La programmation optimiste

Master Informatique — Semestre 2 — UE obligatoire de 3 crédits

Année 2017-2018

Version du 1er février 2018

La programmation optimiste

La programmation à l'aide de verrous repose sur une perspective pessimiste de l'exécution des tâches concurrentes du système.

Il faut se méfier des variables ou des objets partagés !

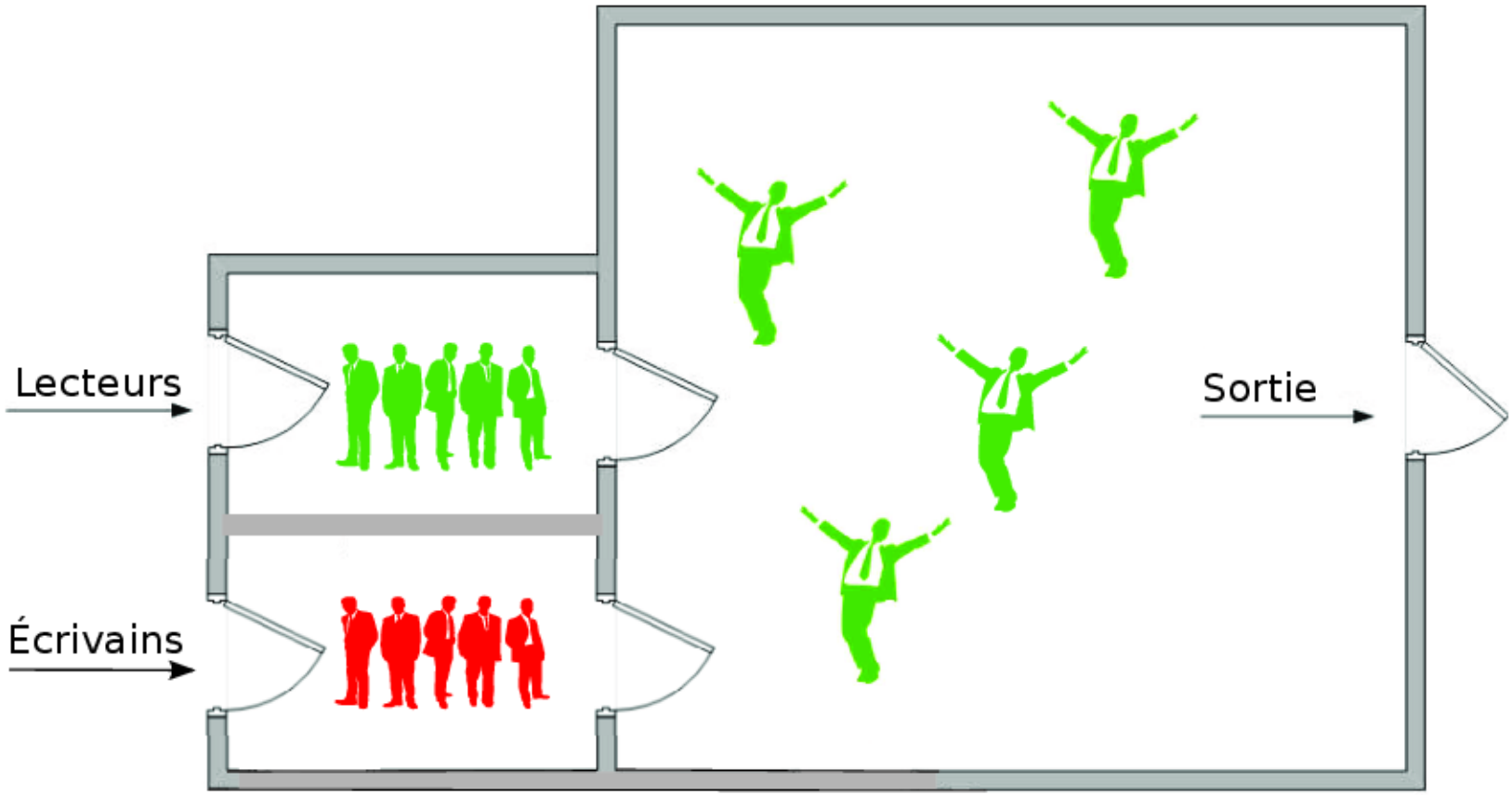
À l'inverse de l'approche classique, la **programmation optimiste** consiste à ne pas prendre de trop de précautions *a priori* mais à vérifier *a posteriori* qu'aucune interférence préjudiciable n'a eu lieu au cours de l'exécution de la « section critique. »

— Pour les **opérations de lecture**, il suffit de recommencer la procédure si elle a échoué (quitte à prendre alors d'avantage de précautions), *car les données obtenues sont potentiellement corrompues* ;

— Pour les **opérations d'écriture**, en revanche, il faut parvenir à effectuer une modification complète et correcte des données ou bien aucune modification du tout, quitte à devoir retenter l'opération ultérieurement : on parle alors d'*écriture atomique conditionnelle*.

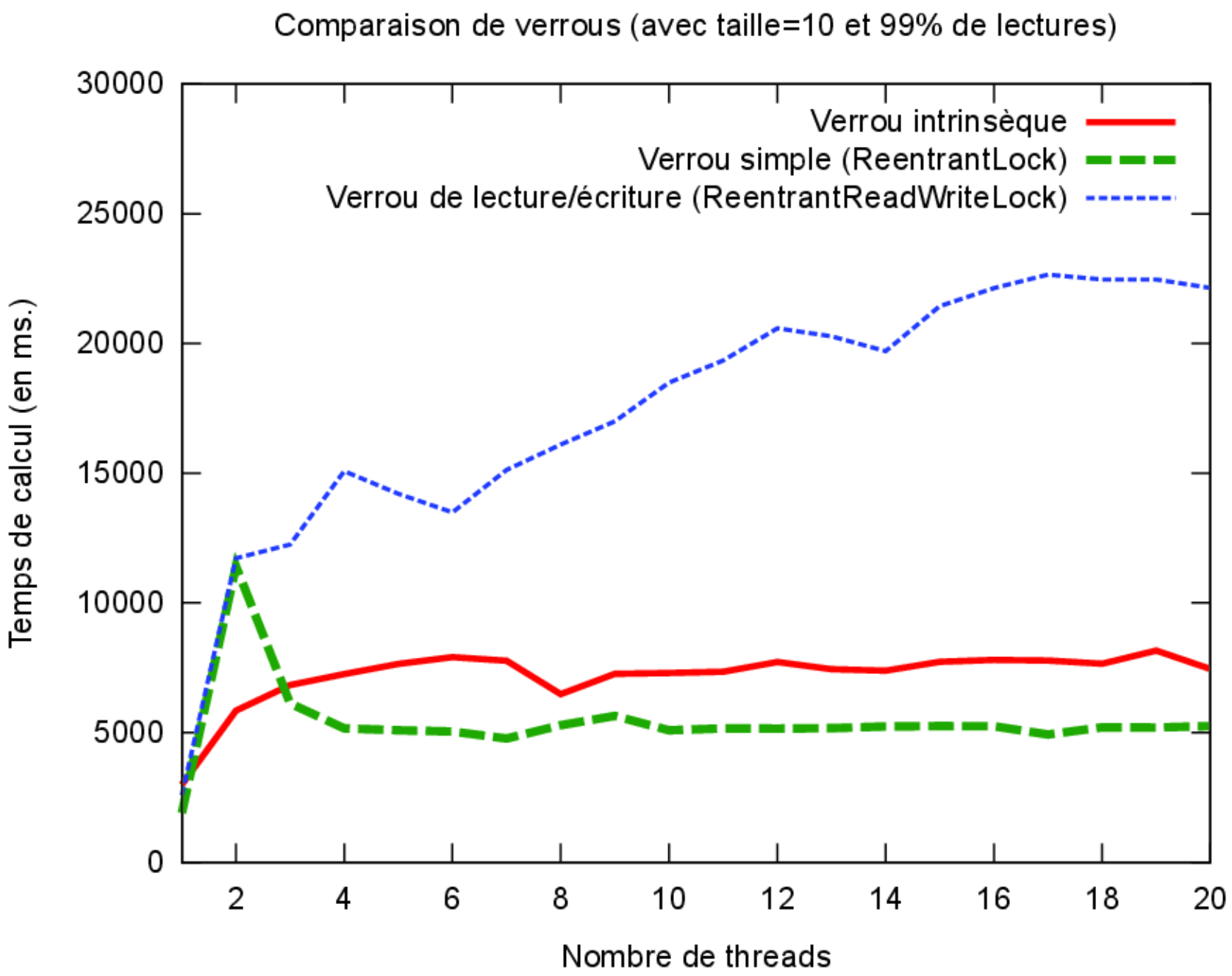
Utilisation du Stamped Lock (de Java 8)

Rappel : l'intérêt d'un verrou de Lecture/Ecriture



Plusieurs threads peuvent posséder le verrou de lecture simultanément !

Motivation : la faiblesse du verrou de Lecture/Écriture



Introduction du verrou « timbré »

Introduit dans Java 8, *le verrou timbré (StampedLock)* n'est pas véritablement un verrou : il n'implémente pas l'interface **`Lock`**.

- Il permet de distinguer les opérations de lecture et d'écriture, comme le verrou de Lecture/Écriture de la classe **`ReentrantReadWriteLock`**.
- Plus léger que le verrou de Lecture/Écriture, il sera **plus performant** dans beaucoup de situations où le verrou de Lecture/Écriture peut être envisagé.
- Les écritures ont lieu en exclusion mutuelle, mais elle ne sont pas bloquées par les lectures en cours : **les données lues peuvent être corrompues !**

  $\rightsquigarrow$ il faudra vérifier à la fin de la lecture qu'aucune écriture n'a eu lieu simultanément.
- **Il n'est pas réentrant !** Chaque appel renvoie un *timbre* différent.
- Il est possible de **transformer un timbre de lecture en un timbre d'écriture**.

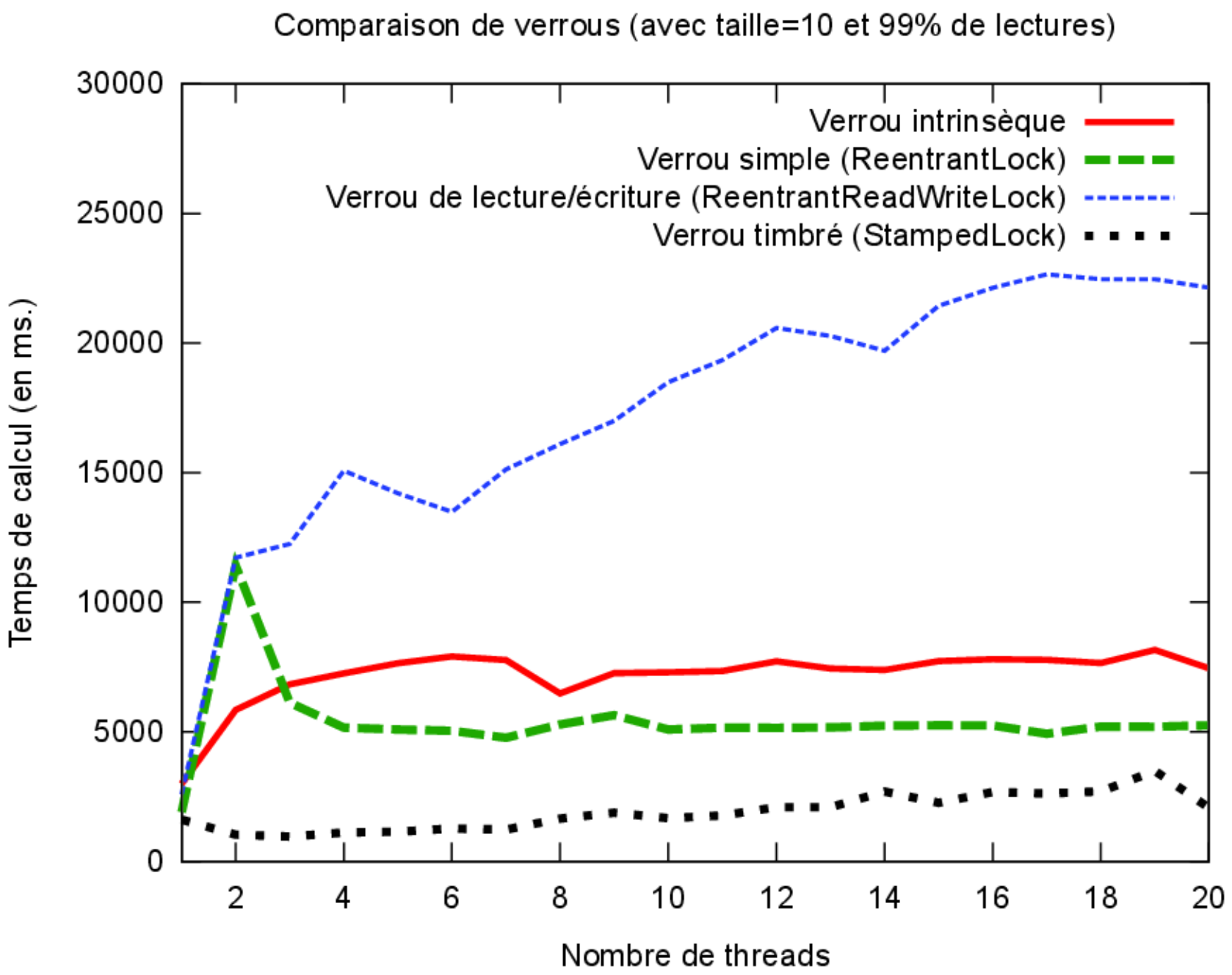
Utilisation d'un verrou « timbré » (1/2)

```
import java.util.concurrent.locks.StampedLock;
...
static private StampedLock verrouTimbré = new StampedLock();
...
public void run(){
  long timbre ;
  for (int i = 0 ; i < part; i++) {
    if (alea.nextInt(100) < 1) {          // 1% d'écritures
      timbre = verrouTimbré.writeLock(); // en exclusion mutuelle
      try {
        drapeaux[alea.nextInt(taille)] = alea.nextBoolean() ;
      } finally { verrouTimbré.unlockWrite(timbre); }
    }
    else {                                // 99% de lectures
```

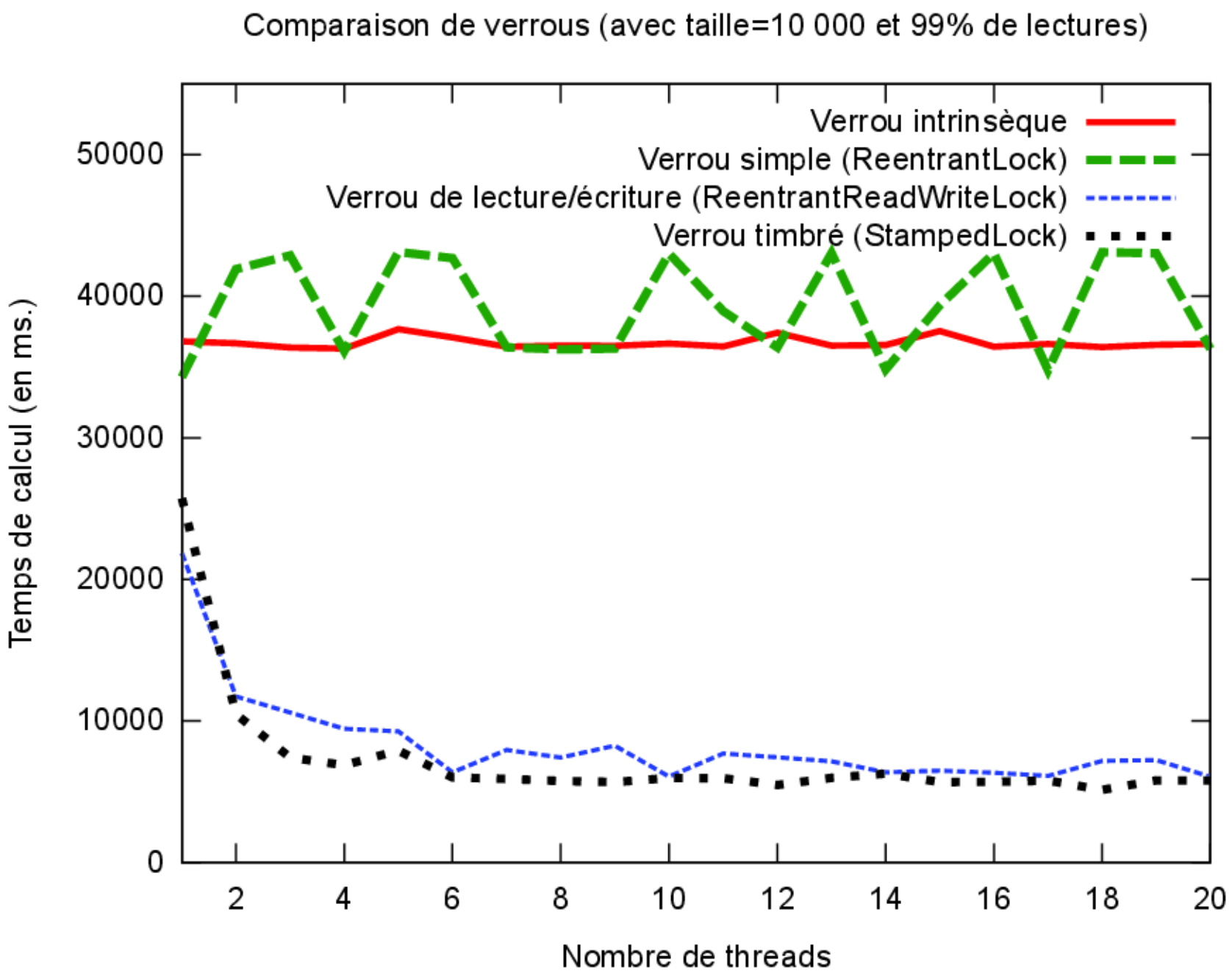
Utilisation d'un verrou « timbré » (2/2)

```
  else {                                                  // 99% de lectures
    long timbre = verrouTimbré.tryOptimisticRead();
    somme = 0;
    for (int j=0 ; j < taille ; j++) if (drapeaux[j]) somme++ ;
    if (! verrouTimbré.validate(timbre)){ // Il y a eu corruption!
      timbre = verrouTimbré.readLock();    // Plus de risque...
      try {
        somme = 0;
        for (int j=0; j < taille; j++) if (drapeaux[j]) somme++ ;
      } finally { verrouTimbré.unlockRead(timbre);}
    }
  }
}
```

Performances sur un tableau à 10 élements



Performances sur un tableau à 10 000 éléments



- ✔ *Utilisation du Stamped Lock (de Java 8)*
- ☛ *Emploi de collections concurrentes*

Motivations

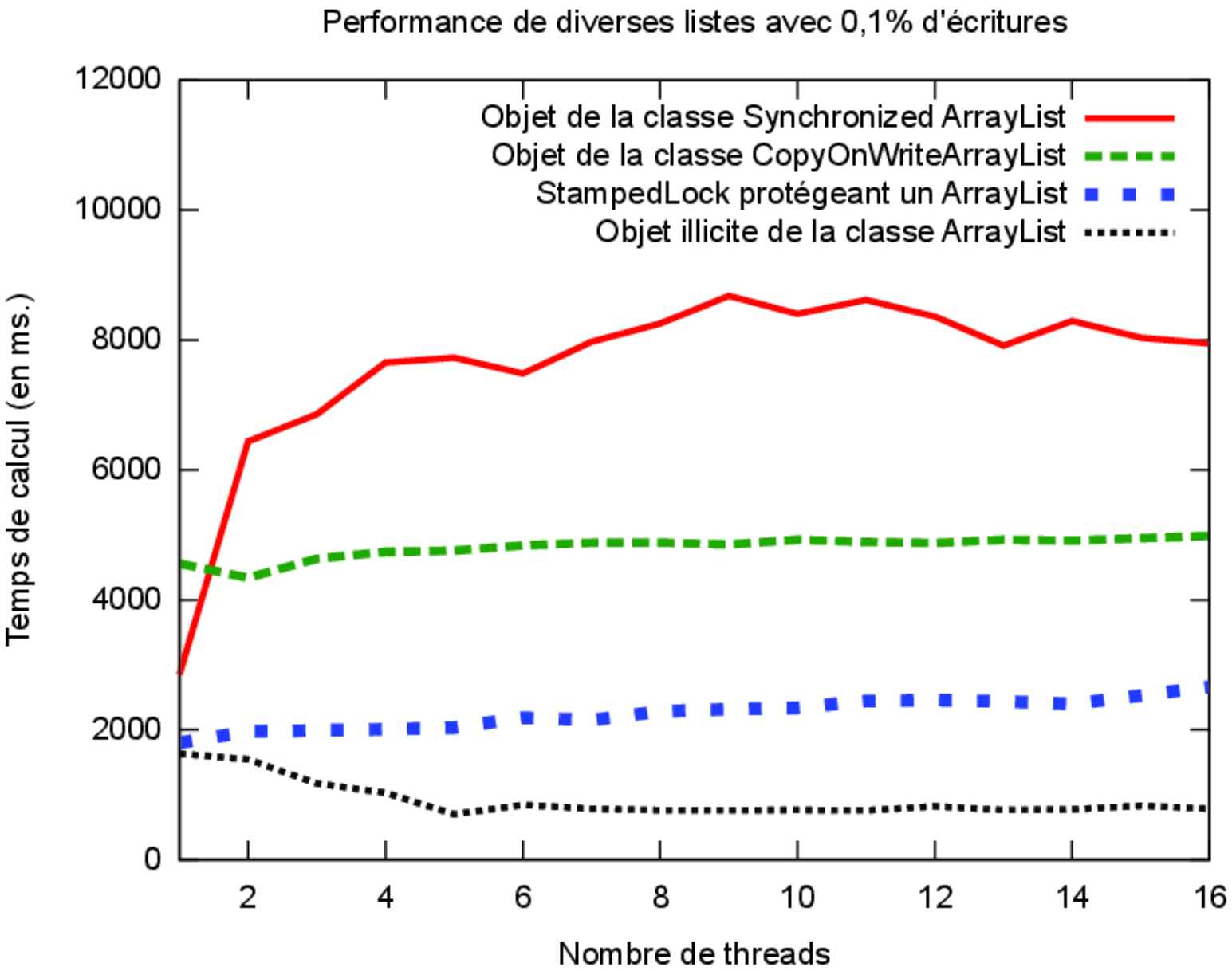
Les collections « concurrentes » autorisent souvent plusieurs modifications des données en parallèle, de même que des lectures en même temps que des modifications.

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

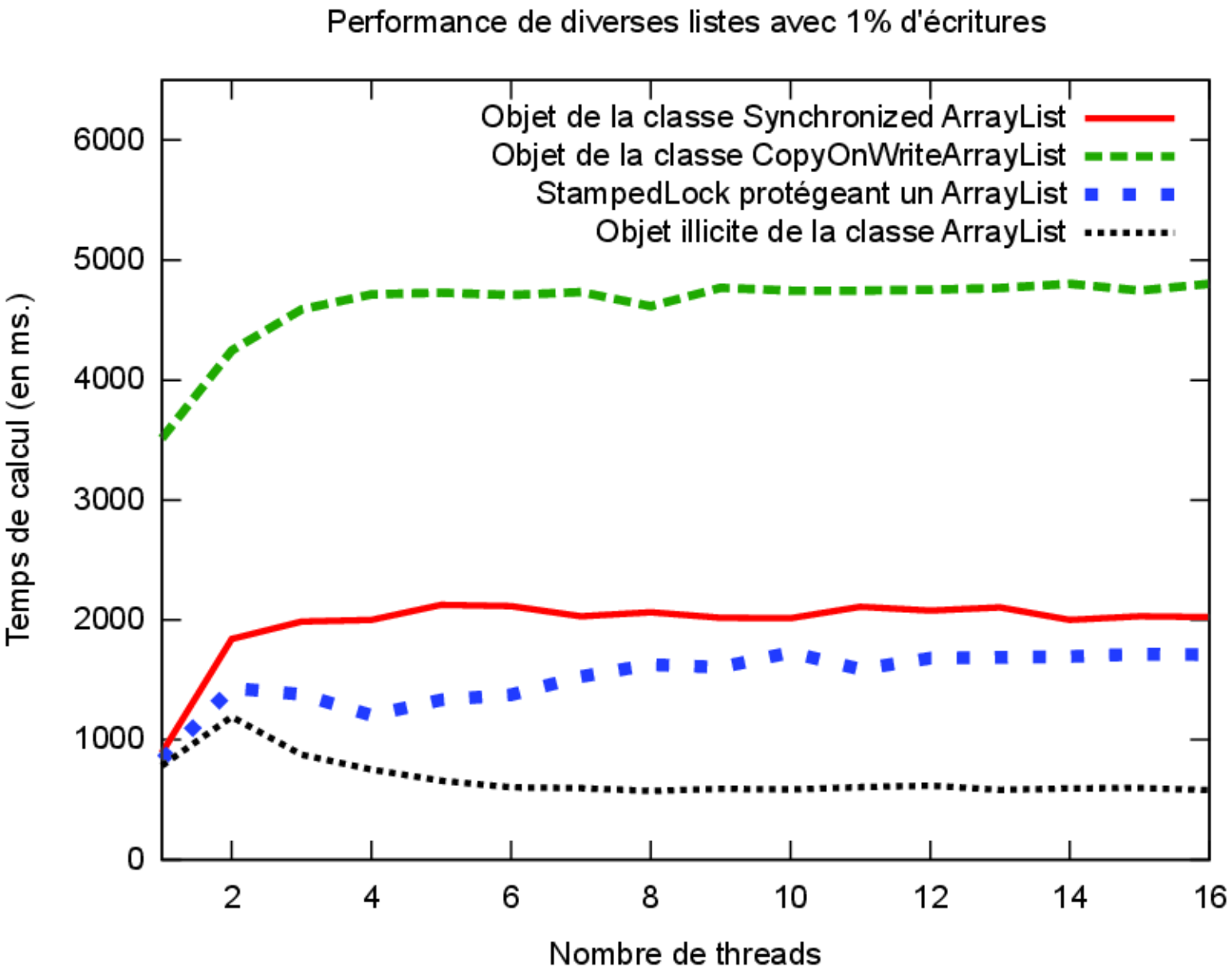
Comparaison de listes : benchmark

```
static List<Integer> liste ;
static int taille = 1000                    // Taille constante de la liste
static int rapportLectureEcriture = 100;
...
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 100
    } else {
      liste.get(index) ;                  // Lecture le reste du temps
    }
  }
}
```

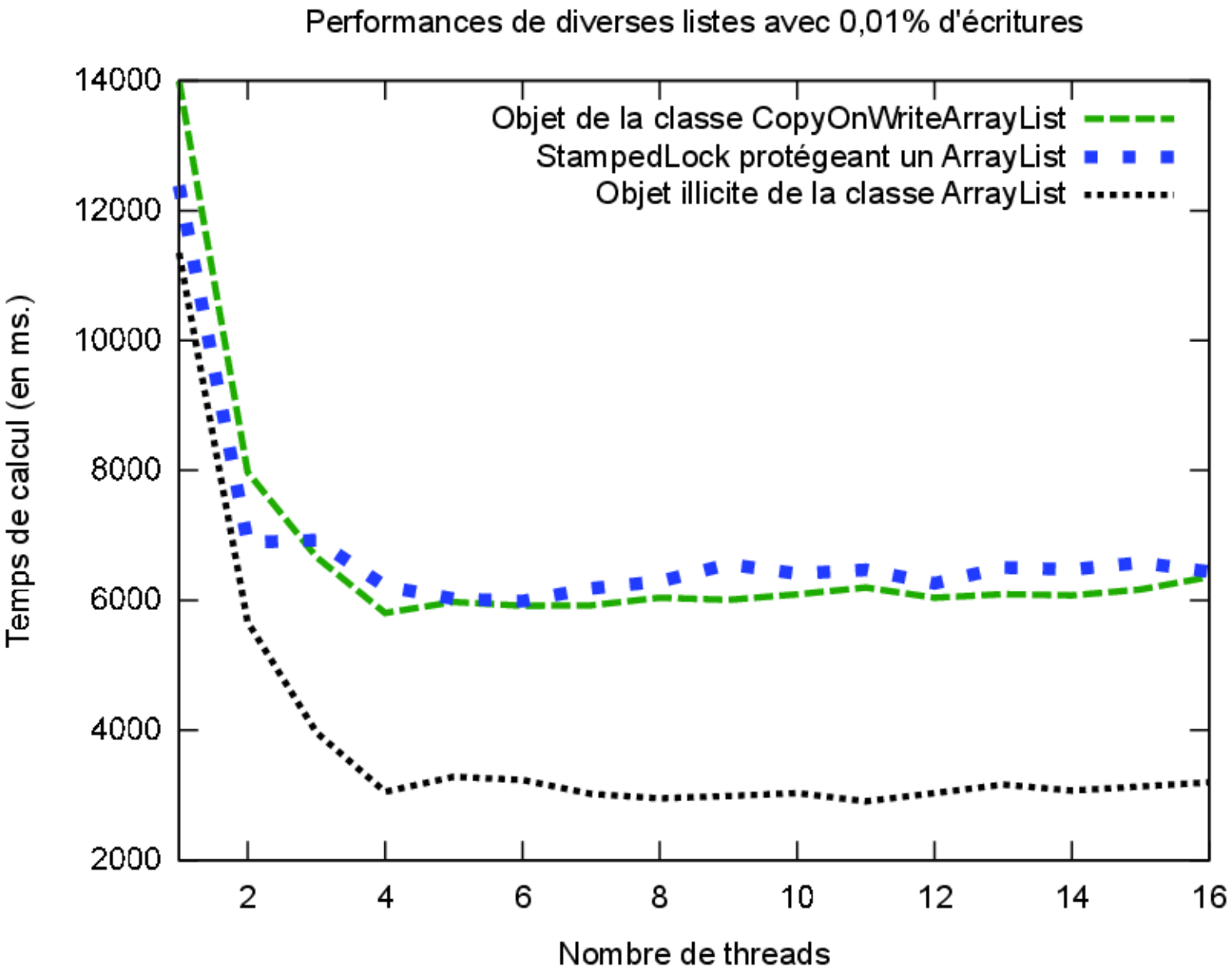
Comparaison de listes (1/3) : intérêt du StampedLock



Comparaison de listes (2/3) : faiblesse du CopyOnWriteArrayList



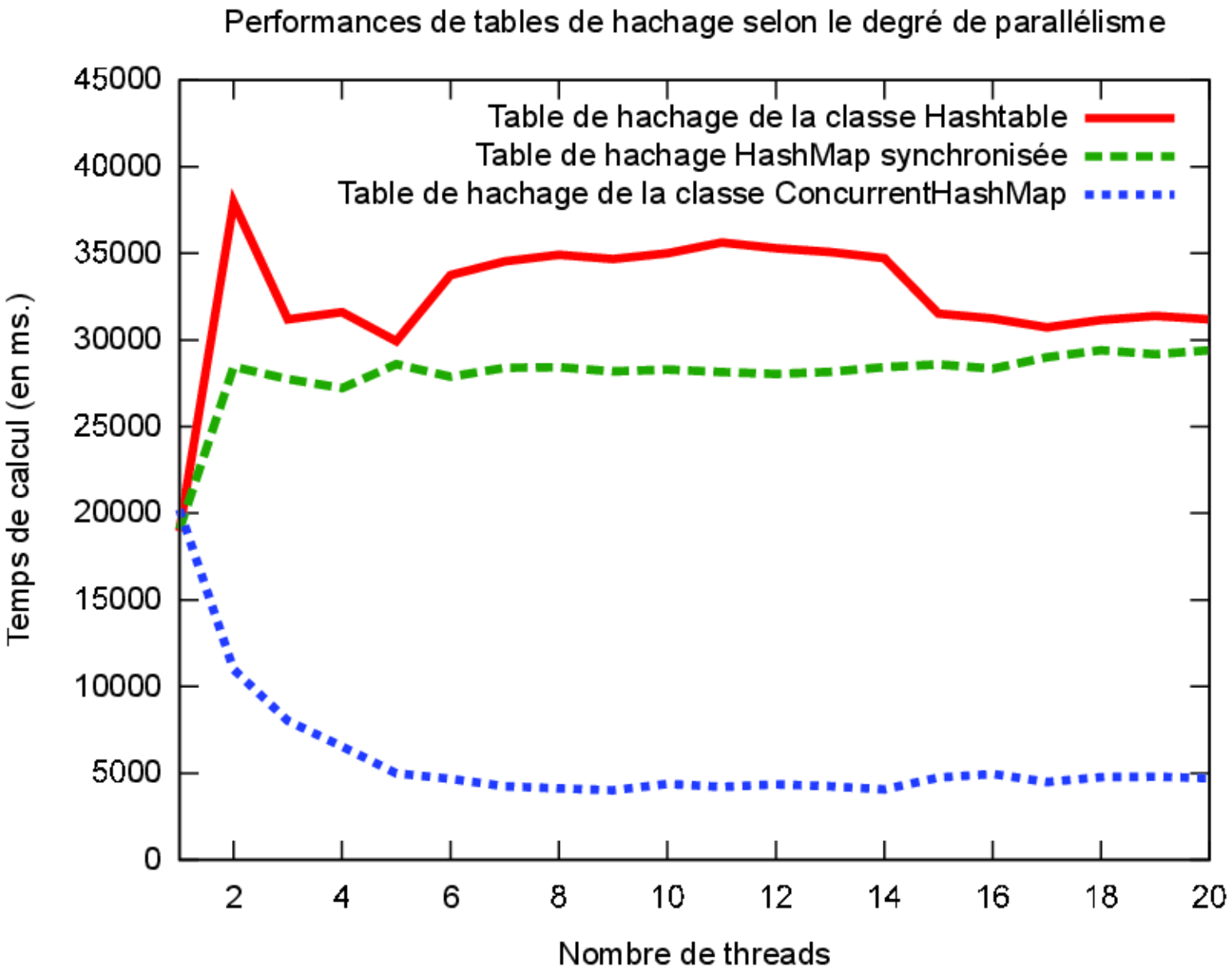
Comparaison de listes (3/3) : efficacité de CopyOnWriteArrayList



Comparaison de 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



Un problème d'atomicité

Utiliser une collection concurrente ne garantit pas de produire du code thread-safe, car **la composition d'instructions atomiques ne forme que très rarement 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 issue 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.*

- ✔ *Utilisation du Stamped Lock (de Java 8)*
- ✔ *Emploi de collections concurrentes*
- ☛ *Exemple d'écriture optimiste*

À propos des verrous

Les verrous sont un moyen simple de se prémunir contre toute interférence préjudiciable entre les parties du programme qui peuvent s'exécuter en parallèle, en garantissant l'**atomicité** de parties de code ≪ critiques. ≫

Néanmoins les verrous pâtissent d'un certain nombre d'inconvénients bien connus :

- Les **interblocages** sont un risque principal, qui demande de la rigueur ;
- Les **performances sont réduites** s'il y a inutilement trop de verrous ;
- Les **performances sont réduites** si un thread *lent* saisit une série de verrous ;
- Les **inversions de priorité**, lorsqu'un thread *non-prioritaire* possède un verrou ;
- **L'intolérance aux fautes**, si un thread s'arrête en possédant un verrou ;
- Les **structures de données** proscrivent souvent les mises-à-jour en parallèle.

Exemple : calcul d'une image fractale ligne par ligne

Nous avons étudié lors du premier TP comment paralléliser le calcul de l'image fractale ci-dessous en lançant 4 threads collaboratifs.

- L'image est calculée **ligne par ligne**.
- Certaines lignes sont **plus longues à calculer** que d'autres.
- Les lignes sont attribuées **dynamiquement** à chaque thread, en fonction de leur disponibilité.

Il faut ici incrémenter le compteur des lignes déjà attribuées **de manière atomique** pour

1. ne pas calculer deux fois la même ligne ;
2. s'assurer de calculer toutes les lignes ;
3. tout en dessinant correctement chaque ligne attribuée.

Retour sur le problème de l'incrémentation bornée

```
static volatile private int cpt = 0;
static volatile private Object monVerrou = new Object();

public int récupérer(){
  synchronized(monVerrou){
    if ( cpt > taille ) return taille;
    return cpt++; // renvoie la valeur non-incrémentée
  }               // celle-ci sera toujours <= taille
}
```

Quand la valeur **`taille`** est retournée, toutes les lignes de 0 à **`taille`**-1 ont déjà été attribuées : le dessin est terminé. La méthode **`récupérer()`** renverra successivement 0, 1, 2, ..., **`taille`**, puis indéfiniment **`taille`**.

La même chose sans verrou ?

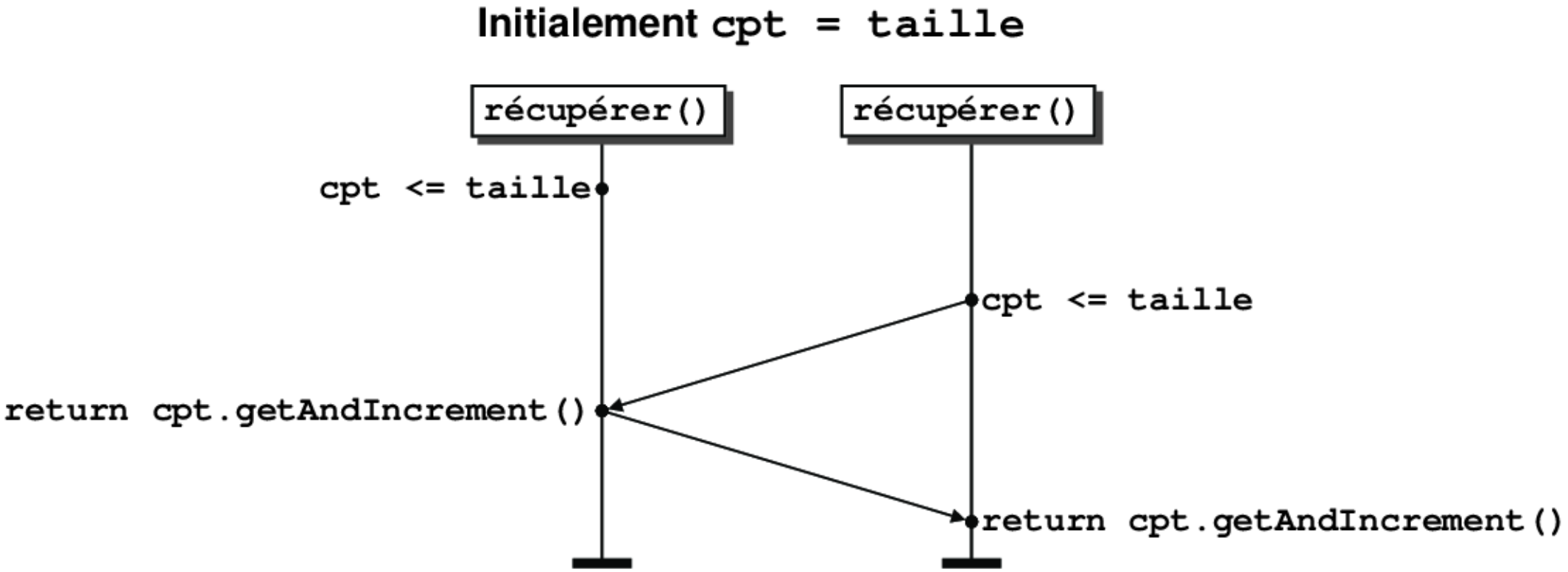
Une première solution sans verrou : utiliser un compteur atomique

```
import java.util.concurrent.atomic.AtomicInteger;
...
  private static AtomicInteger cpt = new AtomicInteger(0);
  public int récupérer() {
    if ( cpt.get() > taille ) return taille;
          // cpt.get() renvoie un int
    return cpt.getAndIncrement();
           // équivalent à un cpt++ réalisé atomiquement;
  }
...
```

Cette méthode alternative n'est pas atomique ; elle ne retournera pas nécessairement une valeur $\leqslant$ **taille** !

Exercice : Comment corriger ceci ?

Un scenario extrême



Que renvoie chacun des deux threads ?

Une opération atomique fondamentale, mais un peu curieuse

```
boolean compareAndSet(valeurAttendue, valeurNouvelle)
```

⇝ Affecte **atomiquement** la valeur **`valeurNouvelle`** dans l'objet atomique **à condition que** la valeur courante de cet objet soit effectivement égale à **`valeurAttendue`**.

Il faut deviner la valeur courante pour la modifier !

⇝ Retourne **`true`** si l'affectation a eu lieu, **`false`** si la valeur courante de l'objet atomique est différente de **`valeurAttendue`** au moment de l'appel.

Un appel à **`compareAndSet()`** sera remplacé par la machine virtuelle Java par l'opération assembleur correspondante dans la machine : par exemple, les instructions de comparaison et échange CMPXCHG8B ou CMPXCHG16B des processeurs Intel.

Recette pour programmer avec un objet atomique

Pour modifier correctement un objet atomique, il suffit en général de

① fabriquer une copie de la valeur courante de l'objet atomique ;

② préparer une modification de l'objet *à partir de la copie obtenue* ;

③ appliquer une mise-à-jour de l'objet atomique conformément à l'étape 2, si sa valeur courante correspond encore à celle copiée à l'étape 1 (et sinon, recommencer).

Exemple d'implémentation de **`getAndIncrement()`** pour un **`AtomicInteger`** :

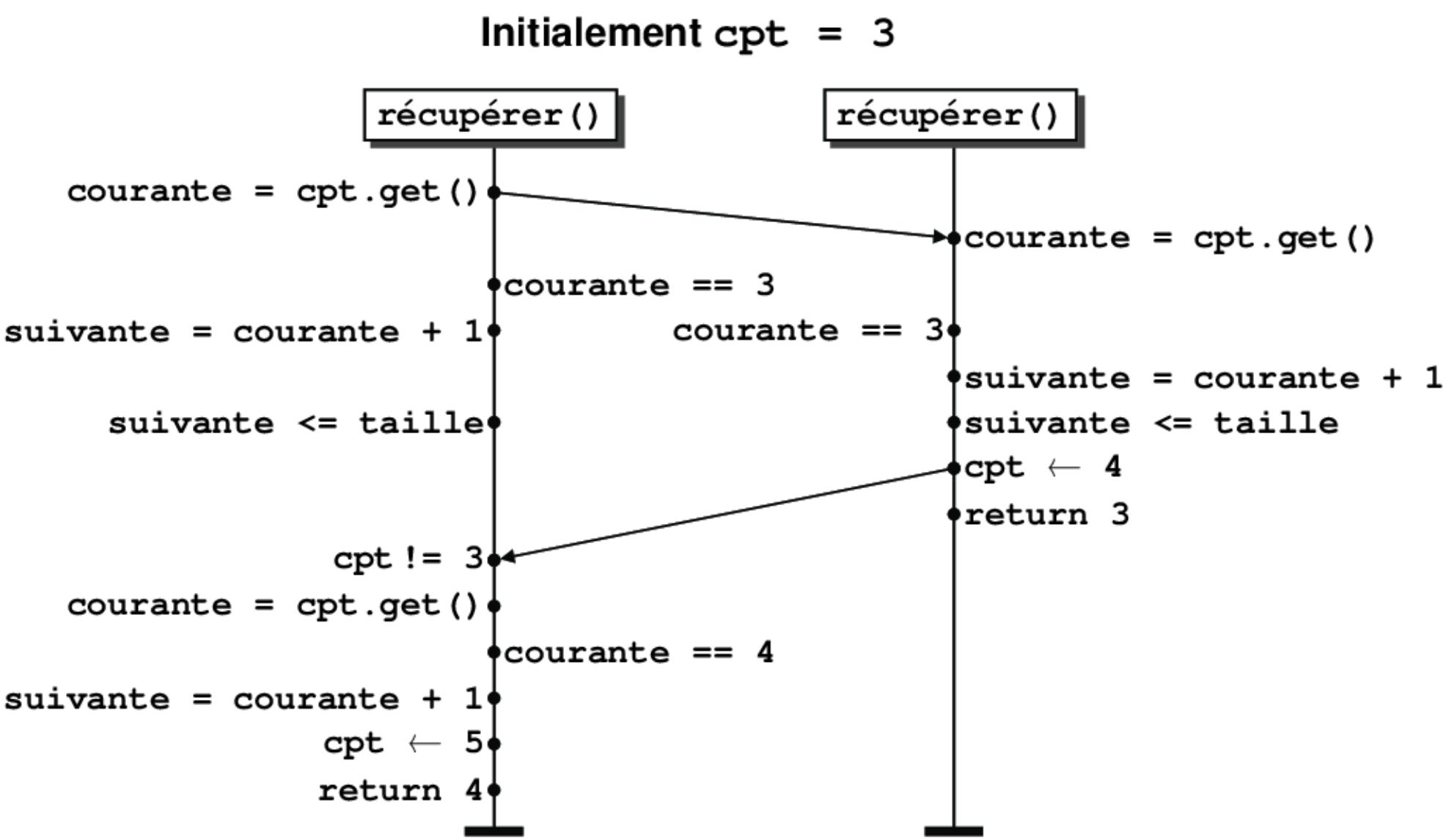
```
public final int getAndIncrement() {
  for (;;) {
    int courant = get();
    int suivant = courant + 1;
    if (compareAndSet(courant, suivant)) return courant;
  }
}
```

Exemple typique de codage « optimiste »

```
private static AtomicInteger cpt = new AtomicInteger(0);
public int récupérer(){
  do {
    int courante = cpt.get();
    int suivante = courante + 1;
    if ( suivante > taille ) suivante = taille;
  } while ( ! cpt.compareAndSet(courante, suivante) );
  return courante; // Cette valeur sera toujours <= taille
}
```

Si aucun thread n'interfère entre le début et la fin d'une itération alors la méthode **`compareAndSet(courante, suivante)`** écrira effectivement la valeur **`suivante`** dans l'entier atomique **`cpt`** : la boucle terminera et la méthode renverra la valeur de **`cpt`** avant sa modification. En revanche, **`compareAndSet(courante, suivante)`** ne fera rien, et renverra **`false`**, si la valeur de **`cpt`** a été modifiée, car alors **`courante`** ne correspond plus à la valeur courante de **`cpt`**. Il y aura un nouveau tour de boucle !

Fonctionnement non-atomique de `récupérer()` (1/2)



Le thread à droite réussit le **`compareAndSet()`** en premier : il renvoie **3** ; l'autre thread exécute un *tour de boucle supplémentaire* et renvoie **4**.

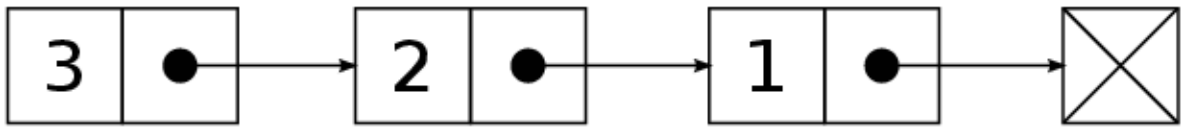
Exemple d'une pile concurrente

Master Informatique — Semestre 2 — UE obligatoire de 3 crédits

La brique de base : le noeud

```
class Noeud {                  // On s'intéresse à une pile d'entiers
    final Integer valeur;
    Noeud suivant;             // Référence vers le noeud suivant

    public Noeud(Integer valeur) {              // Constructeur
      this.valeur = valeur;
    }
}
```

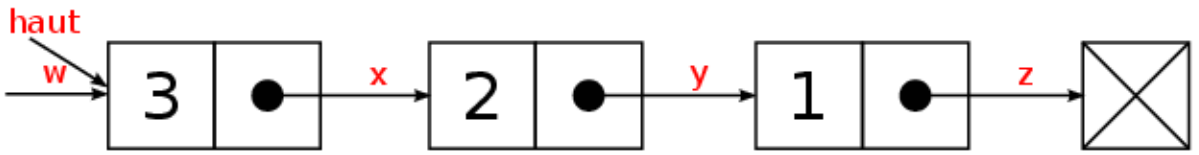


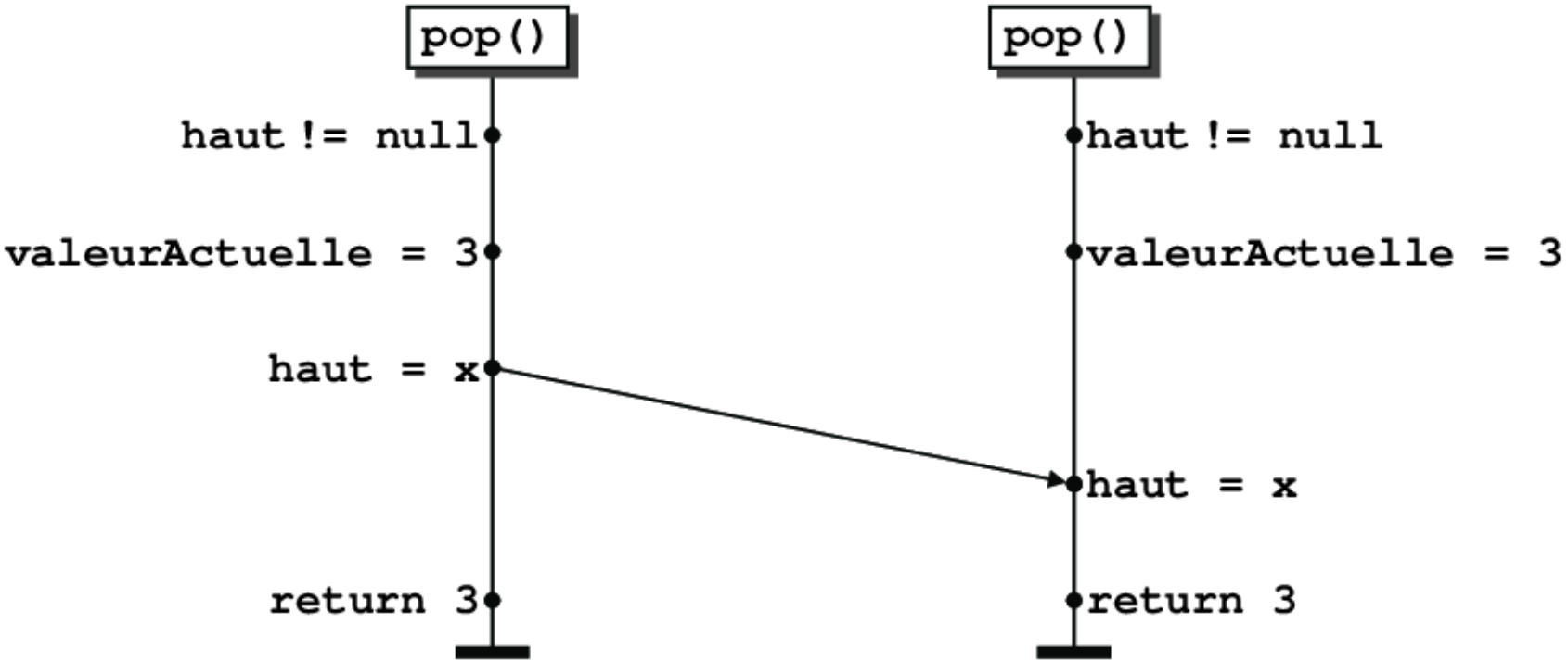
Une pile synchronisée (avec un verrou, donc)

```
class PileSynchronisee { // C'est une espèce de liste de Noeuds
  private volatile Noeud haut;
  // haut est (une référence vers) le premier noeud de la pile
  public synchronized void push(Integer valeur) {
    Noeud nouveauHaut = new Noeud(valeur);
    nouveauHaut.suivant = haut;
    haut = nouveauHaut;       // Modification du haut de la pile
  }
  public synchronized Integer pop() {
    if (haut == null) return null;    // La pile peut être vide
    Integer valeurActuelle = haut.valeur;
    haut = haut.suivant;      // Modification du haut de la pile
    return valeurActuelle;
  }
}
```

La même chose sans verrou ?

Évidemment, on ne doit pas supprimer brutalement le verrou





À la fin, les deux threads renvoient la valeur 3 et la pile contient encore deux éléments : «2» suivi de «1».

Algorithme de Treiber (1/2)

```
public class PileConcurrente {      // Code emprunté au livre JCIP
  AtomicReference<Noeud> haut = new AtomicReference<Noeud>();
  // haut est une référence à un noeud, manipulable atomiquement
  // Il désigne le noeud en haut de la pile (le premier noeud)
  public void push(Integer valeur) {
    Noeud nouveauHaut = new Noeud(valeur);
    Noeud actuelHaut ;
    do {
      actuelHaut = haut.get(); // Référence vers le 1er noeud
      nouveauHaut.suivant = actuelHaut;
    } while ( ! haut.compareAndSet(actuelHaut, nouveauHaut) );
  }
```

Si cette méthode s'exécute de manière atomique, alors les références **`haut.get()`** et **`actuelHaut`** restent égales : le CAS s'applique et **`haut`** prend la valeur de **`nouveauHaut`**, c'est-à-dire une référence vers le nouveau noeud.

Seconde recette : pour programmer avec des références atomiques

Les objets atomiques sont précieux pour programmer sans verrou. Mais il en existe de peu de sortes... Pour manipuler des données un peu complexes de manière atomique, comme ici un noeud, il faut souvent encapsuler les données dans un objet et utiliser une **référence atomique** à cet objet.

Pour effectuer correctement une modification de l'objet désigné par la référence atomique, il suffit en général

① d'obtenir une copie de la valeur de la référence atomique (ici, la valeur de **`haut`**) de l'objet X (de la classe **`Noeud`**) ;

② de construire, *à l'aide de la référence copiée*, un **nouvel objet** Y, de la même classe, qui met en oeuvre la modification demandée : Y est un candidat pour remplacer X.

③ d'appliquer une mise-à-jour de la référence atomique vers la référence de Y construite à l'étape 2, si la valeur courante de cette référence atomique correspond encore à celle de X, copiée à l'étape 1 (et sinon, recommencer).

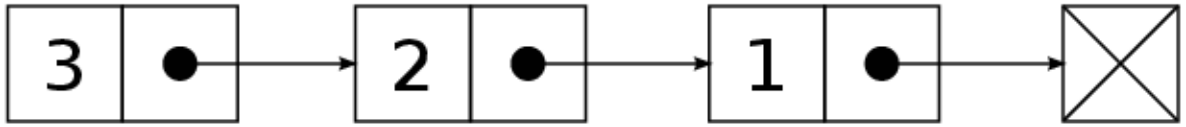
Algorithme de Treiber (2/2)

```
  public Integer pop() {
    Noeud actuelHaut;
    Noeud nouveauHaut;
    do {
      actuelHaut = haut.get();
      if (actuelHaut == null) return null;
      nouveauHaut = actuelHaut.suivant;
    } while ( ! haut.compareAndSet(actuelHaut,nouveauHaut) );
    return actuelHaut.valeur;
  }
}
```

Si cette méthode s'exécute de manière atomique, alors les références **`haut.get()`** et **`actuelHaut`** restent égales : CAS s'applique et **`haut`** prend la valeur de **`nouveauHaut`**, c'est-à-dire une référence vers le noeud en seconde position.

Le problème ABA

À votre avis ?

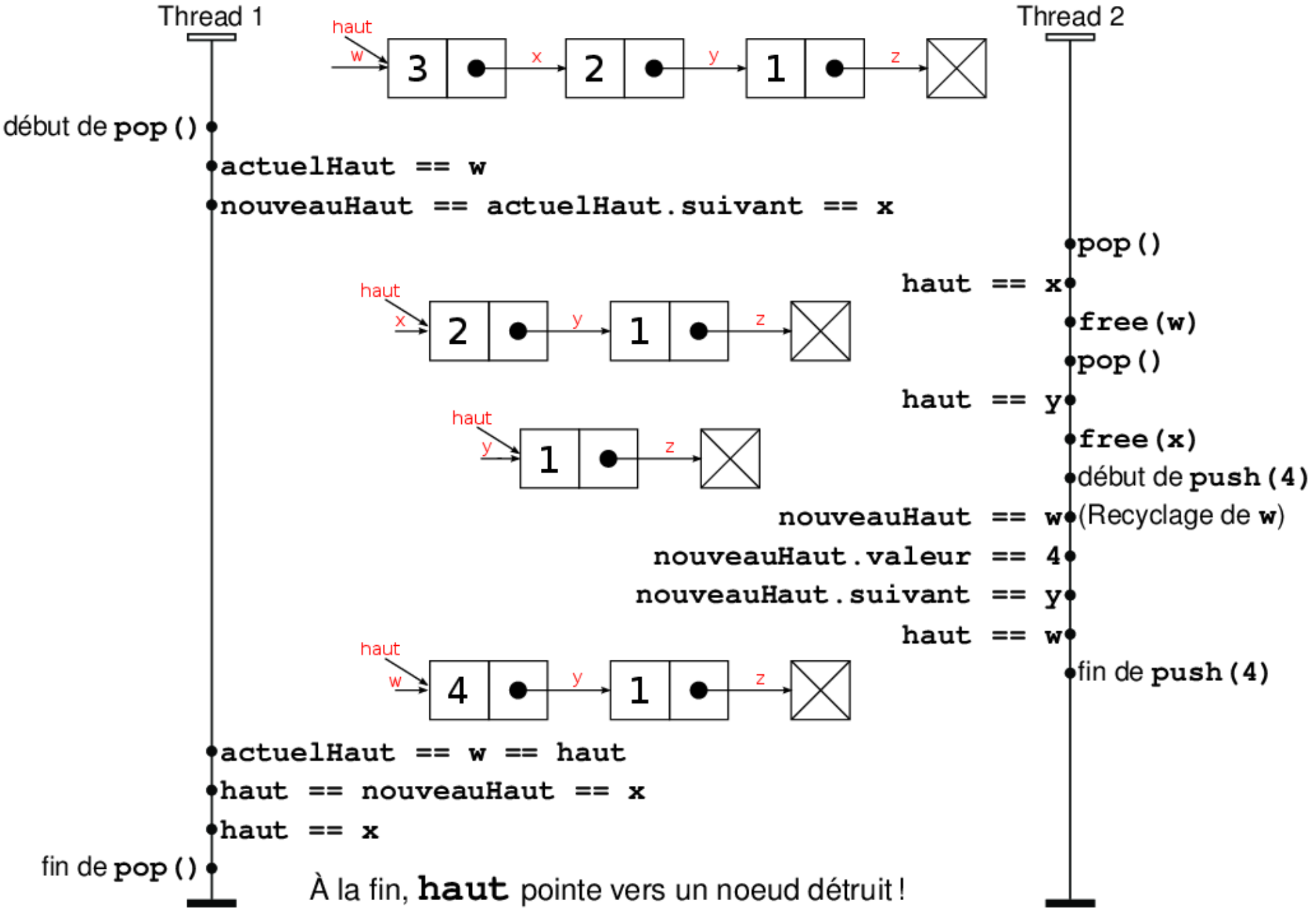


Un thread appelle `pop()`.

En parallèle, un autre thread exécute `pop();pop();push(new Integer(4))`.

Que contiendra la pile à la fin ?

Scenario ABA problématique, si on code en C (c'est-à-dire sans GC)



Problème ABA

Ce scenario illustre le problème connu sous le nom de « ABA » :

- Le Thread 1 observe une variable partagée dont la valeur est **`A`** ;
- Le Thread 1 prépare une modification atomique de cette variable en s'appuyant sur la valeur courante observée ;
- Le Thread 2 modifie la variable partagée et lui attribue la valeur **`B`** ;

> Si le Thread 1 reprend la main ici et essaie d'appliquer le CAS, il observera le changement de valeur et le CAS ne s'appliquera pas ; le Thread 1 devra recommencer : tout va bien !

- Le Thread 2 continue et modifie une seconde fois la valeur de la variable partagée et lui attribue à nouveau la valeur **`A`** ;
- Le Thread 1 reprend la main et essaie d'appliquer le CAS : la valeur trouvée étant celle attendue, le CAS s'applique (mais il ne devrait peut-être pas).

> Si la variable partagée contient simplement une *donnée* (Boolean, Integer, ou Long), tout va bien : la modification préparée reste cohérente.
>
> En revanche, si la variable partagée est une **référence** (comme **`haut`**) et si la modification préparée s'appuie sur l'objet référencé (comme **`nouveauHaut`**), cette modification n'est plus valide car l'objet référencé par **`A`** peut avoir changé, même si **`A`** apparaît inchangée.

Absence de problème ABA en Java (sur ce code)

Le scenario catastrophe décrit ici ne peut avoir lieu en Java, avec le code présenté, car

- Lors du `pop()` du Thread 1, `actuelHaut` devient une nouvelle référence vers le haut de la pile : `actuelHaut == w`.
- **Le ramasse-miette (GC) ne peut pas réclamer ce noeud, même s'il est supprimé de la pile par le Thread 2, car il reste une référence active vers cet objet.**
- Le noeud construit pour insérer 4 aura une référence différente de `w`.
- Le nouveau haut de la pile contiendra alors une référence différente de `w`.
- Le Thread 1 constatera que le haut de la pile a été modifié : le CAS ne s'appliquera pas ; le Thread 1 devra recommencer un tour de boucle pour effectuer le `pop()` réclamé.

- ✔ *Le problème ABA*
- ☛ *Illustration du problème ABA en Java*

Une pile de noeuds (1/2)

Le ramasse-miette est un atout important, mais subtile, pour éviter le problème ABA en Java. Ce n'est cependant pas une panacée. Dans certains cas, les références utilisées par une structure fuient en dehors de cette structure, ce qui ramène au problème ABA.

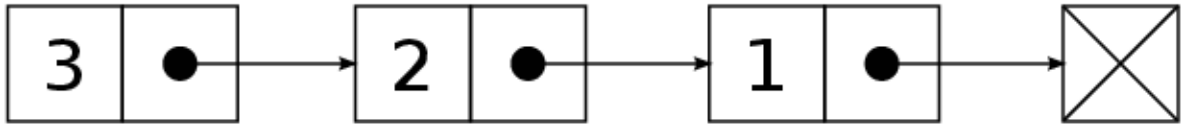
```
public class PileDeNoeuds {
  private AtomicReference<Noeud> haut=new AtomicReference<Noeud>();
  // haut est une référence à un noeud, manipulée atomiquement

  public void push(Noeud entrée) {
    Noeud actuelHaut;
    do {
      actuelHaut = haut.get(); // Référence du premier noeud
      entrée.suivant = actuelHaut;
    } while ( ! haut.compareAndSet(actuelHaut, entrée) );
  }
```

Une pile de noeuds (2/2)

```
  public Noeud pop() {
    Noeud actuelHaut;
    Noeud nouveauHaut;
    do {
      actuelHaut = haut.get();
      if (actuelHaut == null) return null;
      nouveauHaut = actuelHaut.suivant;
    } while ( ! haut.compareAndSet(actuelHaut,nouveauHaut) );
    return actuelHaut;
  }
}
```

À votre avis ?



Un thread appelle **`A=pop()`**.

Un autre thread exécute **`A'=pop(); B=pop(); push(A')`**.

Que peut contenir la pile à la fin ?

— «1» ?

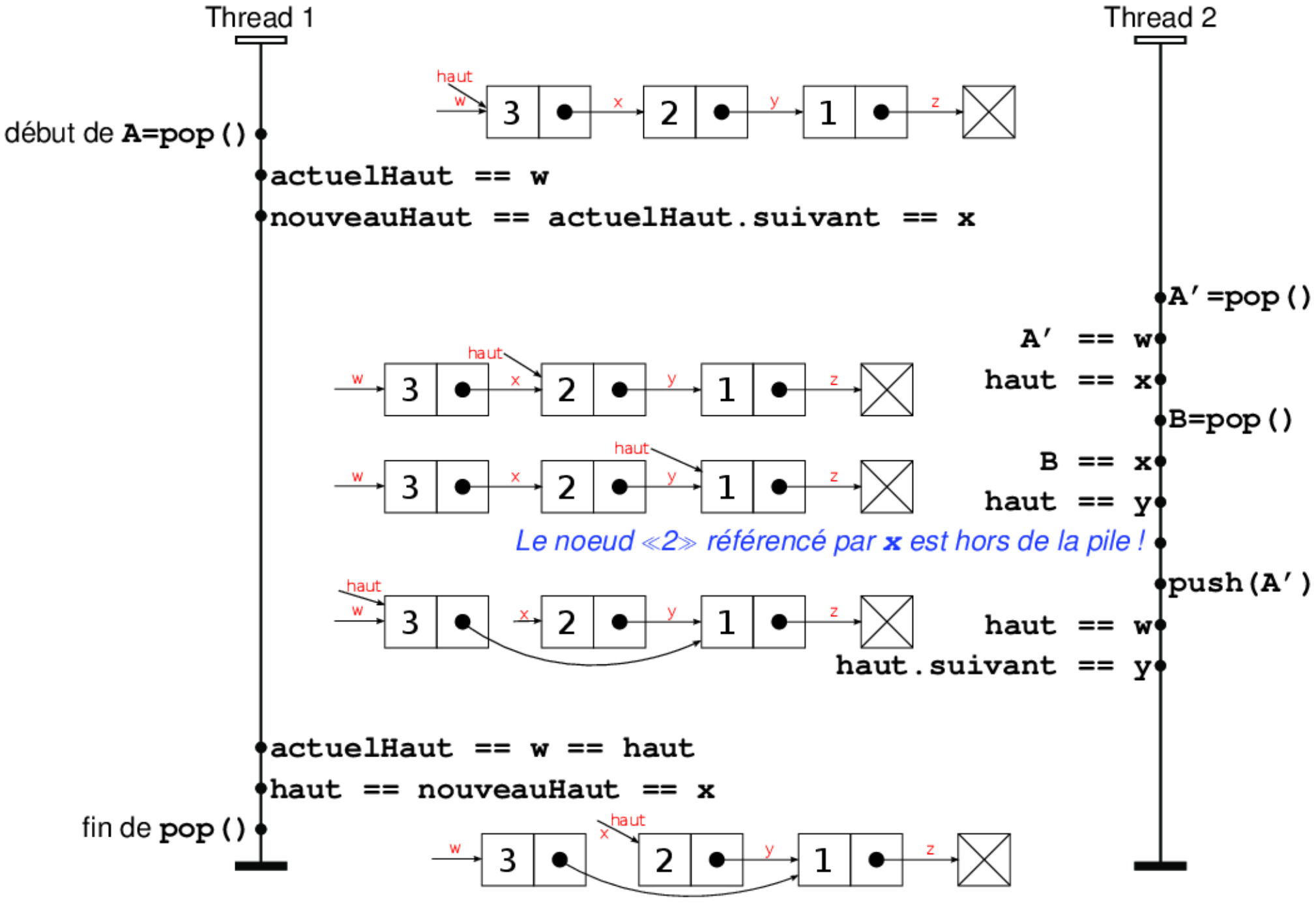
— «2» ?

— «3» ?

— «2» suivi de «1» ?

— autre chose ?

Problème ABA avec cette manipulation de noeuds (en Java)



- ✔ *Le problème ABA*
- ✔ *Illustration du problème ABA en Java*
- ☛ *Un mot sur les performances*

Bémol à propos de cet exemple

Une pile n'est vraiment pas la structure de données la plus pertinente pour illustrer la conception de structures de données non bloquantes (sans verrou) :

- Le **gain de performance**, s'il y en a un, sera faible ;
- Une file ou une liste triée, avec des opérations plus complexes, serait plus appropriée.

Néanmoins, l'étude de cette pile permet :

- de se convaincre que *l'éventuel surcoût induit par la suppression des verrous devrait être nul, ou faible ;*
- d'approfondir un peu l'étude des *techniques de programmation sans verrou* à l'aide de références atomiques ;
- d'introduire *le problème ABA*.

Un mot sur les performances

Dans une situation de contention faible, c'est-à-dire lorsqu'il y a peu de threads agissant en concurrence sur la structure de donnée, les structures sans verrou seront en général plus performantes que les structures « synchronisées » parce que

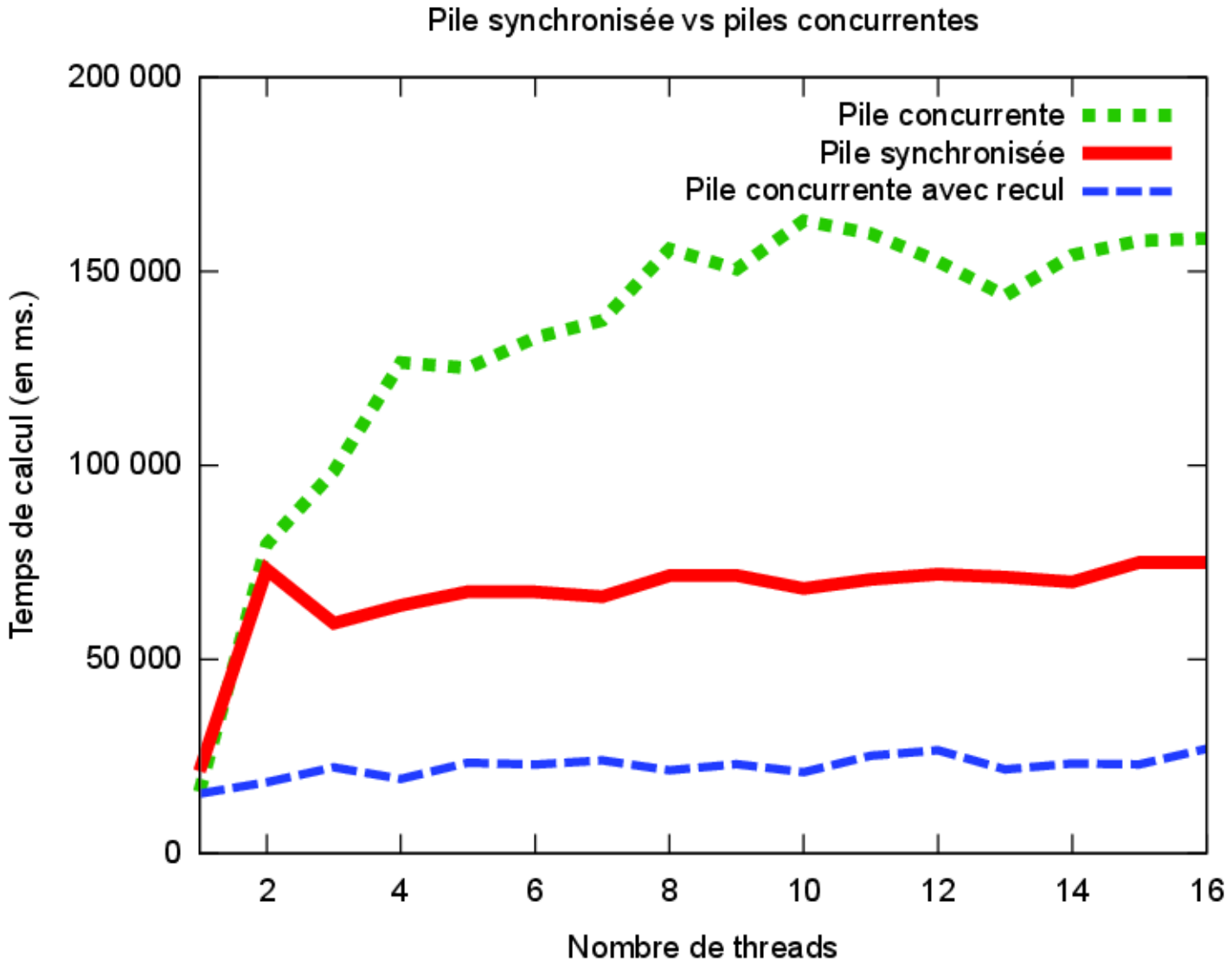
- l'opération CAS réussira la plupart du temps ;
- *la prise d'un verrou coûte au moins aussi chère qu'une opération CAS, et souvent plus* (même en cas de contention légère) ;
- la boucle d'itération, en cas d'échec fugace, évite un changement d'état interne du thread et la perte de l'accès au processeur (changement de contexte).

En cas de forte contention, c'est-à-dire lorsque de nombreux threads tentent d'accéder aux mêmes données, les verrous (notamment les verrous intrinsèques) offrent souvent de meilleures performances que l'attente active ; mais ces situations sont assez rares (et parfois artificielles). **Elles peuvent être aussi le fruit d'une mauvaise conception**. Enfin, les techniques utilisées par les verrous pour la *gestion de la contention* peuvent être appliquées à ces structures de données.

Benchmark : empilage et dépilage aléatoires et répétitifs

```
public void run(){
  for (int i = 1; i <= part; i++){
    if (alea.nextInt(100) < pourcentage) {
      pile.push(alea.nextBoolean()) ;
      // EMPILEMENT D'UN BOOLÉEN ALÉATOIRE
    }
    else {
      pile.pop() ;
      // DÉPILEMENT D'UN ÉLÉMENT
    }
  }
}
```

Performances sur une machine à 8 coeurs



En cas de forte contention, pour la pile concurrente, le temps de calcul est proportionnel au nombre de threads actifs (8 au maximum) : **`haut`** est un « hotspot. »

Ajout du recul en cas d'observation d'une contention

```
class PileConcurrenteAvecRecul {
  AtomicReference<Noeud> haut = new AtomicReference<Noeud>();
  public void push(Boolean valeur) {
    int delai = DELAI_MIN ;
    Noeud nouveauHaut = new Noeud(valeur) ;
    Noeud actuelHaut ;
    for (;;) {
      actuelHaut = haut.get() ;
      nouveauHaut.suivant = actuelHaut ;
      if ( haut.compareAndSet(actuelHaut, nouveauHaut) ) return ;
      // Il y a contention sur la variable haut: reculons !
      Thread.sleep( alea.nextInt(delai) ) ;
      if (delai <= DELAI_MAX) delai += delai ;
    }
  }
```