À propos du TP A

Master Informatique — Semestre 1 — UE obligatoire de 3 crédits

Année 2018-2019

Version du 18 novembre 2018

Support matériel dans les salles de TP

```
$ lscpu
Architecture:                    x86_64
Mode(s) opératoire(s) des processeurs :32-bit, 64-bit
Byte Order:                      Little Endian
CPU(s):                          4
On-line CPU(s) list:             0-3
Thread(s) par coeur :            1
Coeur(s) par socket :            4
Socket(s):                       1
Noeud(s) NUMA :                  1
Identifiant constructeur : GenuineIntel
...
```

Cet ordinateur dispose donc de 4 coeurs (réunis sur une socket) et permet d'accueillir un seul thread par coeur : il n'y a pas d'hyperthreading.

Performances attendues (1/2)

Avec 4 threads travaillant chacun sur un coeur, on peut espérer diviser le temps de calcul par 4.

Néanmoins :

- le système consomme de la CPU ;
- la machine virtuelle Java consomme de la CPU ;
- il faut prendre soin de fermer toutes les autres applications.

Performances attendues (2/2)

Plus généralement, avec n threads, on peut espérer un temps de calcul divisé par n.

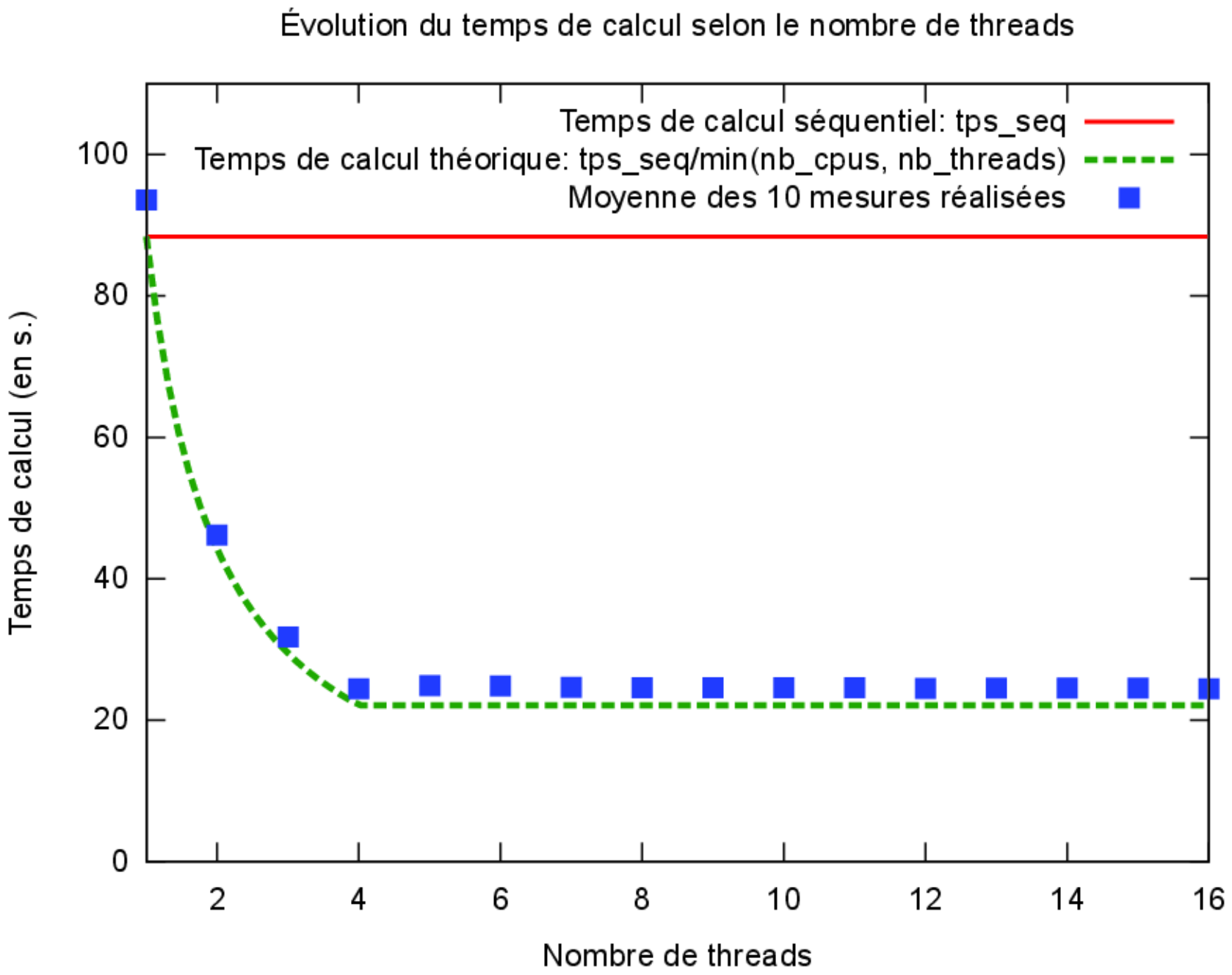
Mais avec seulement p processeurs, il est illusoire d'espérer diviser le temps de calcul par plus que p.

Par conséquent, le gain théorique vaut $\min(p, n)$ et le temps de calcul attendu idéalement vaut

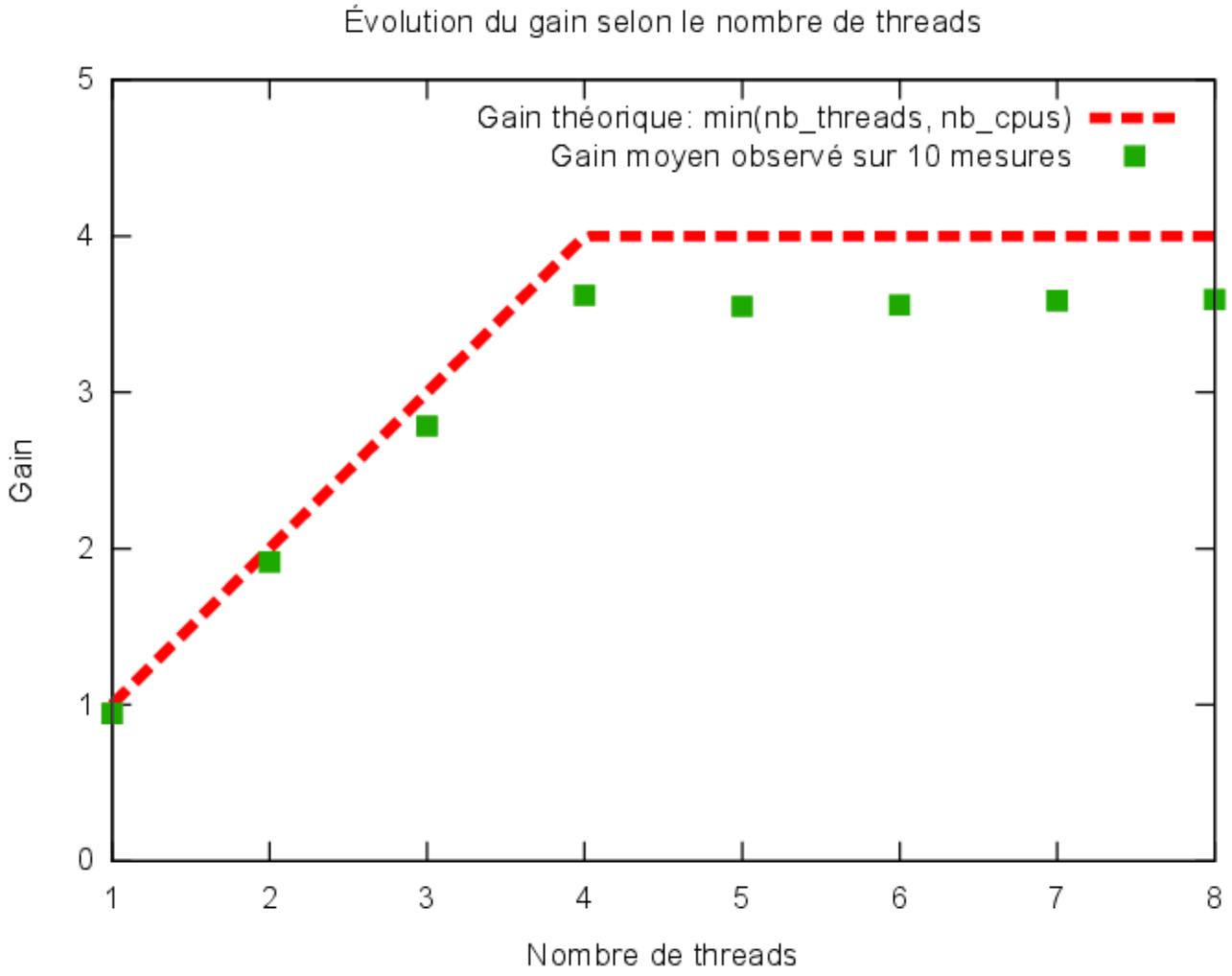
$$\frac{t}{\min(p, n)}$$

où t désigne le temps de calcul du programme séquentiel.

Mesures effectuées : de 1 à 16 threads avec 1 milliard de tirages

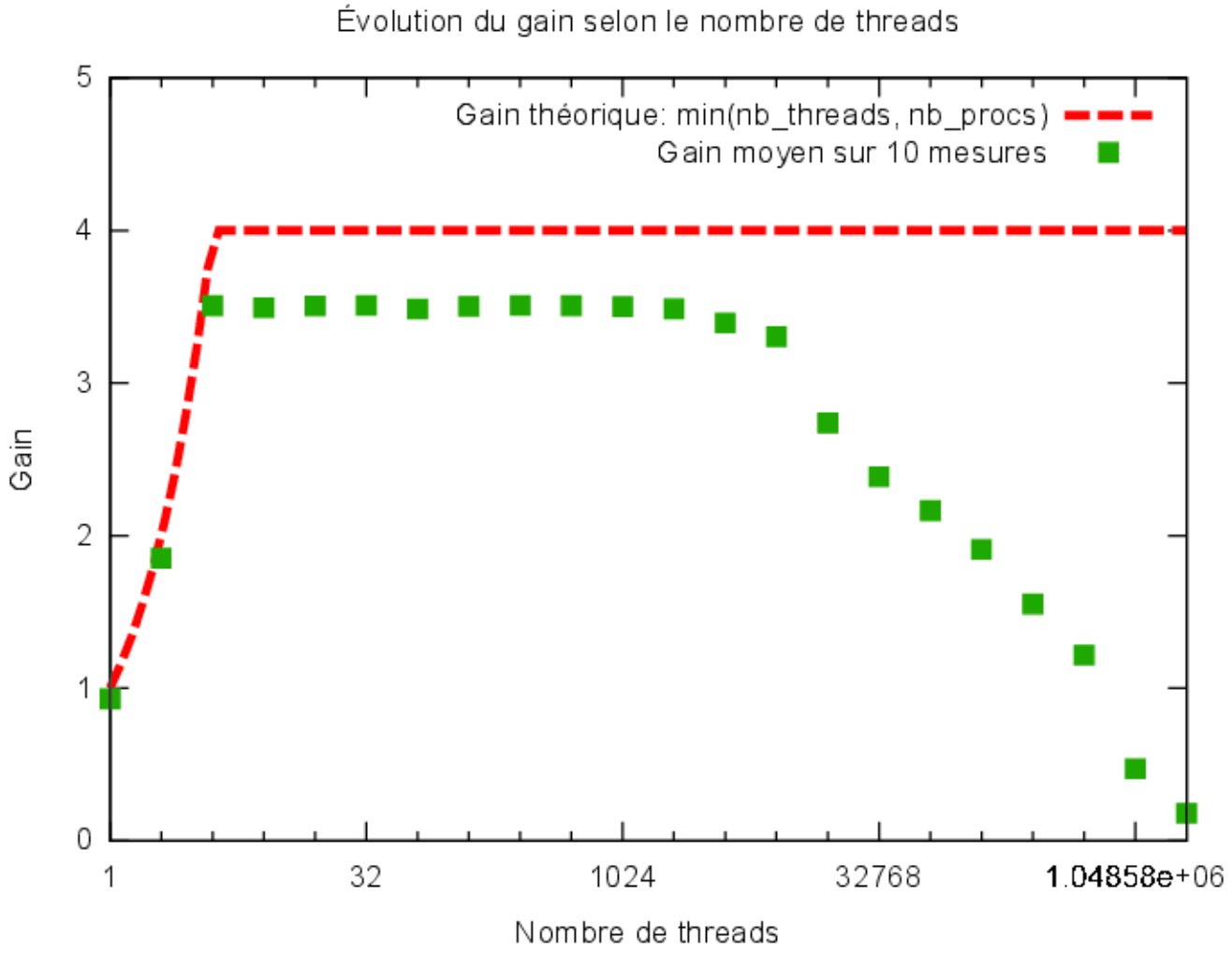


Évolution du gain selon le nombre de threads



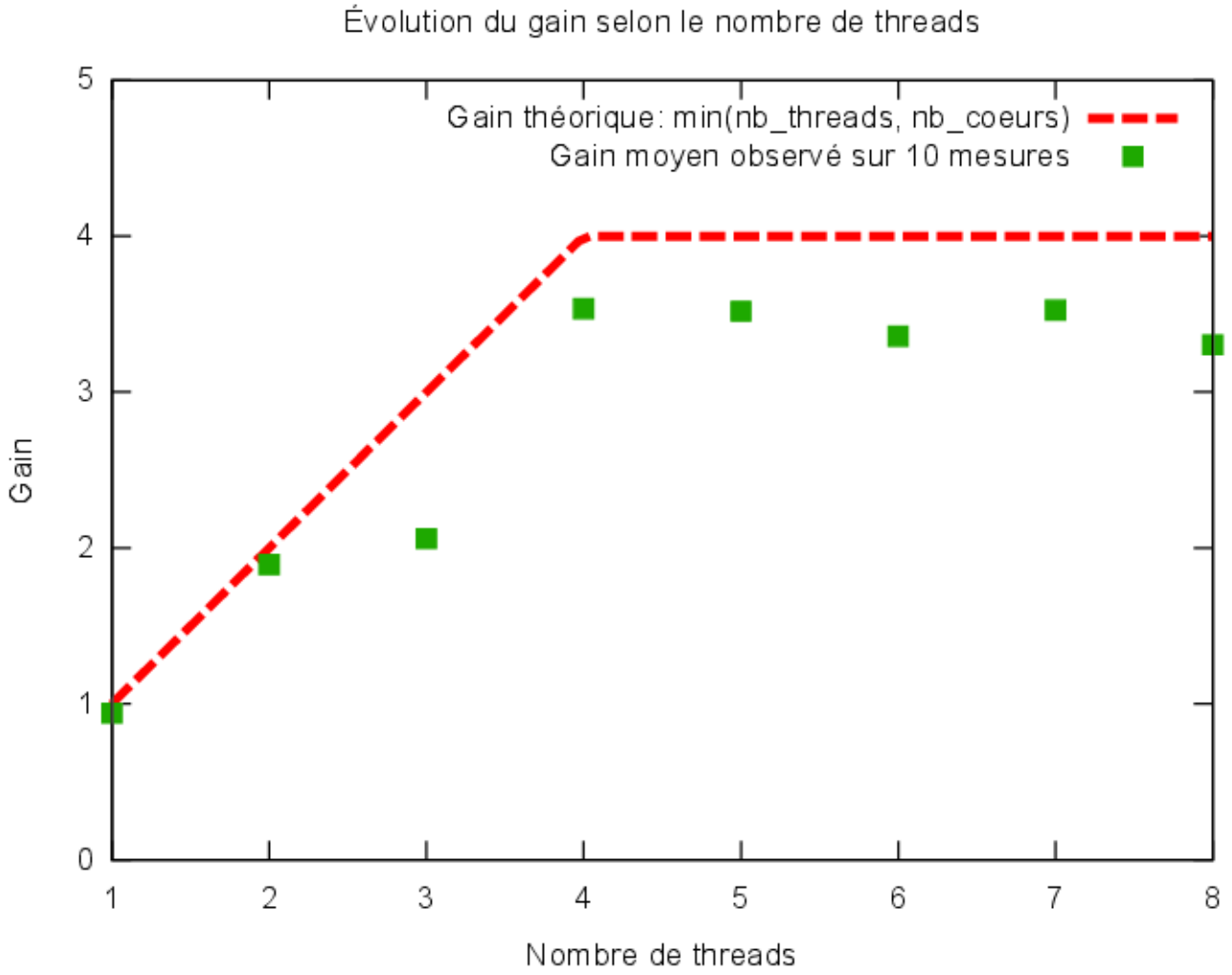
Le gain croît linéairement avec le nombre de threads jusqu'au nombre de processeurs disponibles ; au-delà, le gain stagne. La consommation de temps de calcul par le système et la machine virtuelle explique que l'on n'atteind pas exactement un gain de 4.

Utilité de considérer jusqu'à 2 millions de threads



Si l'on charge Java d'un nombre considérable de threads simultanément, les performances seront dégradées. Avec 2 millions de threads, le temps de calcul vaut 5 fois plus que le programme séquentiel !

Nécessité de réaliser plusieurs mesures (en salle A.314)




```
...
3 10.466 10.542 10.481 10.463 10.546 10.463 46.525 10.468 ...
...
```

✔ *Sur une machine des salles de TP*

☛ *Sur un Macbook Pro*

Support matériel

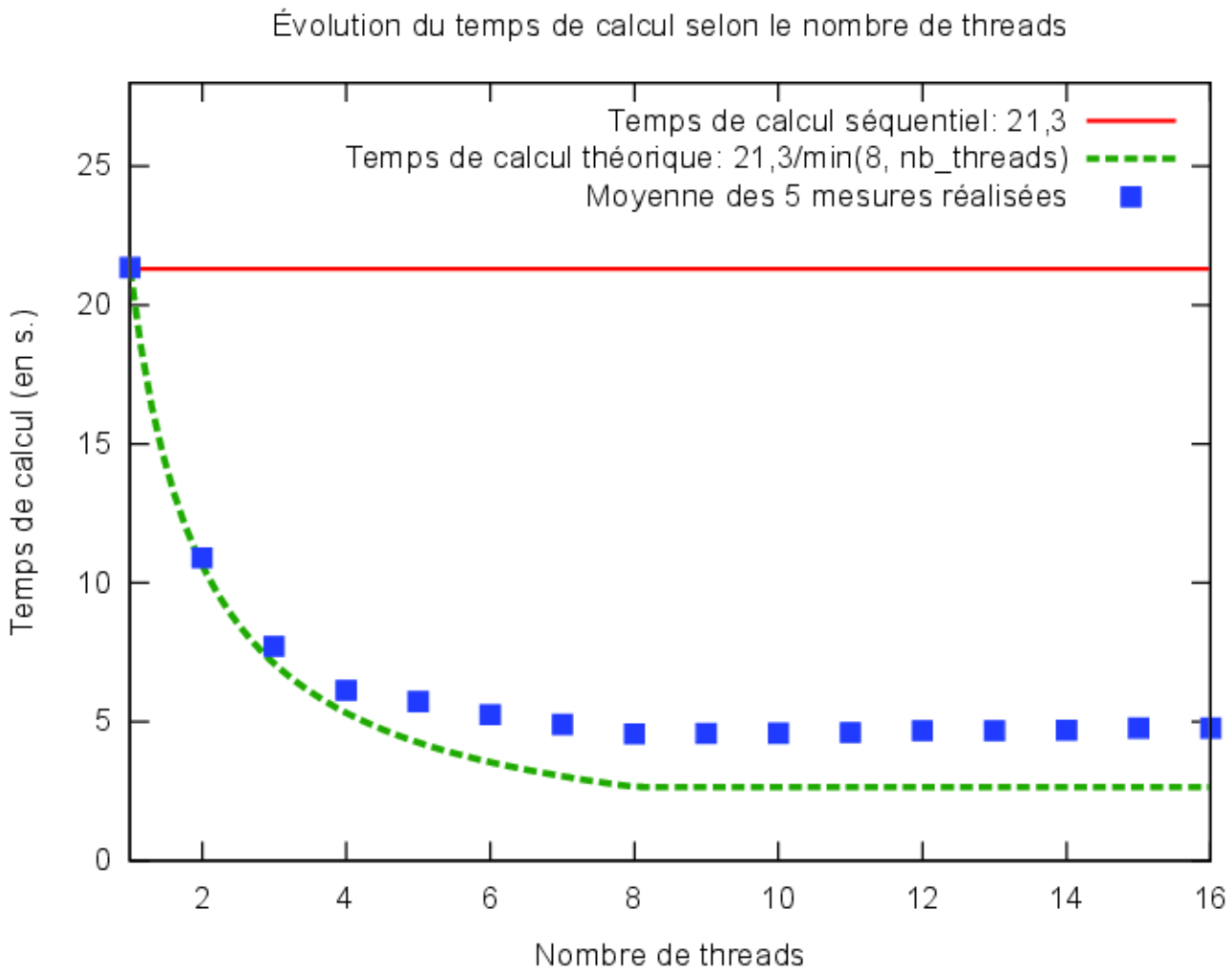
Le programme Java annonce 8 processeurs disponibles. Néanmoins, le système indique :

```
$ sysctl hw.physicalcpu
hw.physicalcpu: 4
$ sysctl hw.logicalcpu
hw.logicalcpu: 8
```

Il y a donc en réalité seulement 4 coeurs, même si on peut espérer que chaque coeur puisse accueillir deux threads simultanément par ≪ hyperthreading ≫.

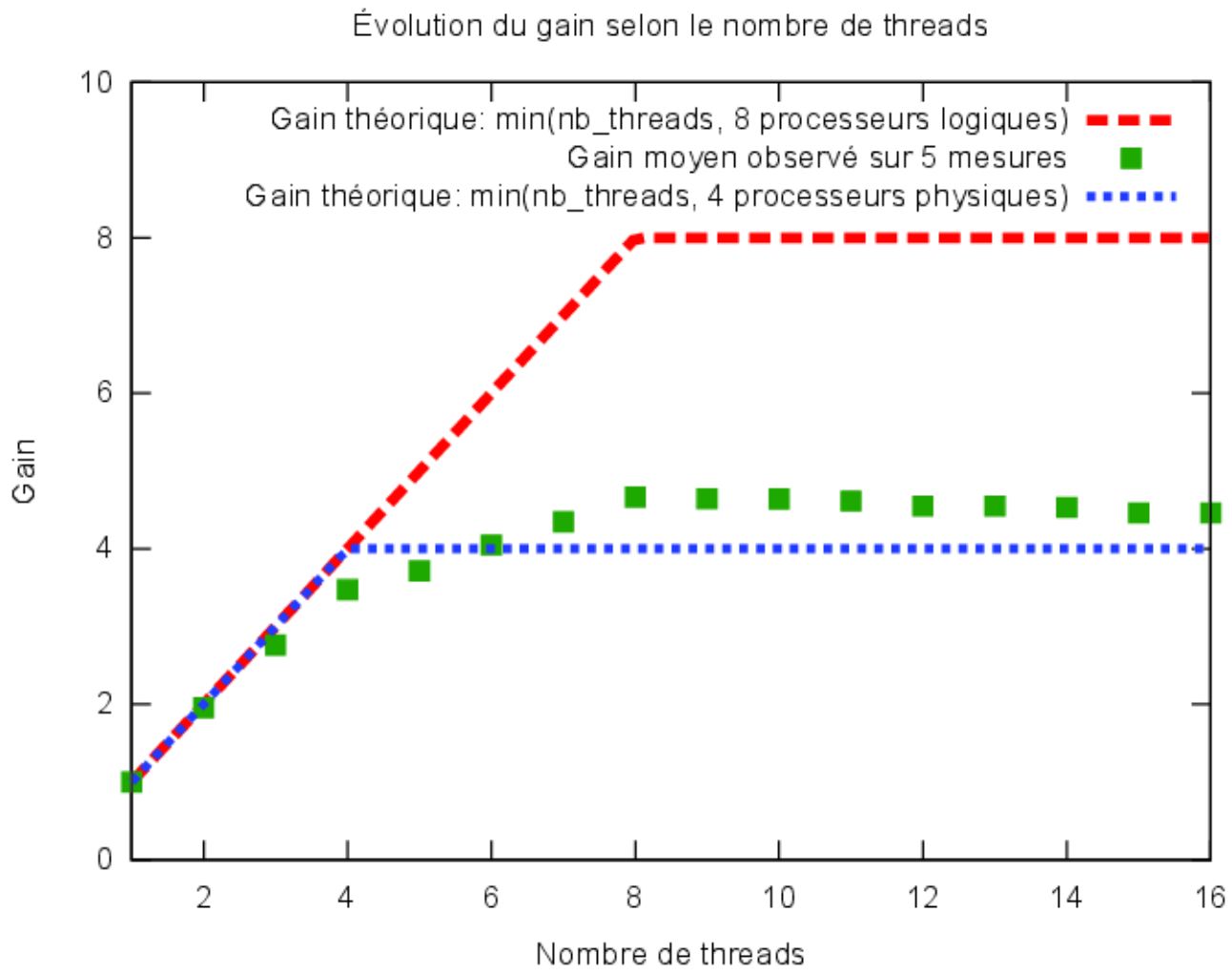
Ce qui conduit à 8 processeurs annoncés par Java.

Mesures effectuées : de 1 à 16 threads



Avec 8 processeurs logiques, on peut espérer diviser le temps de calcul par 8. Mais ce n'est pas vraiment le cas...

Évolution du gain selon le nombre de threads



L'hyperthreading permet effectivement un gain un peu supérieur à 4, le nombre de processeurs physiques, mais bien loin du gain de 8 espéré.

Outils pratiques pour le multitâche

Master Informatique — Semestre 1 — UE obligatoire de 3 crédits

Année 2018-2019

Version du 18 novembre 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 simplement utiliser les bons outils, correctement.

Rappel : les threads héritent de `Thread`

```
public class monThread extends Thread {
  public void run(){
    for (int i = 1; i<=1000; i++)
      System.out.println(i);
  }
}
public class Exemple {
  public static void main(String[] args){
    Thread t = new monThread();
    t.start();
  }
}
```

mais il faut les démarrer !

Rappel : les tâches implémentent `Runnable`

```
public class monRunnable implements Runnable {
  public void run(){
    for (int i = 1; i<=1000; i++)
      System.out.println(i);
  }
}
public class Exemple {
  public static void main(String[] args) {
    Thread t = new Thread( new monRunnable() );
    t.start();
  }
}
```

mais elles nécessitent un thread pour tourner !

Les threadpools : l'idée principale

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

Approche apparemment simple : créer un thread pour chaque nouvelle tâche.

- ⇝ Facile et correct. C'est bien pour commencer...
- ⇝ Peu devenir un peu compliqué (voire peu performant) s'il y a beaucoup de tâches et donc beaucoup de threads à gérer ; car la gestion d'un thread par la JVM consomme du temps et de la mémoire. Le surcoût peut en effet être non négligeable, en particulier *si la tâche est courte* !

Approche plus professionnelle : utiliser un *réservoir de threads* (en anglais, on dit ≪ thread pool ≫) qui traite dynamiquement l'ensemble des tâches soumises.

Threadpool : les grands principes

① Il faut d'abord **installer un réservoir** de threads.

② Il suffit ensuite de **soumettre au réservoir** les tâches à exécuter.

— Chaque thread exécutera plusieurs tâches, les unes après les autres ;

— On ne sait pas quel thread exécutera une tâche soumise ;

— On évite ainsi de créer un thread à chaque nouvelle tâche :

⇝ Le délai d'exécution d'une tâche sera éventuellement réduit, car le thread qui l'exécute est déjà créé ;

⇝ 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.

Ceci dit :

Le gain de performance est secondaire ici

Quelques difficultés pratiques à cette approche

① Il faut trouver, souvent de manière empirique, la **bonne taille** pour le réservoir de threads, en fonction du nombre de processeurs (physiques et logiques) disponibles sur la machine et du type de tâches à exécuter (urgentes ou non).

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

⇝ 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 ».

⇝ Il y a alors **un risque accrû d'interblocage**, s'il n'y a plus aucun thread disponible pour exécuter la tâche qui, une fois terminée, aurait débloqué toutes les autres...

- ✔ *Rappel : Thread vs. Runnable*
- ✔ *Maître, esclaves et tâches à exécuter*
- ☛ *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.

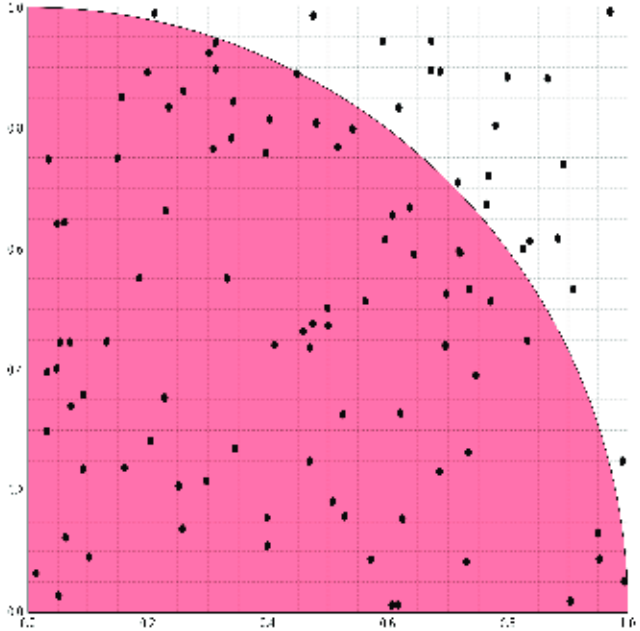
On ne peut pas savoir a priori quand une tâche sera terminée : il faudra donc *détecter la terminaison*.

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

```
class Tirages implements Runnable {
  final long nbTirages;                 // à réaliser par chaque tâche
  long tiragesDansLeDisque = 0 ;        // obtenus par chaque tâche

  Tirages(long nbTirages){
    this.nbTirages = nbTirages;
  }

  public void run(){
    double x, y;
    for (long 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 = alea.nextDouble();
```

du programme séquentiel (et parallèle) par

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

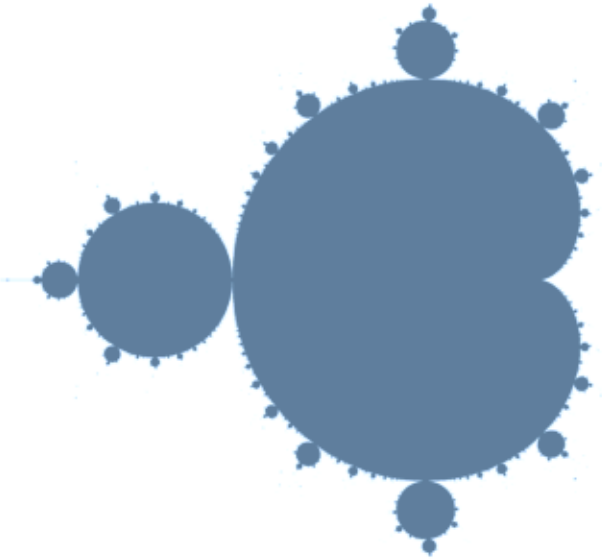
Exemple : estimation de $\pi/4$ par Monte-Carlo (2/2)

```
ExecutorService executeur = Executors.newFixedThreadPool(10) ;
Tirages[] mesTaches = new Tirages[10] ;
for (int j = 0; j < 10 ; j++) {
  mesTaches[j] = new Tirages( nbTirages/10 ) ;
  executeur.execute(mesTaches[j]) ;
}
executeur.shutdown();    // Il n'y a plus aucune tâche à soumettre
while (! executeur.awaitTermination(1, TimeUnit.SECONDS)) {
  System.out.print("#") ;
}                        // Toutes les tâches ont été exécutées
System.out.println() ;
long tiragesDansLeDisque = 0 ;
for (int j = 0; j < 10 ; j++) {
  tiragesDansLeDisque += mesTaches[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, soit 500 tâches. La valeur 10 devra donc être remplacée par 4 ou 500, selon le contexte.

- ✔ *Rappel : Thread vs. Runnable*
- ✔ *Maître, esclaves et tâches à exécuter*
- ✔ *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 Tirages implements Callable<Long>{
  final long nbTirages ;
  long tiragesDansLeDisque = 0 ;
  Tirages(long nbTirages){
    this.nbTirages = nbTirages;
  }

  public Long call(){
    for (long i = 0; i < nbTirages; i++) {
      double x = ThreadLocalRandom.current().nextDouble(1);
      double 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 soumet à un **`Executor`** une tâche particulière **qui renvoie un résultat**, on ne peut pas savoir quand la tâche sera exécutée ni quand précisément le résultat renvoyé par 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>> mesPromesses=new ArrayList<Future<Long>>();
for (int j = 0; j < 10 ; j++){
  Tirages tache = new Tirages( nbTirages/10 ) ;
  Future<Long> promesse = executeur.submit(tache) ;
  mesPromesses.add(promesse) ;    // On stocke les promesses
}
long tiragesDansLeDisque = 0 ;
for (int j = 0; j < 10 ; j++){
  Future<Long> promesse = mesPromesses.get(j);     // Non bloquant
  tiragesDansLeDisque += promesse.get();           // Bloquant !
}
double resultat = (double) tiragesDansLeDisque / nbTirages ;

executeur.shutdown() ; // Il n'y a plus aucune tâche à soumettre
```

- ✔ *Rappel : Thread vs. Runnable*
- ✔ *Maître, esclaves et tâches à exécuter*
- ✔ *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 particulières ;

$\rightsquigarrow$ *si on ne sait pas utiliser les Futures...*

— Guetter les fins des tâches asynchrones soumises à l'aide de **`Futures`** ;

$\rightsquigarrow$ *c'est beaucoup plus professionnel...*

— Utiliser un **service de complétion** : c'est souvent le plus simple !

$\rightsquigarrow$ *c'est aussi le plus élégant !*

Utiliser un service de completion

```
ExecutorService executeur = Executors.newFixedThreadPool(10) ;
CompletionService<Long> ecs =
                    new ExecutorCompletionService<Long>(executeur) ;

for (int j = 0; j < 10 ; j++) {
  ecs.submit( new Tirages( nbTirages/10 ) );
}

int tiragesDansLeDisque = 0;
for (int j = 0; j < 10; j++) {
  tiragesDansLeDisque += ecs.take().get(); // take() est bloquant
}

double resultat = (double) tiragesDansLeDisque / nbTirages ;
```

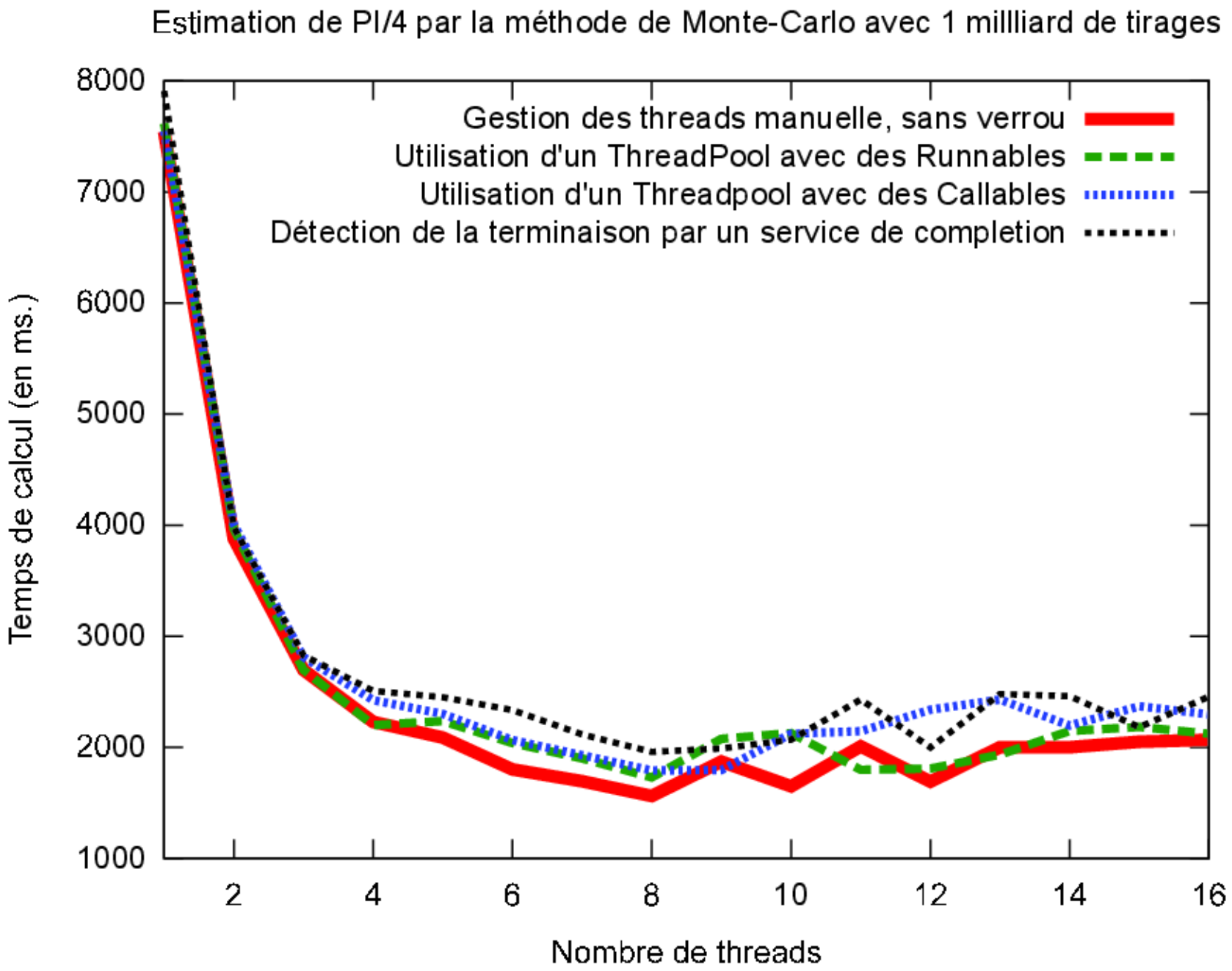
En décomposant pour mieux comprendre

```
ExecutorService executeur = Executors.newFixedThreadPool(10) ;
CompletionService<Long> ecs =
                    new ExecutorCompletionService<Long>(executeur) ;

for (int j = 0; j < 10 ; j++) {
  Tirages tache = new Tirages( nbTirages/10 );
  ecs.submit(tache);
}

int tiragesDansLeDisque = 0;
for (int j = 0; j < 10; j++) {
  Future<Long> promesse = ecs.take();        // Bloquant !
  Long resultatAttendu = promesse.get();     // Non bloquant !
  tiragesDansLeDisque += resultatAttendu;    // Ordre indéterminé
}
double resultat = (double) tiragesDansLeDisque / nbTirages ;
```

À propos des performances (sur une machine à 8 proc. logiques)



Les trois types de collections en Java

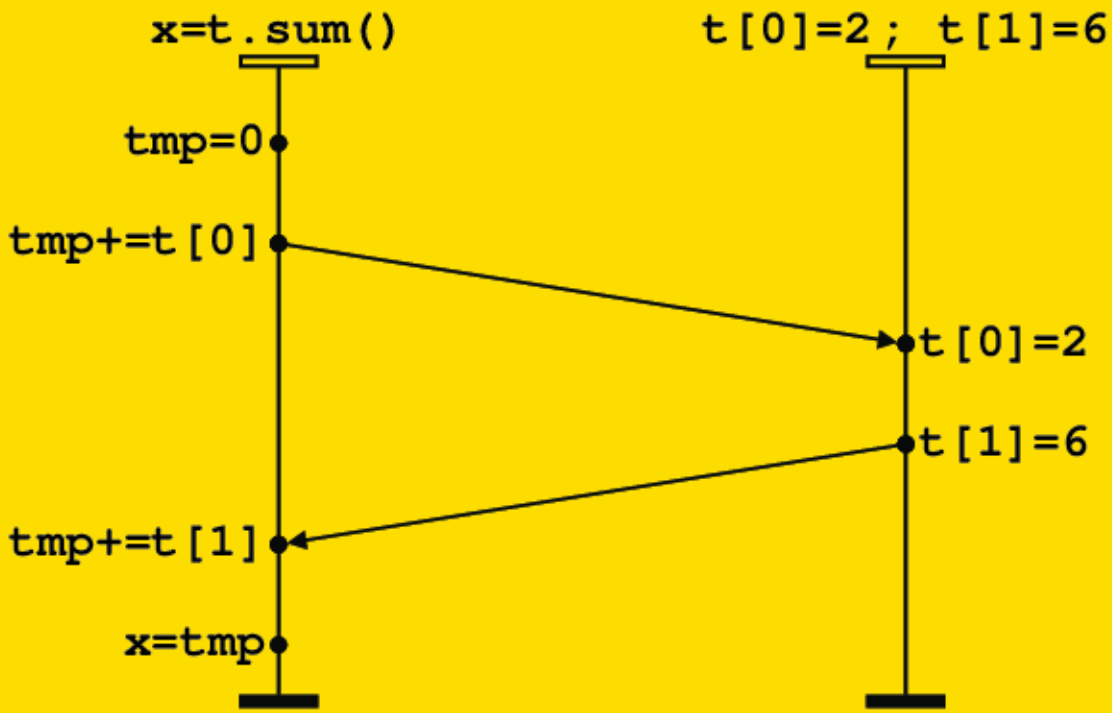
Master Informatique — Semestre 1 — UE obligatoire de 3 crédits

Année 2018-2019

Version du 18 novembre 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.

Un problème d'atomicité

Utiliser une collection concurrente ou synchronisée ne garantit pas de produire un code correct, car **la composition d'instructions atomiques ne forme quasiment jamais une suite d'instructions atomique.**

```
while(! maListe.isEmpty() ) {
  Element e = maListe.remove(0);
  System.out.println(e.m);
}
```

n'est pas thread-safe !

Même dans le cas où **`maListe`** est une instance d'une classe concurrente ou synchronisée, ce code pourra lever une exception *si la liste est vidée entre le moment du test dans la boucle* ***`while`*** *et la tentative de retrait de son premier élément.*

Les collections synchronisées

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`** ≫ : le verrou intrinsèque de la collection est requis pour chaque opération sur cette collection.

Autres constructions du même genre :

```
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)
```

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.

Itérateurs de collections synchronisées

Les itérateurs d'une classe synchronisée ne sont pas synchronisés : ils sont même au contraire ≪ *fail-fast* ≫ car 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 ≪ inadéquate ≫ appliquée au cours de l'itération elle-même.

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.* ≫

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 sur une ArrayList

  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));
}
```

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*

Motivations

Les collections « concurrentes » autorisent souvent plusieurs modifications des données en parallèle, de même que des lectures en même temps que des modifications.

Si ces accès constituent l'essentiel des opérations réalisées, utiliser une collection concurrente plutôt qu'une collection synchronisée induira un **gain de performance.**

Les nouvelles collections concurrentes

Ces « nouvelles » collections possèdent des propriétés communes :

1. Il est (presque toujours) interdit de stocker **`null`**.
2. Ces collections fonctionnent indépendamment du verrou intrinsèque de ses objets : contrairement aux collections dites synchronisées, tenter de 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.

- ✔ *Les collections synchronisées*
- ✔ *Les collections concurrentes*
- ☛ *La classe* **`ConcurrentHashMap`**

La classe `ConcurrentHashMap`

Les méthodes de **`ConcurrentHashMap`** sont à peu près les mêmes que celles disponibles avec **`HashTable`**.

Néanmoins

- Au contraire de **`HashTable`**, les lectures d'un objet ne bloquent ni les autres lectures *ni même les mises à jour*.
- Une lecture reflète l'état de la table après la dernière mise à jour complètement terminée (celles en cours ne sont pas prises en compte).

A propos de ConcurrentHashMap$< K, V >$

A hash table supporting full concurrency of retrievals and adjustable expected concurrency for updates. This class obeys the same functional specification as Hashtable, and includes versions of methods corresponding to each method of Hashtable. However, even though all operations are thread-safe, retrieval operations do not entail locking, and there is not any support for locking the entire table in a way that prevents all access. This class is fully interoperable with Hashtable in programs that rely on its thread safety but not on its synchronization details.

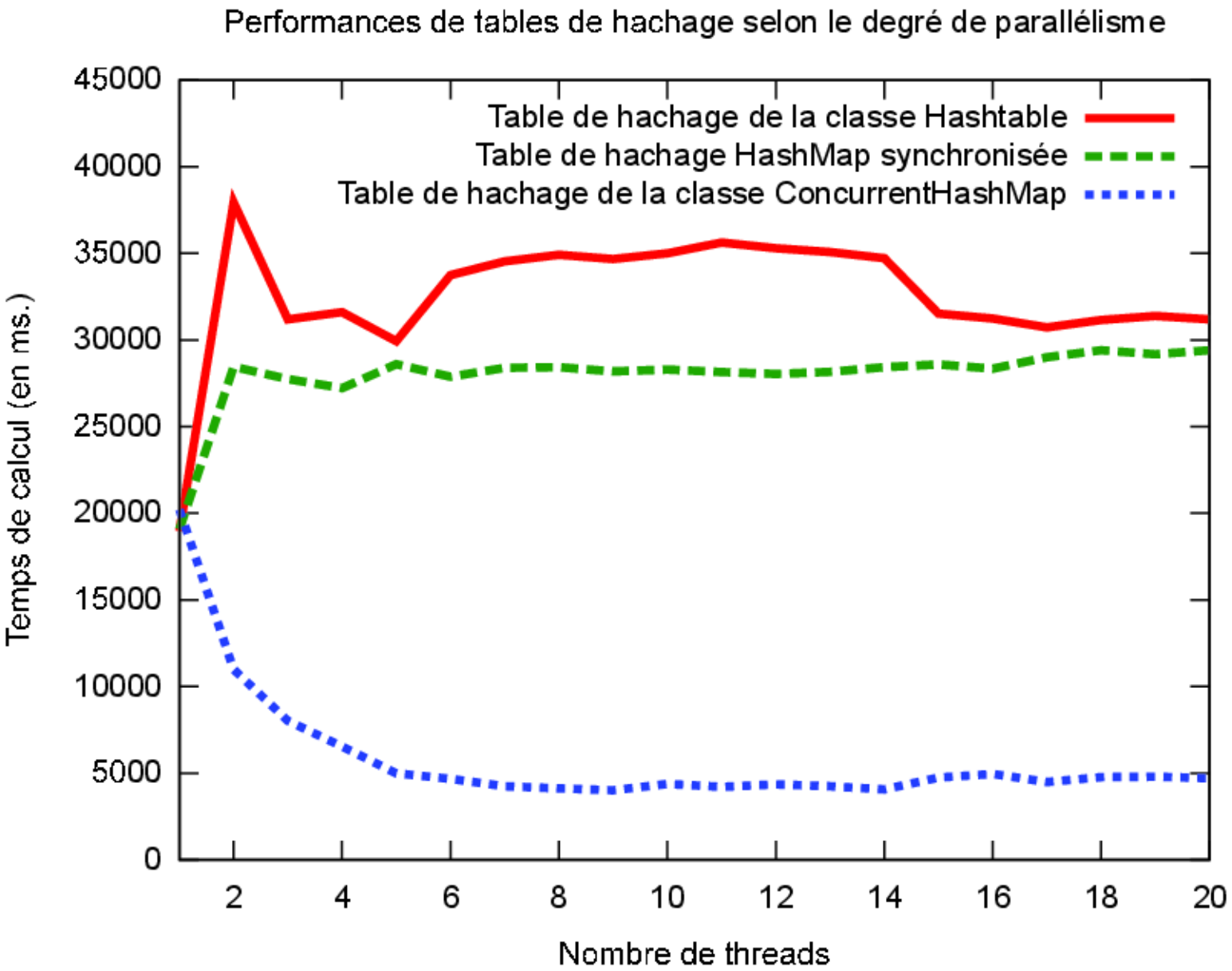
Retrieval operations (including get) generally do not block, so may overlap with update operations (including put and remove). Retrievals reflect the results of the most recently completed update operations holding upon their onset. ... However, iterators are designed to be used by only one thread at a time.

The allowed concurrency among update operations is guided by the optional concurrencyLevel constructor argument (default 16), which is used as a hint for internal sizing. The table is internally partitioned to try to permit the indicated number of concurrent updates without contention.

Comparaison de trois tables de hachage : benchmark

```
// table=new Hashtable<String, Integer>();
table=Collections.synchronizedMap(new HashMap<String, Integer>());
// table=new ConcurrentHashMap<String, Integer>();
...
public void run(){
  for (int i = 0; i < part; i++) {
    // On choisit une clef au hasard
    Integer clef = alea.nextInt(0,1_000_000);
    // On cherche la valeur correspondant à la clef
    Integer valeur = table.get(String.valueOf(clef));
    // On ajoute une entrée avec cette clef dans la table
    table.put(String.valueOf(clef), clef);
  }
}
```

Comparaison de trois tables de hachage



- ✔ *Les collections synchronisées*
- ✔ *Les collections concurrentes*
- ✔ *La classe* **`ConcurrentHashMap`**
- ☛ *La classe* **`CopyOnWriteArrayList`**

La classe `CopyOnWriteArrayList`

La classe **`CopyOnWriteArrayList`** est une variante « thread-safe » de **`ArrayList`** dans laquelle toutes les opérations de mise à jour sont effectuées en faisant une **copie** du tableau sous-jacent (le tableau n'est jamais modifié).

Cette variante n'est intéressante que lorsque les parcours de la liste sont beaucoup plus fréquents que les mises à jour. La Javadoc explique :

« *This is ordinarily too costly, but may be more efficient than alternatives when traversal operations vastly outnumber mutations, and is useful when you cannot or don't want to synchronize traversals, yet need to preclude interference among concurrent threads.* »

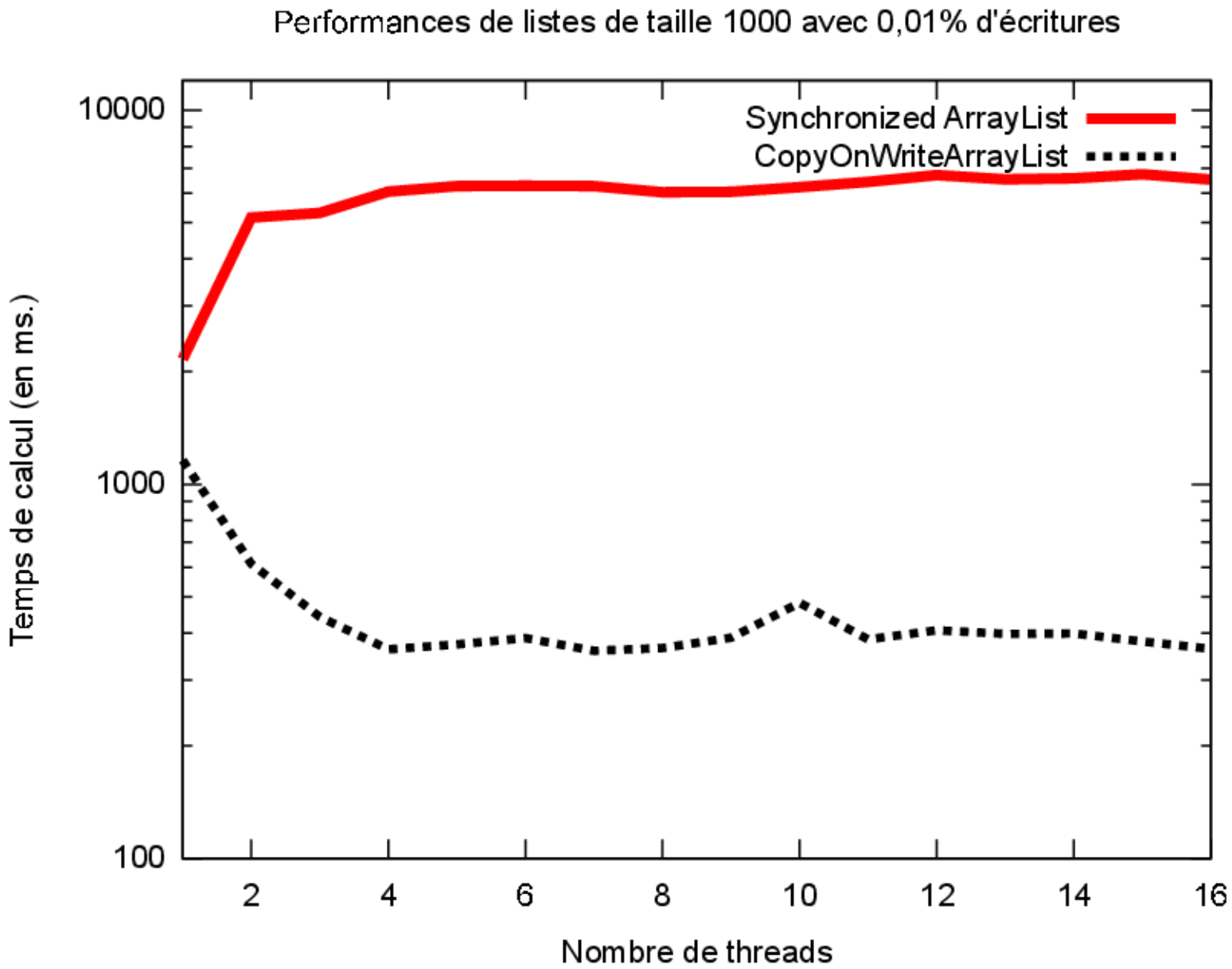
Les itérateurs reflètent la liste au moment où ils ont été créés ; **ils ne peuvent modifier la liste**. En effet, la Javadoc dit :

« *Element-changing operations on iterators themselves (remove, set, and add) are not supported. These methods throw* ***`UnsupportedOperationException`***. »

Comparaison de deux listes : benchmark

```
static List<Integer> liste ;
static int taille = 1000                    // Taille constante de la liste
static int rapportLectureEcriture = 10_000;
...
public void run(){
  for (int i = 0; i < part; i++) {
    int index = alea.nextInt(taille) ;
    if (alea.nextInt(rapportLectureEcriture) < 1) {
      Integer valeur = alea.nextInt(valeurMaximale) ;
      liste.add(index, valeur) ;    // Écriture 1 fois sur 10_000
    } else {
      liste.get(index) ;            // Lecture le reste du temps
    }
  }
}
```

Comparaison de 2 listes, selon le nombre de threads



Verrous, barrières et loquets

Master Informatique — Semestre 1 — UE obligatoire de 3 crédits

Année 2018-2019

Version du 18 novembre 2018

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).

— 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 !

Fil rouge classique : le compte bancaire (sous la forme d'un moniteur)

```
public class CompteBancaire {
  private volatile long épargne;

  public CompteBancaire(long épargne) {
    this.épargne = épargne;
  }
  public synchronized void déposer(long montant) {
    épargne += montant;
  }
  public synchronized void retirer(long montant) {
    épargne -= montant;
  }
  public synchronized long solde() {
    return épargne;
  }
}
```

Alternative avec un ReentrantLock (1/2)

```
public class CompteBancaire {
  private volatile long épargne;
  private final Lock verrou = new ReentrantLock();

  public CompteBancaire (long épargne) {
    this.épargne = épargne;
  }

  public void déposer(long montant) {
    verrou.lock();
    try {
      épargne += montant;
    } finally {
      verrou.unlock();
    }
  }
```

Alternative avec un ReentrantLock (2/2)

```
  public void retirer(long montant) {
    verrou.lock();
    try {
      épargne -= montant;
    } finally {
      verrou.unlock();
    }
  }
  public long solde() {
    verrou.lock();
    try {
      return épargne;
    } finally {
      verrou.unlock();
    }
  }
}
```

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.

Attention ! L'emploi d'un verrou équitable entraînera assez souvent une *dégradation des performances*, car il induit de nombreux changements de contextes supplémentaires.

Blanche-Neige équitable

```
class BlancheNeige {      // à l'aide d'un verrou lock équitable
  private final ReentrantLock verrou = new ReentrantLock(true);
  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();
  }
}
```

Blanche-Neige équitable : test.

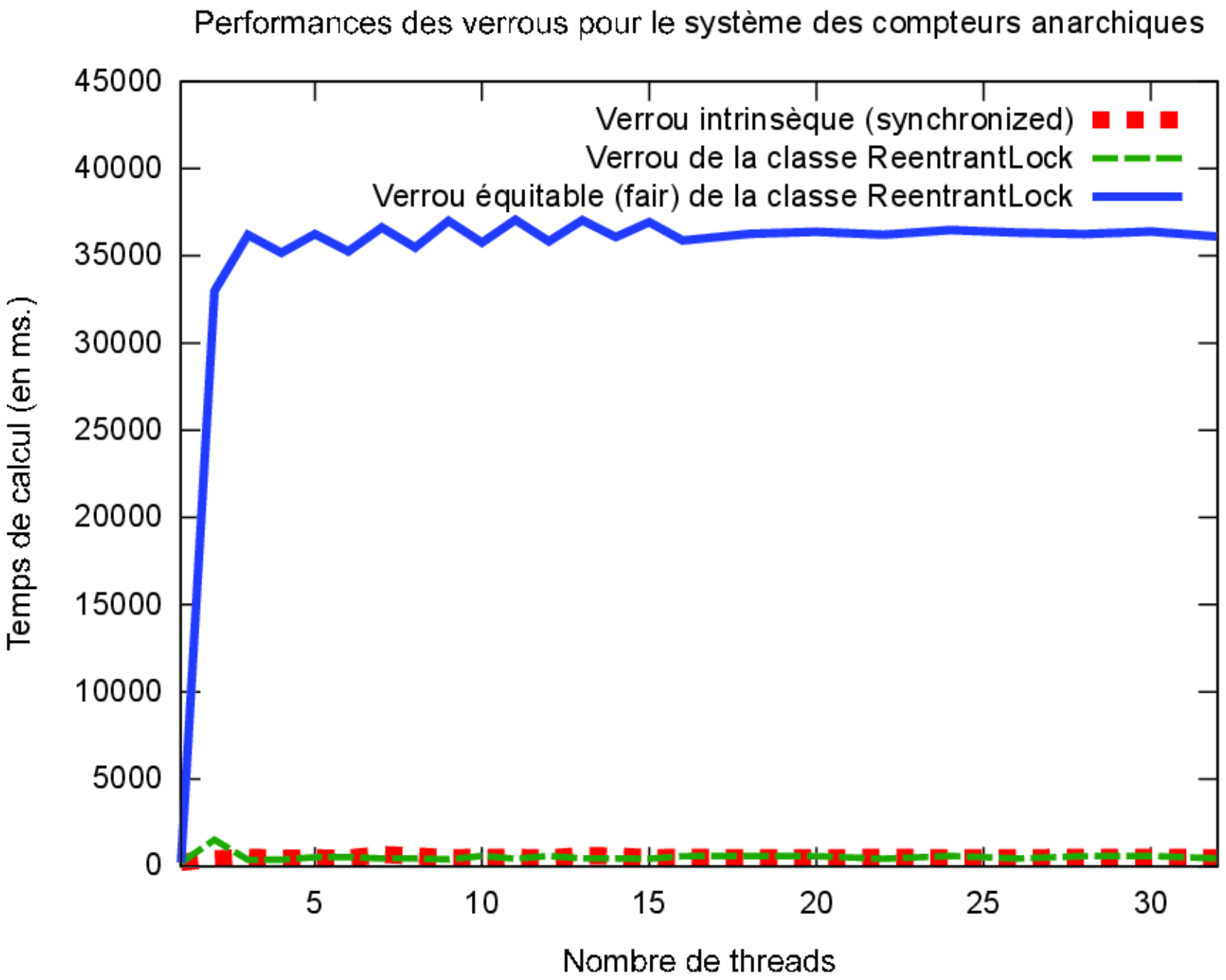
```
$ javac SeptNains.java
$ java SeptNains
Simplet veut la ressource
Timide veut la ressource
Prof veut la ressource
Atchoum veut la ressource
Dormeur veut la ressource
Grincheux veut la ressource
Joyeux veut la ressource
Simplet accède à la ressource
Simplet relâche la ressource
Simplet veut la ressource
Prof accède à la ressource
Prof relâche la ressource
Prof veut la ressource
Timide accède à la ressource
Timide relâche la ressource
Timide veut la ressource
```

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 ?

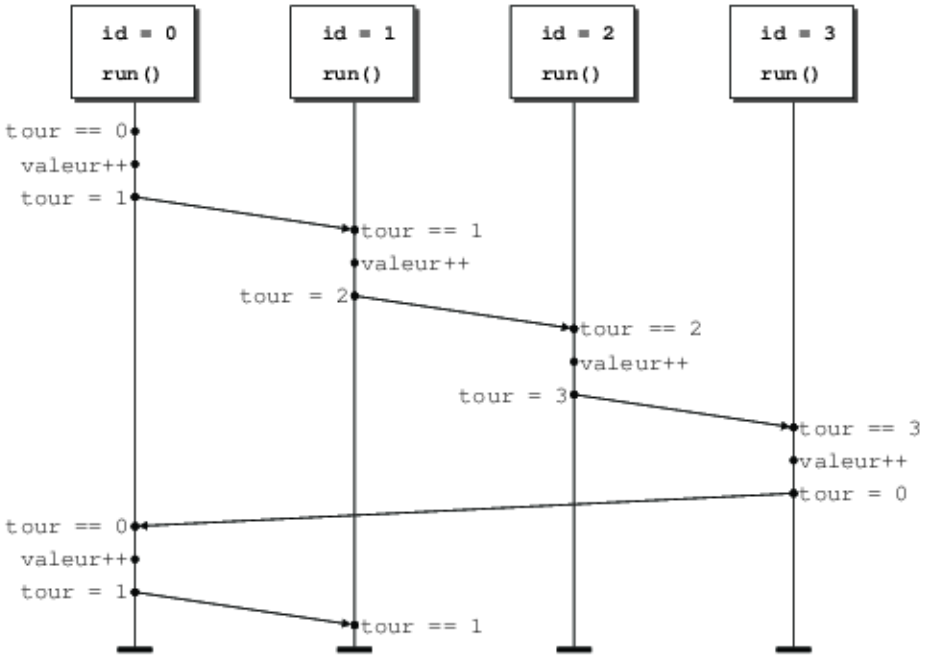
— Les appels à un verrou équitable prennent plus de temps, car il faut gérer l'équité.

— 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.

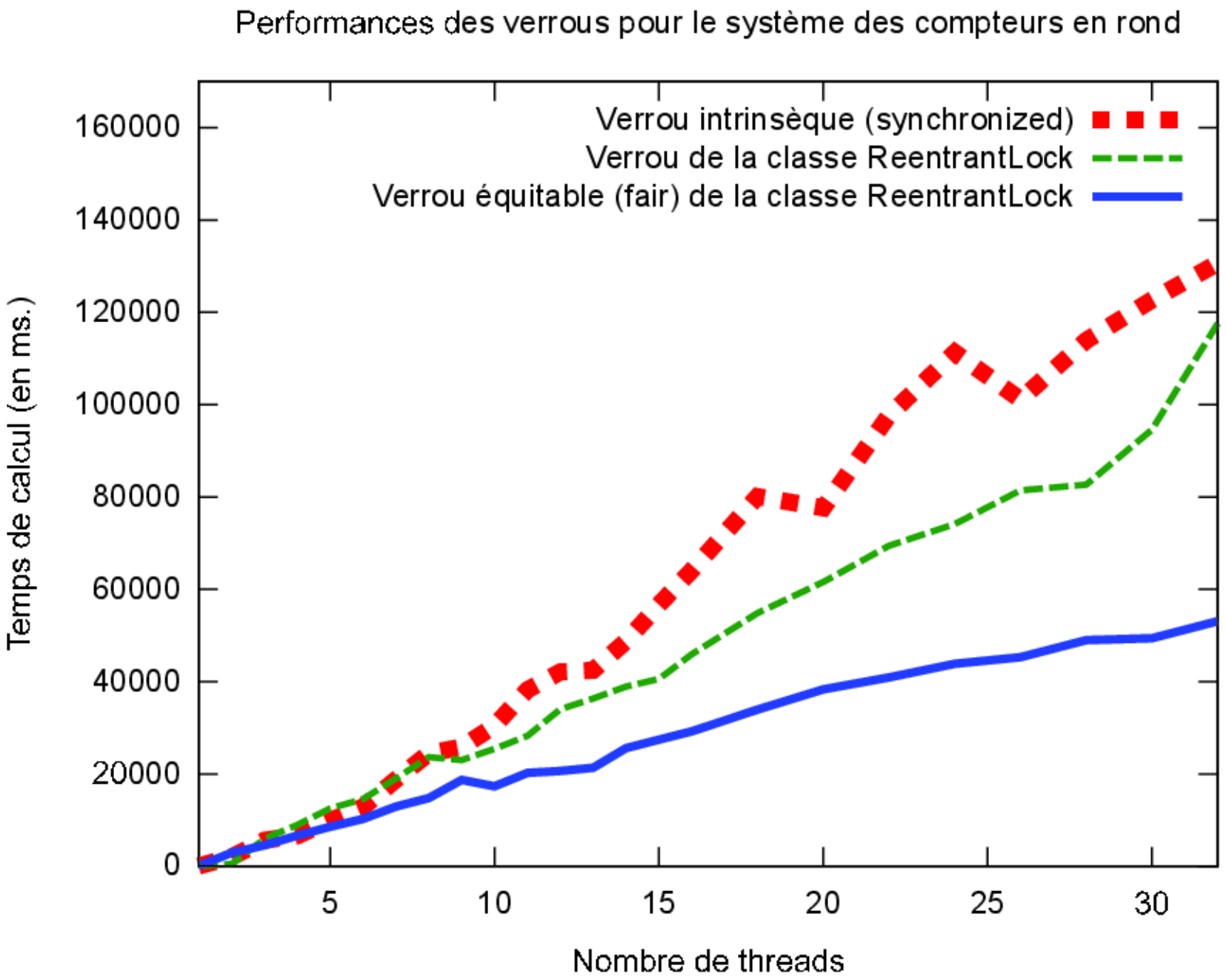
— Autre chose ?

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)



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`**.

Attention ! **`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 !

- ✔ *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 !

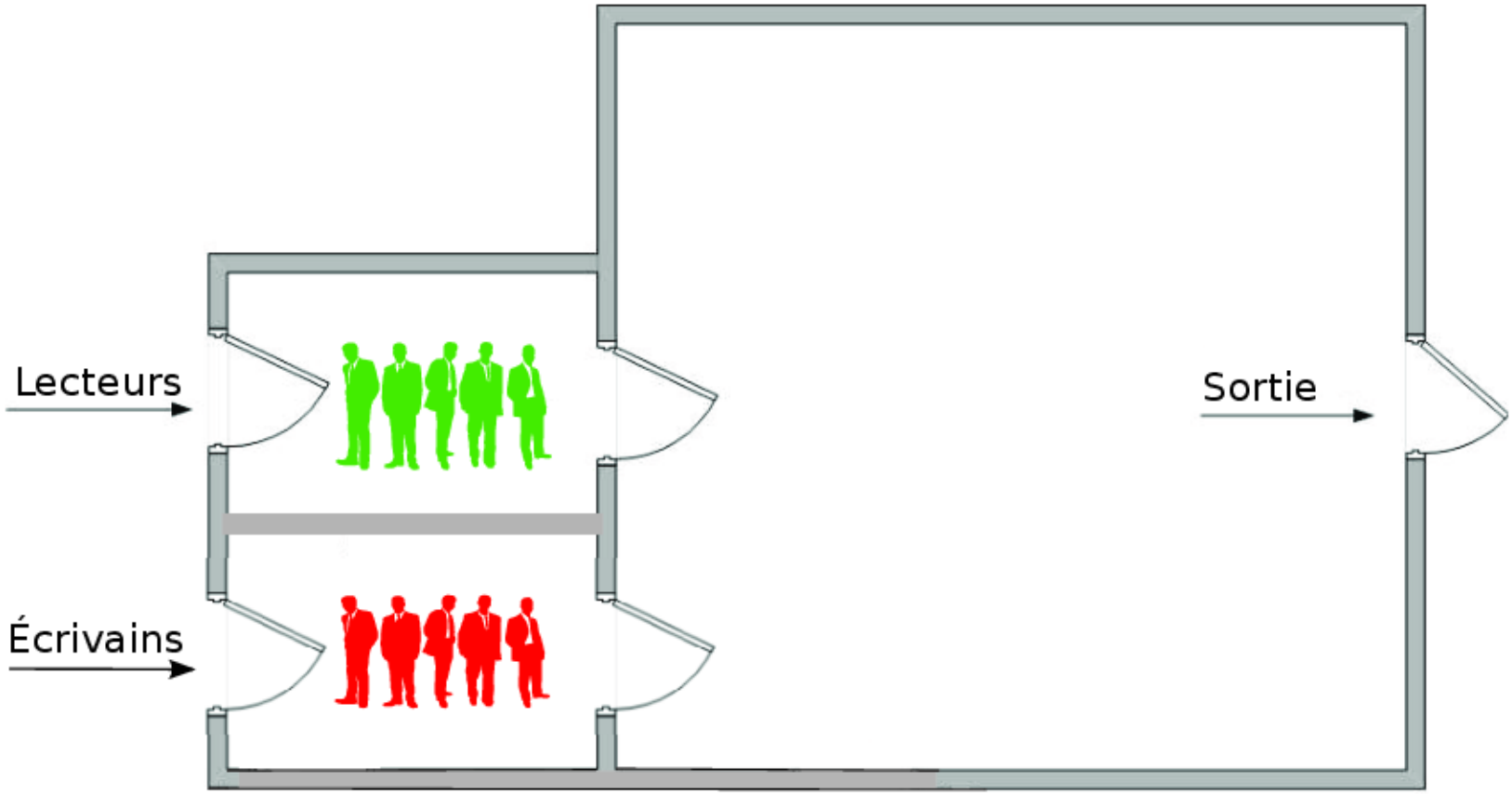
Mais ceci interdit d'effectuer *des lectures en parallèle*, qui pourtant ne posent a priori aucun problème (ni en terme d'atomicité, ni sur le plan matériel).

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 ») qui devra être requis par chaque lecture et un **verrou en écriture** (appelé « write-lock ») qui devra être requis pour chaque écriture.

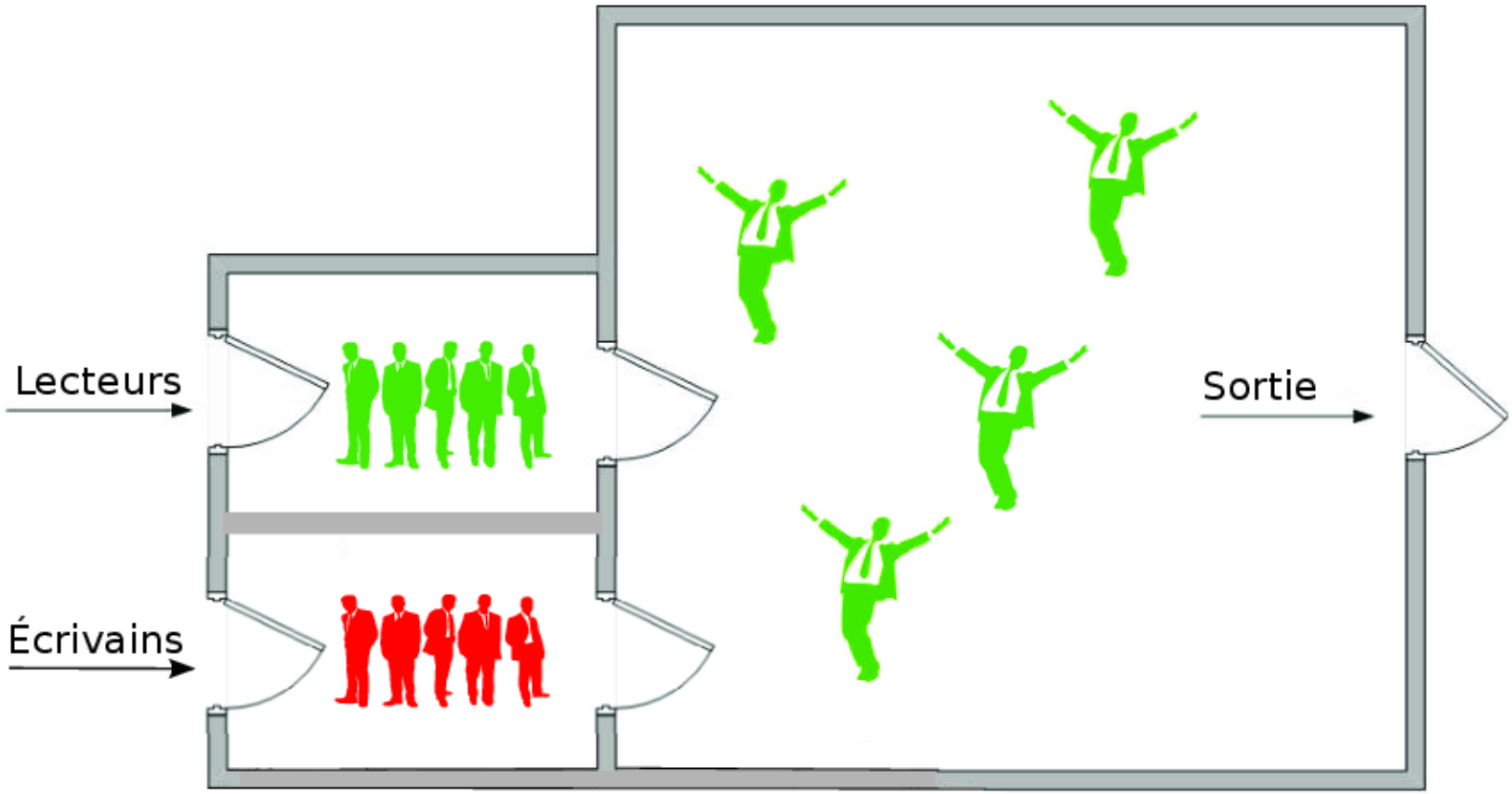
Structure d'un verrou de Lecture/Ecriture



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

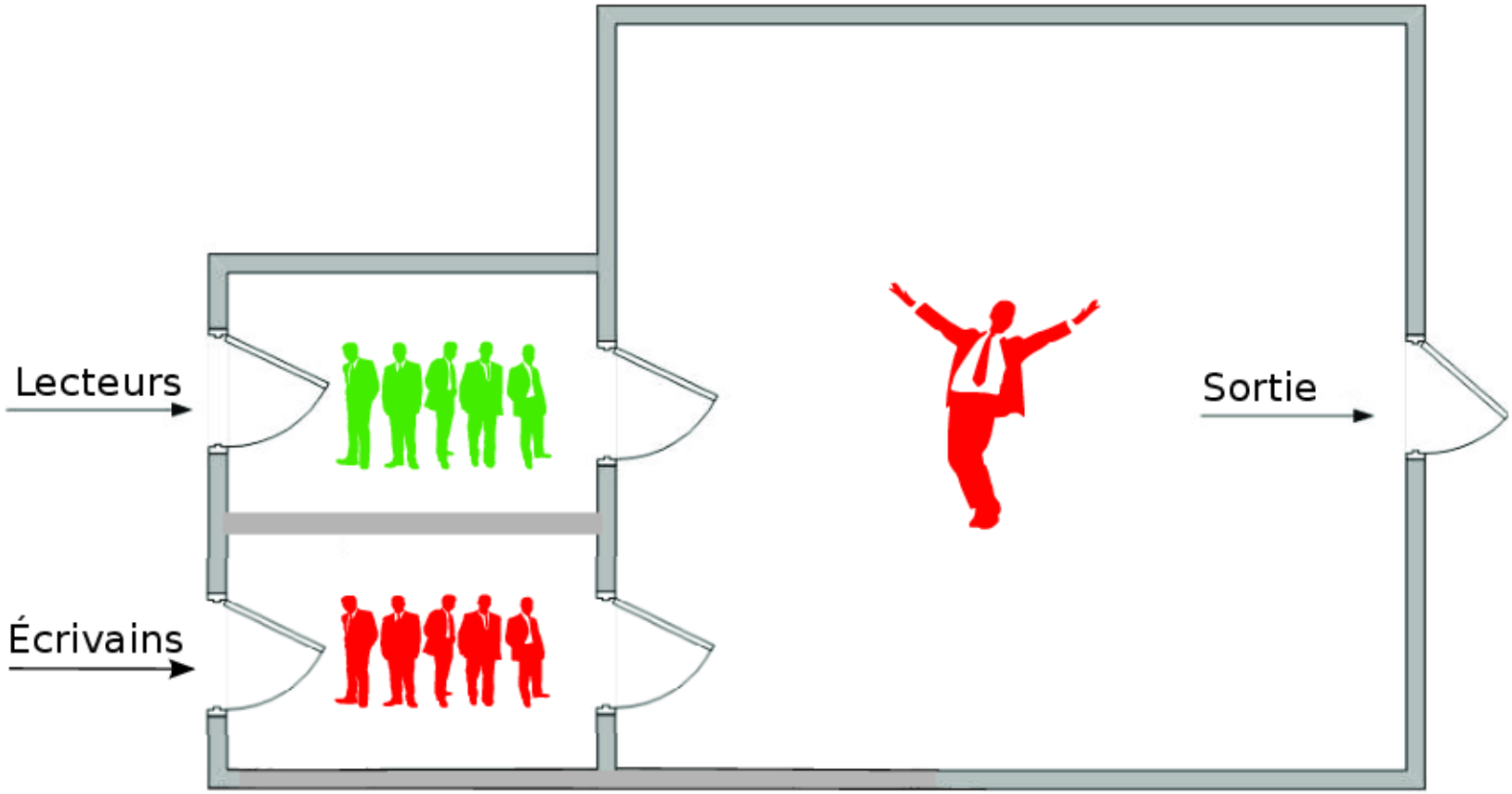
- Les lecteurs attendent le verrou de lecture ;
- Les écrivains attendent le verrou d'écriture.

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 !

Compte bancaire avec un ReentrantReadWriteLock (1/2)

```
public class CompteBancaire {
  private volatile long épargne;
  private final ReadWriteLock verrou=new ReentrantReadWriteLock();

  public CompteBancaire(long épargne) {
    this.épargne = épargne;
  }

  public void déposer(long montant) {
    verrou.writeLock().lock();
    try {
      épargne += montant;
    } finally {
      verrou.writeLock().unlock();
    }
  }
```

Compte bancaire avec un ReentrantReadWriteLock (2/2)

```
  public void retirer(long montant) {
    verrou.writeLock().lock();
    try {
      épargne -= montant;
    } finally {
      verrou.writeLock().unlock();
    }
  }
  public long solde() {
    verrou.readLock().lock();
    try {
      return épargne;
    } finally {
      verrou.readLock().unlock();
    }
  }
}
```

- ✔ *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()`**.

Ce qu'il faut retenir

Java dispose de nombreux outils dédiés à la programmation multithread. Seuls quelques uns des plus importants ont été abordés ici.

L'emploi d'un « *threadpool* » permet de développer un code plus simple et plus élégant qui laisse à la JVM le soin de gérer les threads en charge d'exécuter la liste des tâches.

Il existe *trois catégories de collections* utilisables dans un contexte multithread ; chacune possède ses propres spécificités qui influent sur la manière de les employer. La documentation permet de connaître les usages corrects d'une collection donnée.

Outre les verrous intrinsèques des objets, il existe *d'autres verrous plus souples d'utilisation* et qui, parfois, offrent de meilleures performances. Nous verrons lors du prochain cours un *nouveau type de verrou*, le « stamped lock », qui produit assez souvent des résultats meilleurs que ceux observés avec les verrous de lecture-écriture.