La programmation optimiste

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

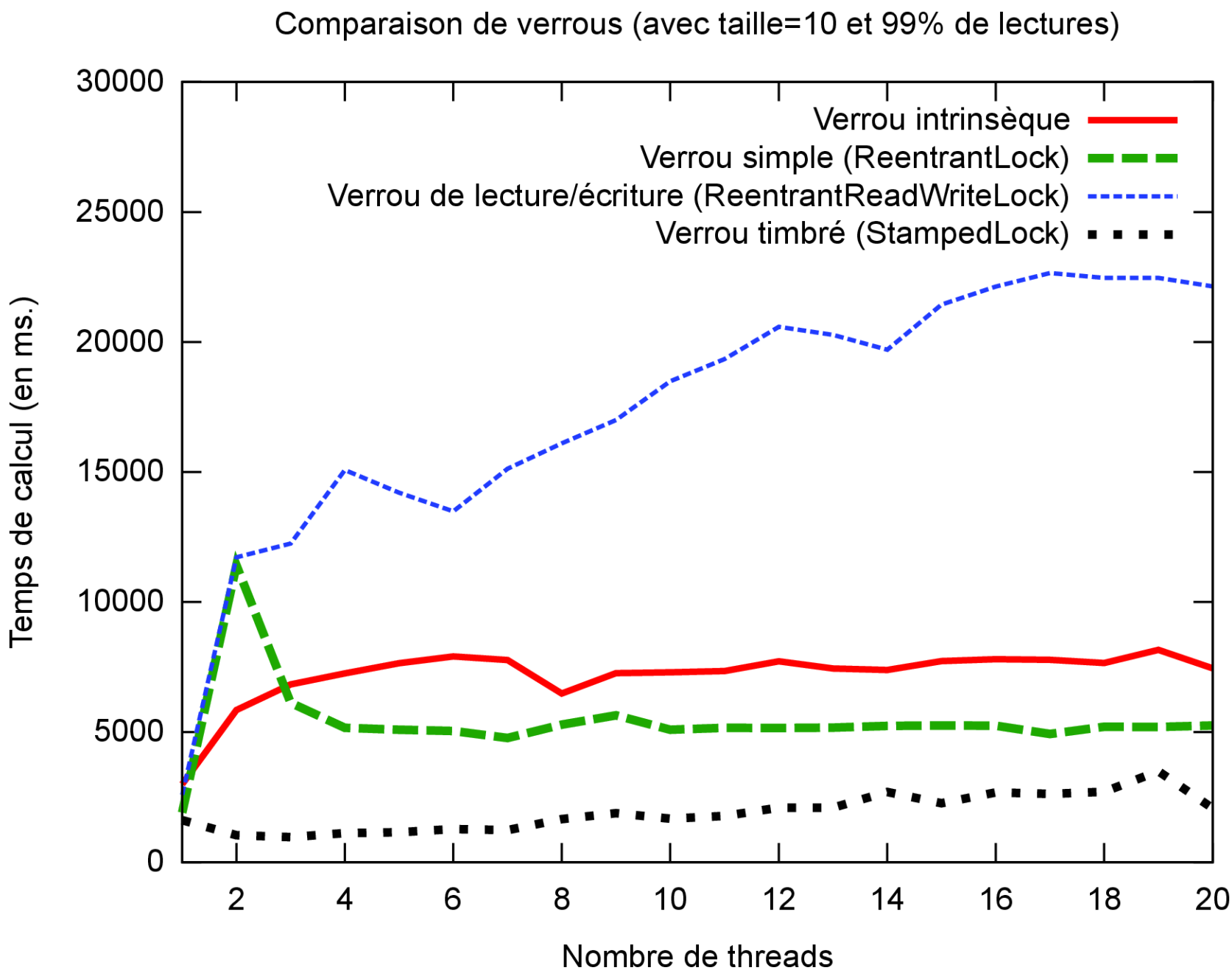
Il faut se méfier des variables ou des objets partagés !

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*.

Performance sur un tableau à 10 élements



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.*

Une opération atomique conditionnelle essentielle

`boolean compareAndSet(valeurAttendue, valeurNouvelle)`

- ⇝ Affecte **atomiquement** la valeur **`valeurNouvelle`** dans l'objet atomique **si** la valeur courante de cet objet est effectivement égale à la valeur **`valeurAttendue`**.

Il faut deviner la valeur courante pour la modifier !

- ⇝ Retourne **`true`** si l'affectation a eu lieu, **`false`** si la valeur 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()`**

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

Problème ABA avec cette manipulation de noeuds (en Java)