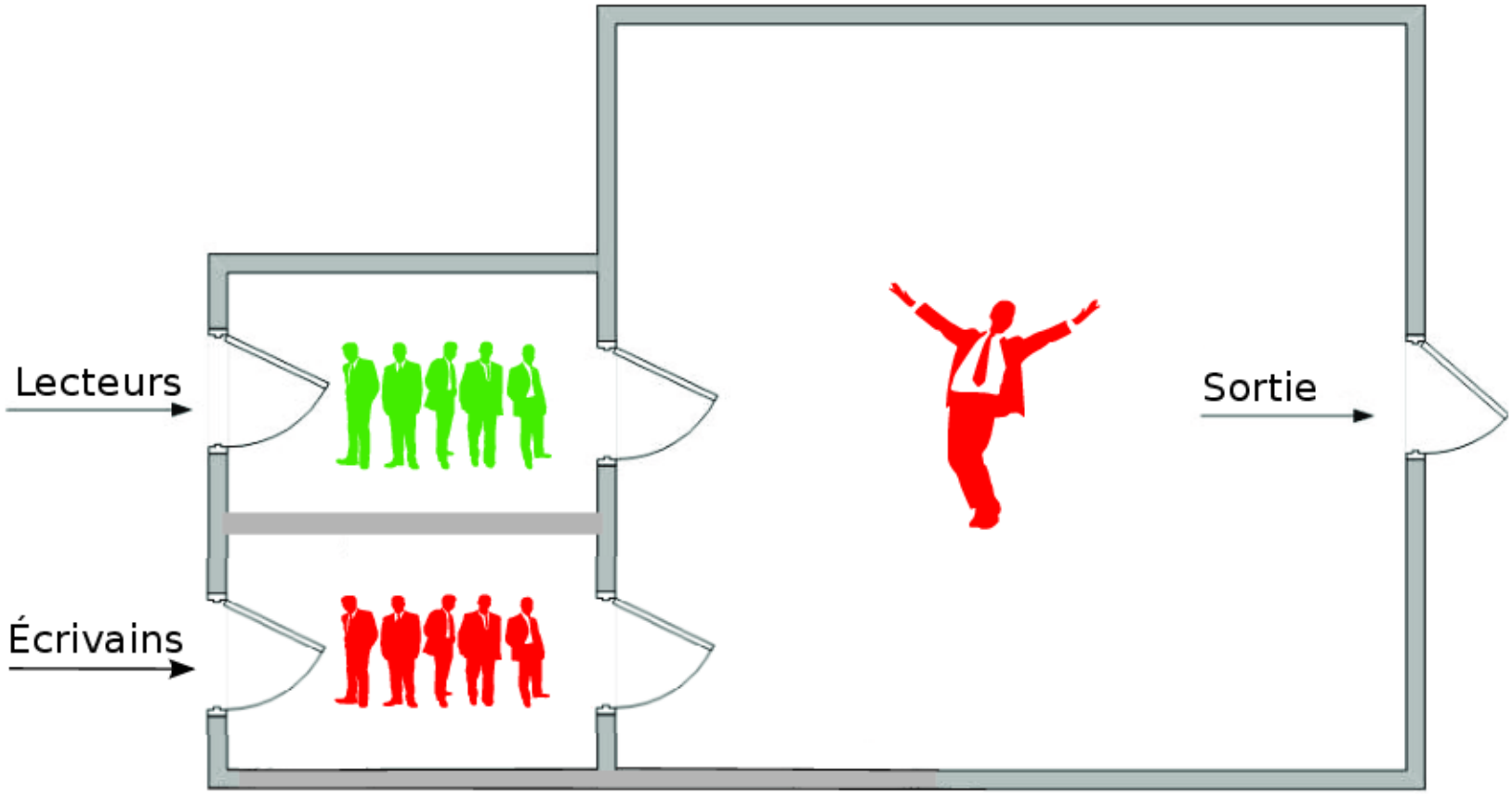
- Les lecteurs attendent le verrou de lecture ;
- Les écrivains attendent le verrou d'écriture.

Fonctionnement d'un verrou de Lecture/Ecriture



Plusieurs threads peuvent posséder le verrou de lecture simultanément ! Ils sont alors implicitement autorisés à lire en parallèle les données protégées par ce verrou.

Fonctionnement d'un verrou de Lecture/Ecriture



Lorsqu'un thread possède le verrou d'écriture, aucun **autre** thread ne possède le verrou de lecture, ni le verrou d'écriture. Il pourra ainsi modifier de manière atomique les données protégées par ce verrou de lecture-écriture.

N.B. Ce thread pourra réclamer en sus le verrou de lecture : il possèdera alors les deux verrous !

ReentrantReadWriteLock

Ainsi, une implémentation de **`ReentrantReadWriteLock`** garantira que

① Plusieurs threads peuvent posséder le verrou de lecture simultanément.

② Si un thread possède le verrou d'écriture alors aucun autre thread ne peut posséder le verrou de lecture, ni même le verrou d'écriture.

③ L'obtention du verrou de lecture est possible si l'on possède déjà le verrou d'écriture.

En revanche **il n'est pas possible de prendre le verrou d'écriture alors que l'on possède le verrou de lecture associé.**

C'est l'une des limites importantes de ces verrous !

Quand peut-on espérer un gain de performance ?

Ces verrous de lecture-écriture s'appuient sur une implémentation plus coûteuse en temps que les verrous simples. L'application ne sera plus performante que si le gain obtenu par les lectures en parallèle est supérieur au coût de l'utilisation de ces verrous.

La Javadoc dit : ≪ ReentrantReadWriteLocks can be used to improve concurrency in some uses of some kinds of Collections. This is typically worthwhile only when *the collections are expected to be large*, accessed by *more reader threads than writer threads*, and *entail operations with overhead that outweighs synchronization overhead*. ≫

Dans le cas contraire, utiliser ces verrous de lecture-écriture peut engendrer un **surcoût**.

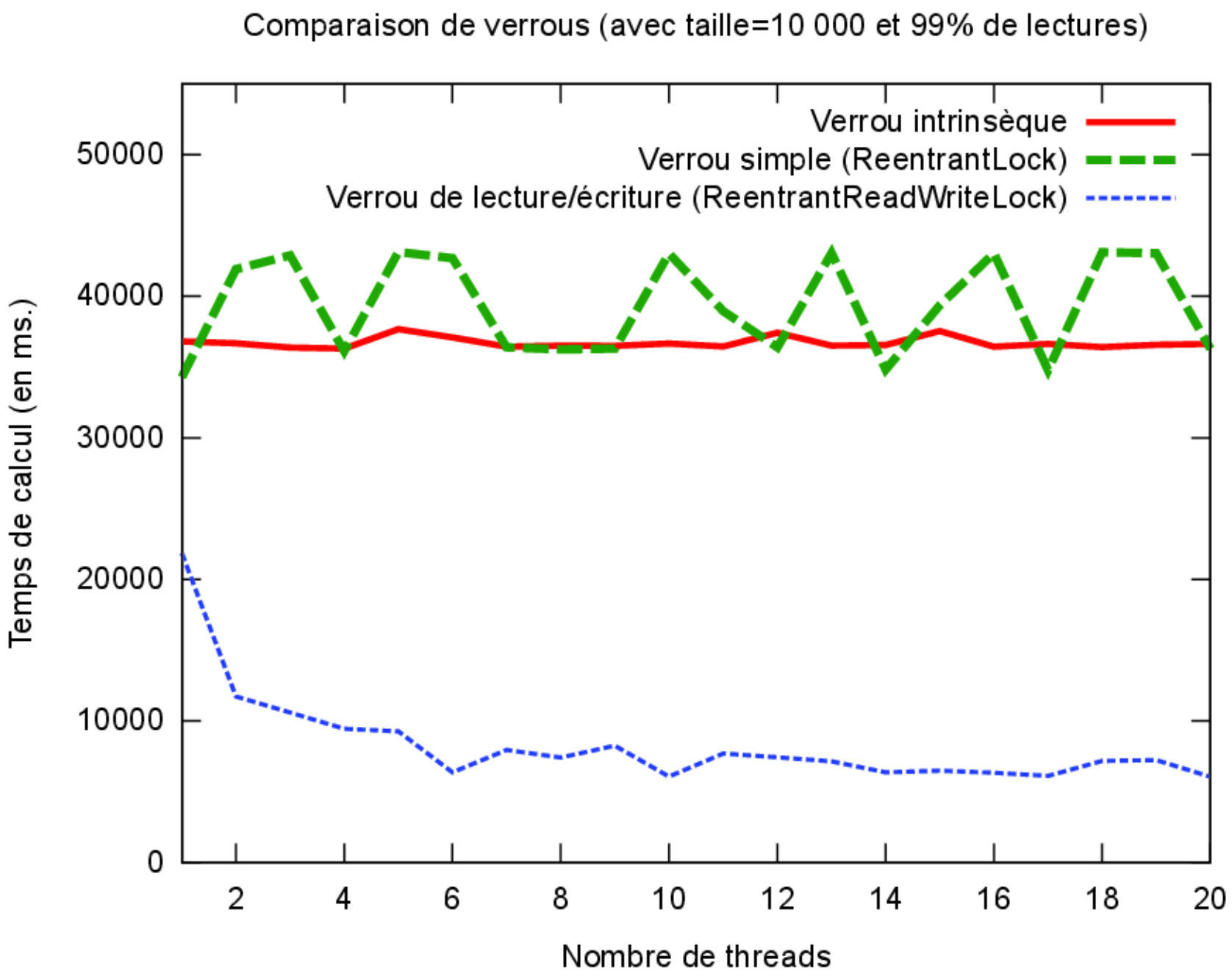
Nouveau benchmark : un tableau de booléens

```
static int taille = 10_000 ;
static boolean[] drapeaux = new boolean[taille];
public void run() {
  for (int i=1 ; i <= part ; i++){
    if (alea.nextInt(100)<1) { // 1% d'écritures
      synchronized(verrou){
        drapeaux[alea.nextInt(taille)] = alea.nextBoolean() ;
      }// Ecriture d'un booléen aléatoire dans une case aléatoire
    } else { // 99% de lectures
      synchronized(verrou){
        somme = 0 ;
        for (int j=0 ; j<taille ; j++) if (drapeaux[j]) somme++ ;
      } // Calcul du nombre de booléens égaux à true
    }
  }
}
```

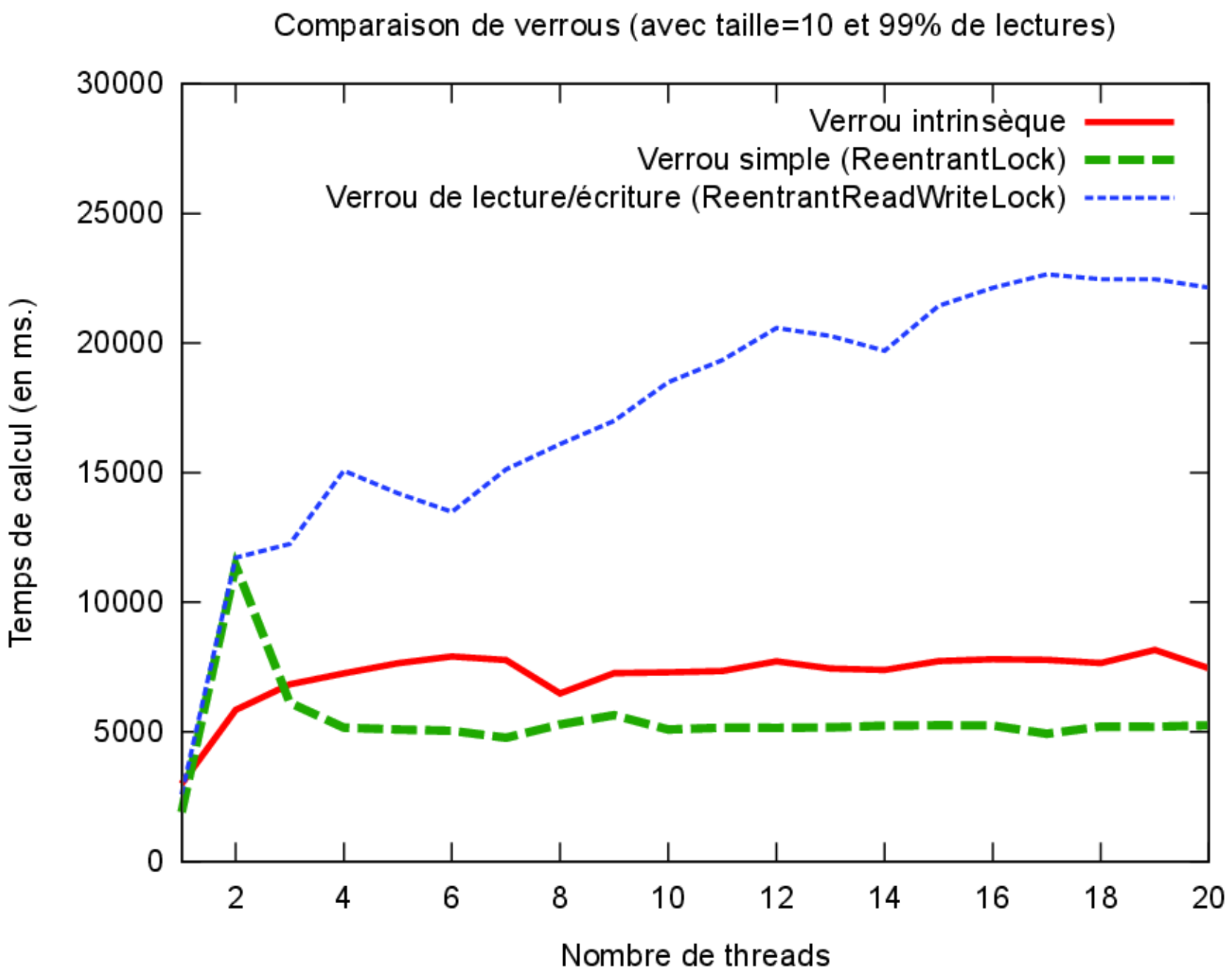
Modification de `run()` avec un verrou `rwl` de Lecture/Ecriture

```
static ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
...
for (int i=1 ; i <= part ; i++){
  if (alea.nextInt(100)<1) {  // 1% d'écritures
    rwl.writeLock().lock() ;  // Je prends le verrou d'écriture
    try {
      drapeaux[alea.nextInt(taille)] = alea.nextBoolean() ;
    } finally { rwl.writeLock().unlock() ; }
  } else { // 99% de lectures
    rwl.readLock().lock() ;   // Je prends le verrou de lecture
    try {
      somme = 0 ;
      for (int j=0 ; j < taille ; j++) if (drapeaux[j]) somme++ ;
    } finally { rwl.readLock().unlock() ; }
  }
}
```

Illustration du gain sur un tableau à 10 000 éléments



Le défaut des verrous RWL : cas d'un tableau à 10 élements



- ✔ *De nouveaux types de verrous*
- ✔ *Les verrous de Lecture/Écriture*
- ☛ *Loquets, barrières, etc.*

Loquet avec compteur

Les objets de la classe

`java.util.concurrent.CountDownLatch`

sont initialisés dans le constructeur, avec un entier.

Un tel objet est décrémenté par la méthode **`countDown()`** qui est non bloquante.

Un ou plusieurs threads se mettent en attente sur ce loquet par **`await()`** jusqu'à ce que le compteur ≪ tombe ≫ à zéro.

N.B. Le compteur ne peut pas être réinitialisé.

Barrière cyclique

Un objet **`barriere`** de la classe

`java.util.concurrent.CyclicBarrier`

représente intuitivement une barrière derrière laquelle n threads (où n est un paramètre du constructeur) s'inscrivent pour attendre en appliquant la méthode **`barriere.await()`**.

Quand n threads y sont arrivés, la barrière « se lève » et les threads continuent leur exécution.

Chaque méthode **`await()`** retourne l'ordre d'arrivée à la barrière.

La barrière est dite « cyclique » car elle se rabaisse ensuite, et les threads suivants y sont à nouveau bloqués.

Le constructeur accepte un **`Runnable`** spécifiant du code à exécuter par le dernier thread arrivé à la barrière.

Barrière cyclique brisée (Broken barrier)

Les barrières sont des objets du même genre que **`CountDownLatch`**, mais réinitialisables explicitement par la méthode **`reset()`**.

Lorsqu'un thread quitte le point de barrière prématurément, à cause d'une interruption ou lors d'un reset de la barrière, tous les threads en attente sur **`await()`** lèvent **`BrokenBarrierException`**.

Les loquets ont un rôle similaire à une barrière mais ils ne peuvent servir qu'une seule fois et leur fonctionnement est différent :

- N'importe quel thread peut décrémenter le compteur en appliquant **`cdl.countDown()`** ;
- D'*autres threads* peuvent pendant ce temps attendre que le compteur soit à zéro s'ils appliquent **`cdl.await()`**.

Outils pratiques pour le multitâche

Master Informatique — Semestre 2 — UE obligatoire de 3 crédits

Année 2017-2018

Version du 28 janvier 2018

Maître, esclaves et tâches

L'idée principale

Un **thread** sert à mener à bien le déroulement d'une **tâche**.

Approche simple : créer un thread pour chaque nouvelle tâche.

- Facile et correct. C'est bien pour commencer...
- Mais peu performant s'il y a beaucoup de threads car la gestion d'un thread par la JVM consomme du temps et de la mémoire. Le surcoût peut être non négligeable surtout si la tâche est courte ou petite !

Approche plus sophistiquée : utiliser un *réservoir de threads* (en anglais, on dit « thread pool ») qui traite dynamiquement les tâches soumises.

Thread pool : les grands principes

① Il faut **installer un réservoir** de threads à l'avance.

- On évite de créer un thread à chaque nouvelle tâche ;
- Le délai d'exécution d'une tâche est réduit, car le thread qui l'exécute est déjà créé ;
- Le surcoût de création des threads est alors amorti sur l'ensemble des tâches qui seront exécutées ;
- Ceci permet de limiter l'utilisation des ressources mémoires de la JVM puisqu'il y a moins de threads à gérer, même s'il y a beaucoup de tâches ;
- Le nombre de threads utilisés peut être fixé en fonction du nombre de coeurs disponibles sur la machine.

② Il reste alors seulement à gérer la liste des tâches **soumises** au réservoir.

Quelques difficultés courantes à cette approche

① Il faut trouver, souvent de manière empirique, la **bonne taille** pour le réservoir de threads, en fonction du nombre de coeurs disponibles et du type de tâches à exécuter.

② Une tâche en cours d'exécution peut éventuellement **bloquer** : le thread qui l'exécute est alors bloqué lui aussi.

③ S'il y a trop de tâches bloquées, il n'y aura plus (ou quasiment plus) de threads disponibles pour exécuter les nouvelles tâches, ce qui entraînera des *performances dégradées*. On parle alors de ≪ thread leakage ≫.

④ Il y a **un risque accrû d'interblocage**, s'il n'y a plus de thread disponible pour exécuter une tâche, qui une fois terminée pourrait débloquer toutes les autres...

- ✔ *Maître, esclaves et tâches*
- ☛ *Interface Executor depuis Java 5*

Qu'est-ce qu'un executor ?

C'est un objet permettant d'exécuter des tâches.

```
public interface Executor {
  void execute (Runnable tache);
}
```

La méthode **`execute()`** permet de soumettre un **`runnable`** à l'exécuteur.

Un **`Executor`** permet de découpler

① ce qui doit être fait, c'est-à-dire la liste des tâches, définies en tant que **`Runnable`**

② de quand, et par qui, les tâches seront exécutées à l'aide d'un **`Thread`**.

Soumission d'une tâche à un `Executor`

Si `r` est un objet `Runnable`, et `e` un objet `Executor`, on peut remplacer

```
(new Thread(r)).start();
```

par

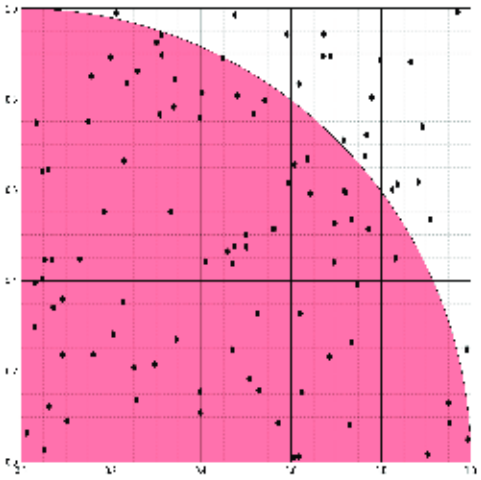
```
e.execute(r);
```

Sans surprise, l'exécution d'une tâche est **asynchrone** : une tâche est successivement soumise, en cours d'exécution, puis terminée.

On ne peut pas savoir a priori quand une tâche sera terminée : il faudra donc *détecter la terminaison*.

Exemple : estimation de $\pi/4$ par Monte-Carlo (1/2)

```
class MultipleLancer implements Runnable {
  long nbTirages;                    // à réaliser par chaque tâche
  long tiragesDansLeDisque = 0 ; // obtenus par chaque tâche
  MultipleLancer(long nbTiragesAEffectuer){
    nbTirages = nbTiragesAEffectuer;
  }
  public void run(){
    double x, y;
    for (int i = 0; i < nbTirages; i++) {
      x = ThreadLocalRandom.current().nextDouble(1);
      y = ThreadLocalRandom.current().nextDouble(1);
      if (x * x + y * y <= 1) tiragesDansLeDisque++ ;
    }
  }
}
```



C'est un Runnable tout-à-fait naturel

À propos de ThreadLocalRandom

Qu'est-ce qu'un **`ThreadLocalRandom`** ?

La Javadoc dit : ≪ A random number generator isolated to the current thread. Like the global Random generator used by the Math class, a ThreadLocalRandom is initialized with an internally generated seed that may not otherwise be modified. When applicable, use of ThreadLocalRandom rather than shared Random objects in concurrent programs will typically encounter much less overhead and contention. Use of ThreadLocalRandom is particularly appropriate when multiple tasks (for example, each a ForkJoinTask) use random numbers in parallel in thread pools. ≫

On remplace donc

```
x = Math.random() ;
```

du programme séquentiel par

```
x = ThreadLocalRandom.current().nextDouble(1);
```

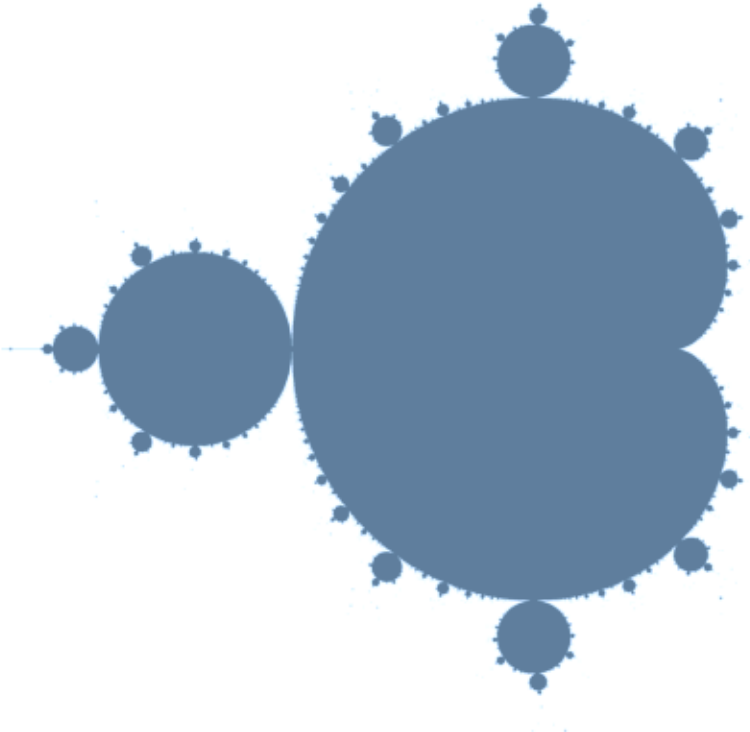
Exemple : estimation de $\pi/4$ par Monte-Carlo (2/2)

```
ExecutorService executeur = Executors.newFixedThreadPool(10) ;
MultipleLancer[] tableDesTaches = new MultipleLancer[10] ;
for (int j = 0; j < 10 ; j++) {
  tableDesTaches[j] = new MultipleLancer( nbTirages/10 ) ;
  executeur.execute(tableDesTaches[j]) ;
}
executeur.shutdown();    // Il n'y a plus aucune tâche à soumettre
while (! executeur.isTerminated()) {// Tant que ce n'est pas fini
  Thread.sleep(1000) ;          // Ce n'est pas une attente active !
  System.out.print("#") ;
}
long tiragesDansLeDisque = 0 ;
for (int j = 0; j < 10 ; j++) {
  tiragesDansLeDisque += tableDesTaches[j].tiragesDansLeDisque ;
}
double resultat = (double) tiragesDansLeDisque / nbTirages ;
```

En guise de Travaux Pratiques

Dans l'exemple précédent, pour simplifier, il y a 10 tâches et 10 threads.

En TP, vous vous inspirerez du code précédent pour utiliser un réservoir de threads afin de calculer l'image ci-dessous :



avec un réservoir de 4 threads et une tâche par ligne de pixels.

- ✔ *Maître, esclaves et tâches*
- ✔ *Interface Executor depuis Java 5*
- ☛ *Tâches asynchrones*

Motivation

Lorsqu'une tâche (qui sera exécutée par un réservoir de threads) doit renvoyer un résultat, au lieu d'être **`Runnable`**, elle doit être déclarée **`Callable`**.

Il n'y a qu'une seule méthode à implémenter pour l'interface **`Callable`** :

```
V call() throws Exception
```

La méthode **`call()`** devra contenir une instruction **`return()`**, car elle doit renvoyer une valeur.

Exemple de Callable

```
class MultipleLancer implements Callable<Long>{
  long nbTirages;
  long tiragesDansLeDisque = 0 ;
  MultipleLancer(long nbTiragesAEffectuer){
    nbTirages = nbTiragesAEffectuer;
  }
  public Long call(){
    double x, y;
    for (int i = 0; i < nbTirages; i++) {
      x = ThreadLocalRandom.current().nextDouble(1);
      y = ThreadLocalRandom.current().nextDouble(1);
      if (x * x + y * y <= 1) tiragesDansLeDisque++ ;
    }
    return tiragesDansLeDisque;
  }
}
```

Notez le return() dans la méthode call().

Problème des tâches qui renvoient un résultat

Lorsque l'on veut soumettre à un **`Executor`** une tâche particulière **qui renvoie un résultat**, on ne sait pas en général (et le plus souvent, on ne veut pas savoir) quand la tâche sera exécutée ni quand précisément le résultat de la tâche sera disponible.

Le résultat de la tâche va être manipulé à l'aide d'un objet de type **`Future`**.

Pour soumettre à un **`ExecutorService`** une telle tâche, il faut utiliser la méthode **`submit()`** (à la place de **`execute()`**) :

```
Future<V> submit(Callable<V> tache)
```

L'objet **`Future<V>`** renvoyé par **`submit()`** va permettre de s'interroger sur le résultat que la tâche va produire.

- Le **`Future`** est renvoyé *immédiatement* lors de la soumission ;
- Le **`Future`** représente une *promesse* de réponse ;
- Le **`Future`** permet de s'enquérir de l'avancement de la tâche, ou de l'annuler.

Exemple de Future

```
ExecutorService executeur = Executors.newFixedThreadPool(10) ;
ArrayList<Future<Long>> listeDesPromesses =
                                        new ArrayList<Future<Long>>() ;
for (int j = 0; j < 10 ; j++){
  MultipleLancer tache = new MultipleLancer( nbTirages/10 ) ;
  Future<Long> promesse = executeur.submit(tache) ;
  listeDesPromesses.add(promesse) ;    // On stocke les promesses
}
long tiragesDansLeDisque = 0 ;
for (int j = 0; j < 10 ; j++){
  Future<Long> promesse = listeDesPromesses.get(j);// Dans l'ordre
  tiragesDansLeDisque += promesse.get();           // Bloquant !
}
double resultat = (double) tiragesDansLeDisque / nbTirages ;
executeur.shutdown() ; // Il n'y a plus aucune tâche à soumettre
```

- ✔ *Maître, esclaves et tâches*
- ✔ *Interface Executor depuis Java 5*
- ✔ *Tâches asynchrones*
- ☛ *Les services de completion*

Construction et clôture d'un réservoir de threads

La construction d'un réservoir de threads a un coût : il peut être intéressant dans certaines applications d'utiliser le même réservoir pour plusieurs séries de tâches, et donc de pouvoir **récupérer les résultats sans clôre le réservoir**.

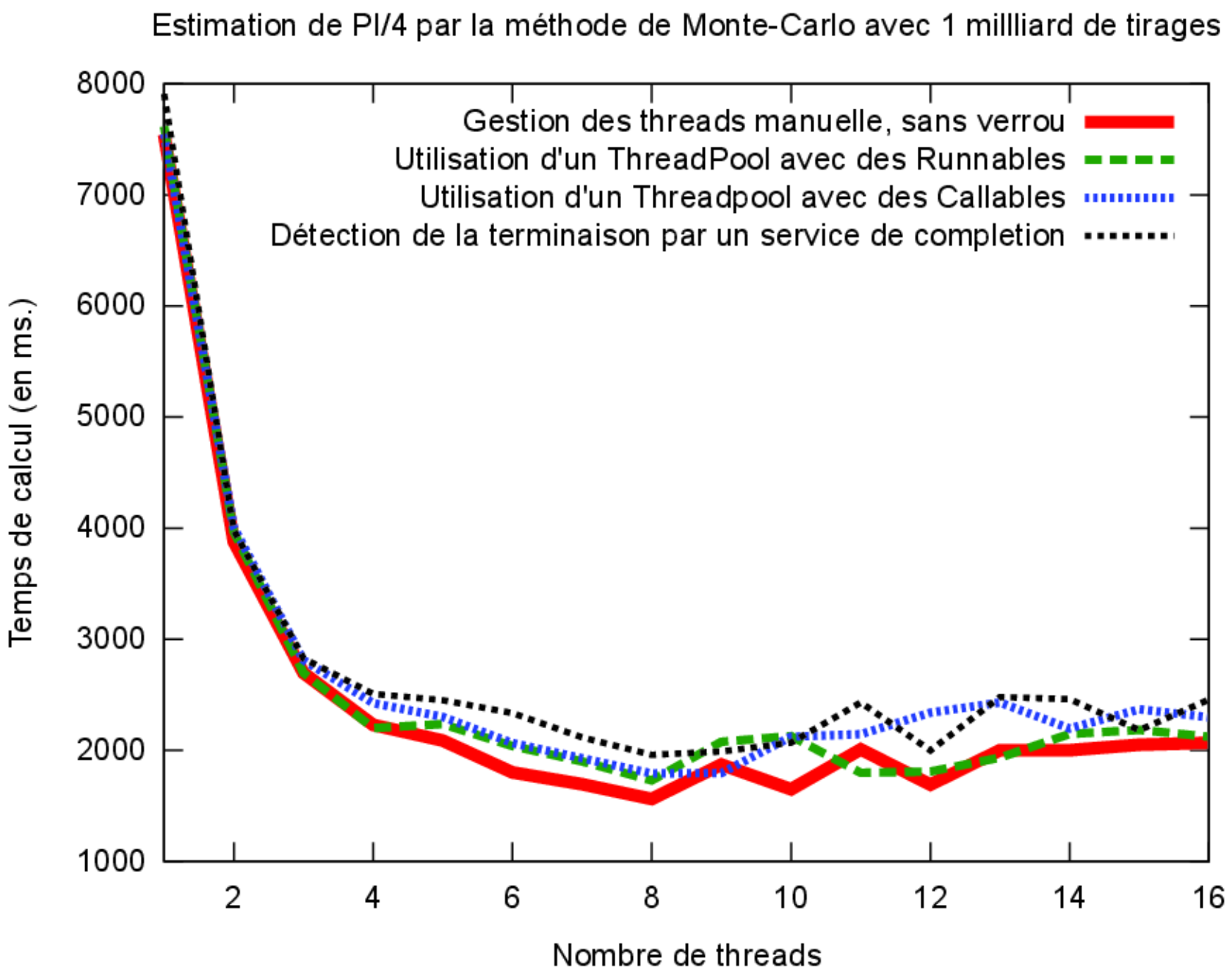
Plusieurs approches sont possibles :

- Utiliser *un loquet* pour détecter la fin d'un ensemble de tâches ;
- Guetter les fins des tâches asynchrones soumises à l'aide d'objets de type `Futures`, *même si la tâche ne renvoie rien !*
- Utiliser un **service de complétion** : c'est souvent le plus simple !

Utiliser un service de completion

```
ExecutorService executeur=Executors.newFixedThreadPool(10) ;
CompletionService<Long> ecs =
                    new ExecutorCompletionService<Long>(executeur) ;
for (int j = 0; j < 10 ; j++) {
  MultipleLancer tache = new MultipleLancer( nbTirages/10 );
  ecs.submit(tache);
}
int tiragesDansLeDisque = 0;
for (int j = 0; j < 10; j++) {
    Long resultatAttendu = ecs.take().get(); // Ordre indéterminé!
    tiragesDansLeDisque += resultatAttendu;
}
double resultat = (double) tiragesDansLeDisque / nbTirages ;
```

À propos des performances, sur une machine à 8 coeurs



Les trois types de collections en Java

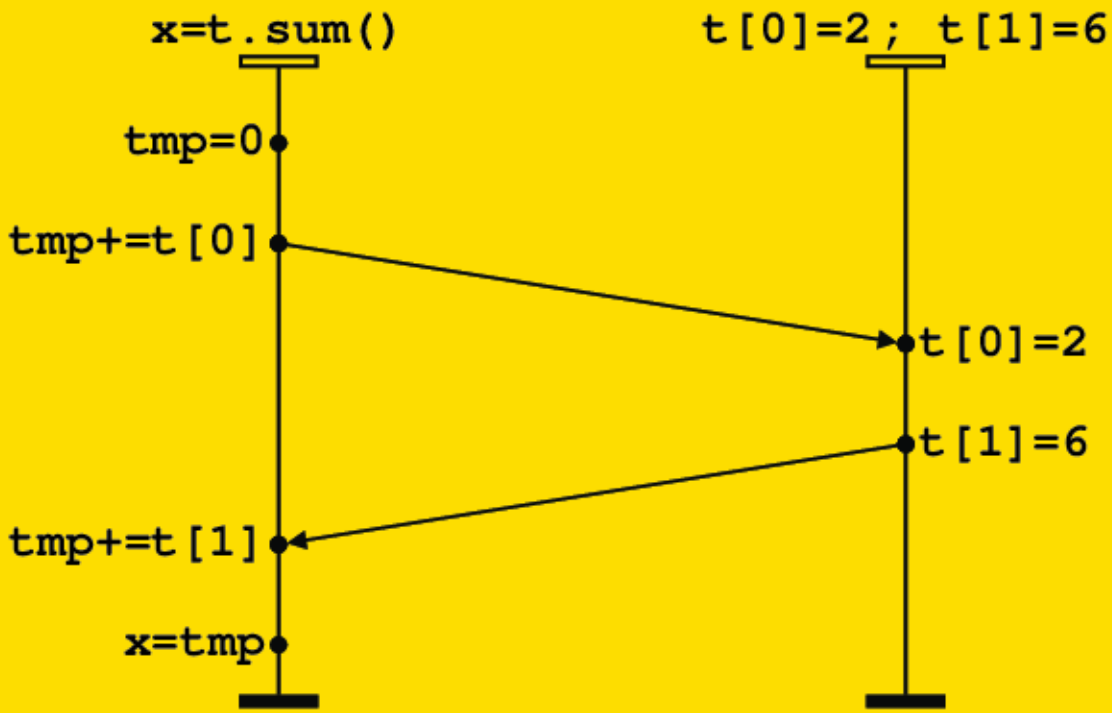
Master Informatique — Semestre 2 — UE obligatoire de 3 crédits

Année 2017-2018

Version du 28 janvier 2018

Itérer sur une collection peut poser des problèmes

Initialement `t[0]=1` et `t[1]=3`



Que vaut `x` à la fin ?

La valeur retournée par **`sum()`** peut correspondre à une liste qui n'a jamais existé !

Collection et concurrence

En Java, il y a trois façons d'obtenir des collections qui fonctionnent correctement en présence de plusieurs threads : on dit « *thread-safe* » pour faire court.

① *Utiliser des collections « synchronisées »*, telles que **`Vector`** ou **`HashTable`**... Chaque appel requiert le *verrou intrinsèque* de l'objet (comme dans un moniteur).

② *Construire des classes synchronisées* à partir de collections qui ne le sont pas (depuis le JDK 1.2).

③ *Utiliser les nouvelles collections « concurrentes »* présentes depuis JDK 1.5 ; celles-ci n'utilisent pas **`synchronized`** ; elles sont construites à l'aide de verrous privés ou d'objets atomiques. Ces classes autorisent les accès simultanés et sont donc potentiellement plus performantes que les collections synchronisées.

Il faut retenir ici qu'il existe en effet des collections qui ne sont pas thread-safe : **`HashMap`**, **`ArrayList`**, etc. mais que l'on peut avantageusement utiliser si l'on sait qu'elles ne seront jamais utilisées par plusieurs threads simultanément.

À propos de Vector

The Vector class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.

...

As of the Java 2 platform v1.2, this class was retrofitted to implement the List interface, making it a member of the Java Collections Framework. Unlike the new collection implementations, Vector is synchronized. If a thread-safe implementation is not needed, it is recommended to use ArrayList in place of Vector.

Construction d'une collection synchronisée sur une qui ne l'est pas

La classe générique **`ArrayList<E>`** ne peut pas être utilisée telle qu'elle dans un contexte multi-thread. Il faut la ≪ synchroniser ≫.

```
List<String> list = synchronizedList(new ArrayList<String>());
```

Tous les appels sur la collection obtenue sont alors ≪ **`synchronized`** ≫.

Autres constructions disponibles :

```
Collection<T> synchronizedCollection(Collection<T> c)
Map<K,V> synchronizedMap(Map<K,V> m)
Set<T> synchronizedSet(Set<T> s)
SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m)
SortedSet<T> synchronizedSortedSet(SortedSet<T> s)
```

Protection des itérateurs des collections synchronisées

```
public static void main(String[] args) {
  final List<String> maListe =
          Collections.synchronizedList(new ArrayList<String>());
          // Fabrique d'une collection synchronisée
  new Thread ( new Runnable() {
              public void run() {
                for(;;) {
                  synchronized(maListe) { // Sécurité nécessaire
                    for(String s:maListe) System.out.println(s);
                  }
                  try { Thread.sleep(1000); } catch(...) {...}
                }
              }}).start();
  for(int i=0; i<100_000; i++) maListe.add(Integer.toString(i));
}
```

Itérateurs de collections synchronisées

Les itérateurs d'une classe synchronisée ne sont pas « **`synchronized`** » : ils sont même « fail-fast : » ils sont susceptibles de lancer l'exception

`ConcurrentModificationException`

dans chacune des conditions suivantes :

1. Si un thread essaie de modifier une collection pendant qu'un autre itère dessus.
2. Si après la création de l'iterateur, le contenu est modifié par une méthode autre que les méthodes **`remove()`** et **`add()`** appliquées par l'itérateur.

Oracle dit : « Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness : the fail-fast behavior of iterators should be used only to detect bugs. »

À propos de la classe Vector

The iterators returned by this class's iterator and list Iterator methods are fail-fast : if the vector is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

...

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness : the fail-fast behavior of iterators should be used only to detect bugs.

- ✔ *Les collections synchronisées*
- ☛ *Les collections concurrentes*

Les nouvelles collections concurrentes

Ces « nouvelles » collections possèdent des propriétés communes :

1. Il est interdit de stocker **`null`**.
2. Ces collections fonctionnent indépendamment du verrou intrinsèque de ses objets : contrairement aux collections dites synchronisées, protéger une itération sur une collection concurrente à l'aide de **`synchronized`** n'interdira pas d'autres accès simultanés à la collection !
3. Les itérateurs ne sont pas « fail-fast » : ils ne renvoient jamais d'exception du type **`ConcurrentModificationException`**. En revanche les modifications ultérieures à la création de l'itérateur peuvent (ou non) être vues par l'itérateur.
4. Les opérations **`addAll(c)`**, **`removeAll()`**, **`retainAll(c)`** ne sont pas garanties d'être atomiques.

Quelles garanties sont offertes ?

Ce que dit la Javadoc

À propos de la collection concurrente **`ConcurrentLinkedQueue`** :

≪ Additionally, the bulk operations **`addAll`**, **`removeAll`**, **`retainAll`**, **`containsAll`**, **`equals`**, and **`toArray`** are not guaranteed to be performed atomically. For example, an iterator operating concurrently with an **`addAll`** operation might view only some of the added elements. ≫

Implicitement,

- *les collections qui ne sont pas expressément indiquées comme étant ≪ thread-safe ≫ ne le sont pas*. Le programmeur doit donc s'assurer que les collections qu'il utilise sont satisfaisantes pour l'usage qu'il en fait.
- *les méthodes des collections concurrentes qui ne sont pas expressément indiquées comme n'étant pas atomiques le sont*. Le programmeur doit donc connaître les méthodes qui ne sont pas atomiques.

Mais ça ne suffit pas à éviter toutes les erreurs liées à l'atomicité.