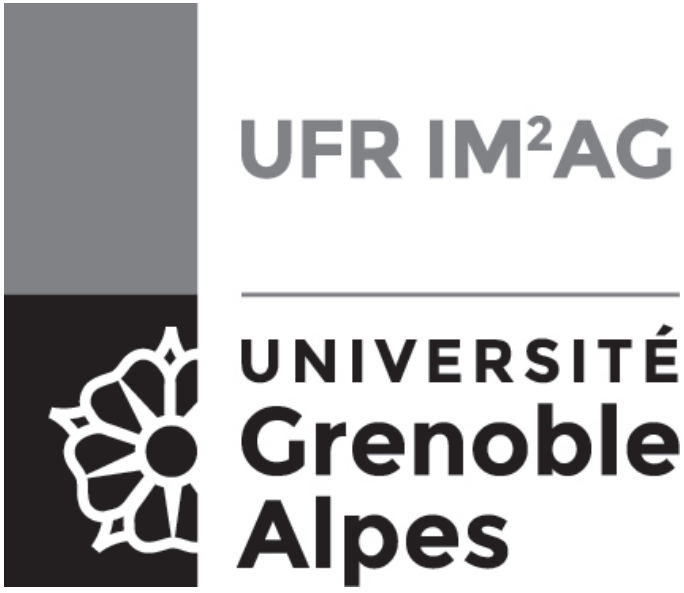


INF362
Programmation multi-threadée

Présentation très incomplète de notions qui seront détaillées et approfondies en M1

Principe

Un fil d'exécution, ou *thread* est

- un contexte d'exécution indépendant (pile, registres)
- qui partage l'espace mémoire du processus (tas, zone statique)

L'utilisation de plusieurs *threads*

- rend la programmation de certains types de serveurs plus simple
- permet de profiter de multiple cœurs dans un même processeur
- peut aider à mieux utiliser les ressources (grâce à l'entrelacement)

Support natif en java

Une classe `Thread` est intégrée au langage

- un *thread* est un objet de la classe `Thread`
- il execute une méthode `run`
  - `Runnable` passé au constructeur
  - ou redéfinie par héritage
- il démarre son exécution via la méthode `start`

Une fois l'exécution d'un *thread* lancée, elle se déroule en concurrence avec celle du *thread* principal

- effectivement en parallèle si la machine dispose de plusieurs cœurs
- selon un entrelacement quelconque sinon

Un programme en Java se termine après la fin de tous les *threads*

Quelques méthodes de `Thread`

`static void sleep(long millis)`

- cause une attente du *thread* qui l'appelle et, typiquement, passe la main à un autre *thread*
- peut lever une `InterruptedException` (causée par `interrupt()`)

`void join()`

- attend la fin du *thread* concerné
- peut lever une `InterruptedException` (causée par `interrupt()`)

`void interrupt()`

- interrompt une attente (`sleep`, `join` ou `wait`) dans le *thread* concerné qui reçoit alors une `InterruptedException`

Quelques exemples (naïfs)

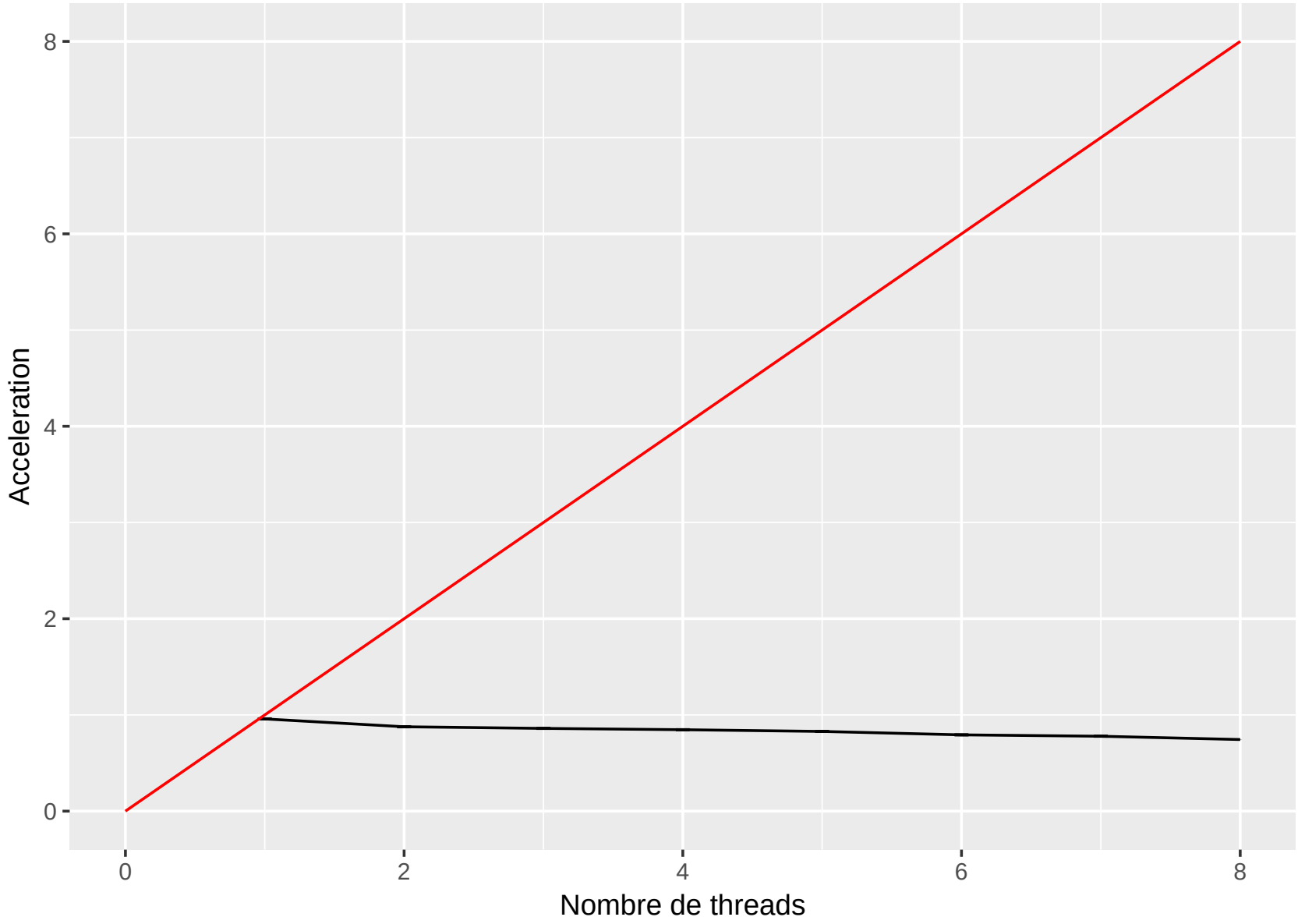
Séquence de phases parallèles

- on démarre tous les *threads* de la première phase
- on attend leur terminaison
- on démarre tous les *threads* de la seconde phase

Tri parallèle

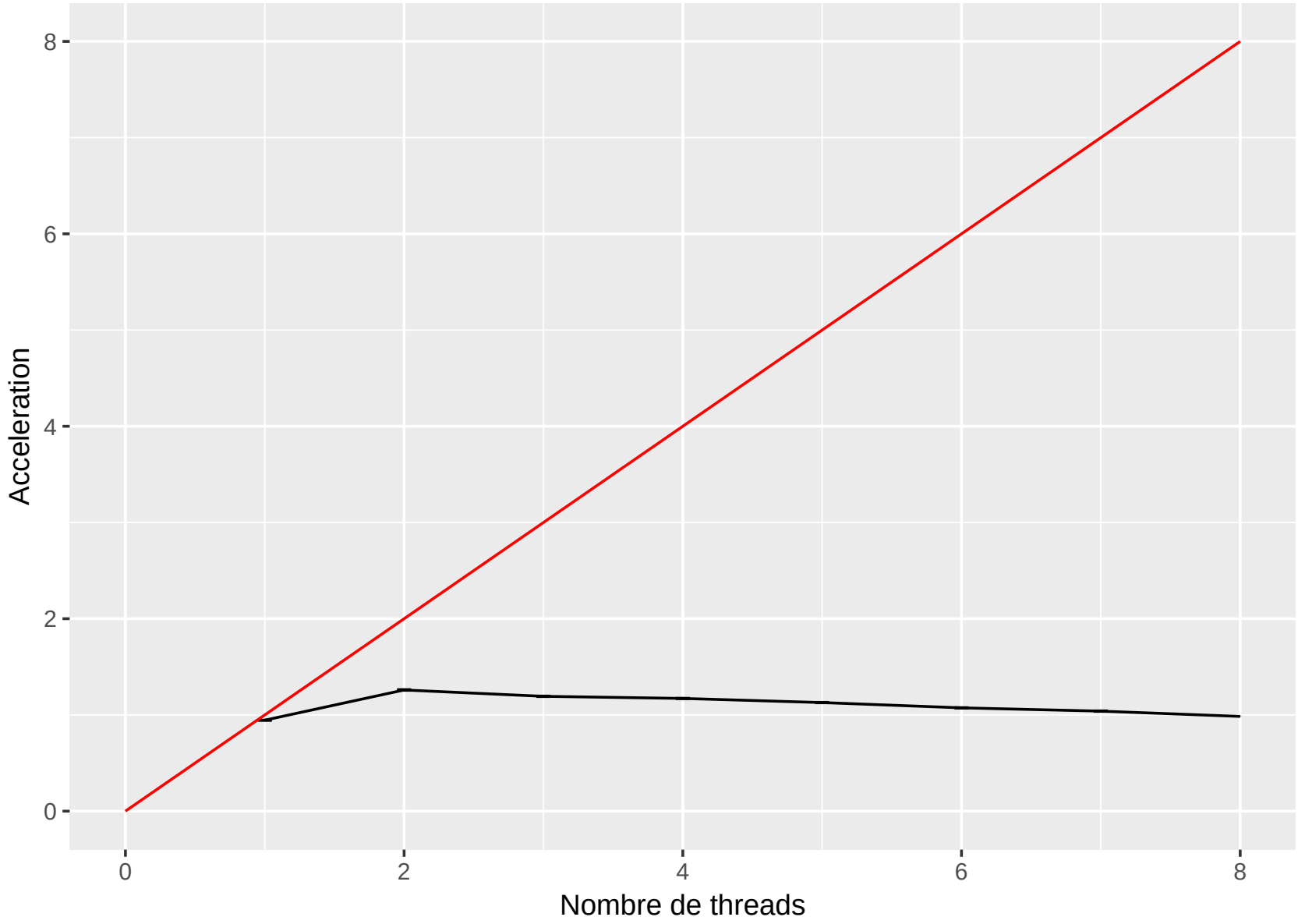
- on découpe le tableau en sous tableaux
- on trie en parallèle les sous tableaux
- on attend la fin de tous les sous tris
- on fusionne les sous tableaux triés

Tri de 10^8 entiers : 1 cœur sans hyperthreading



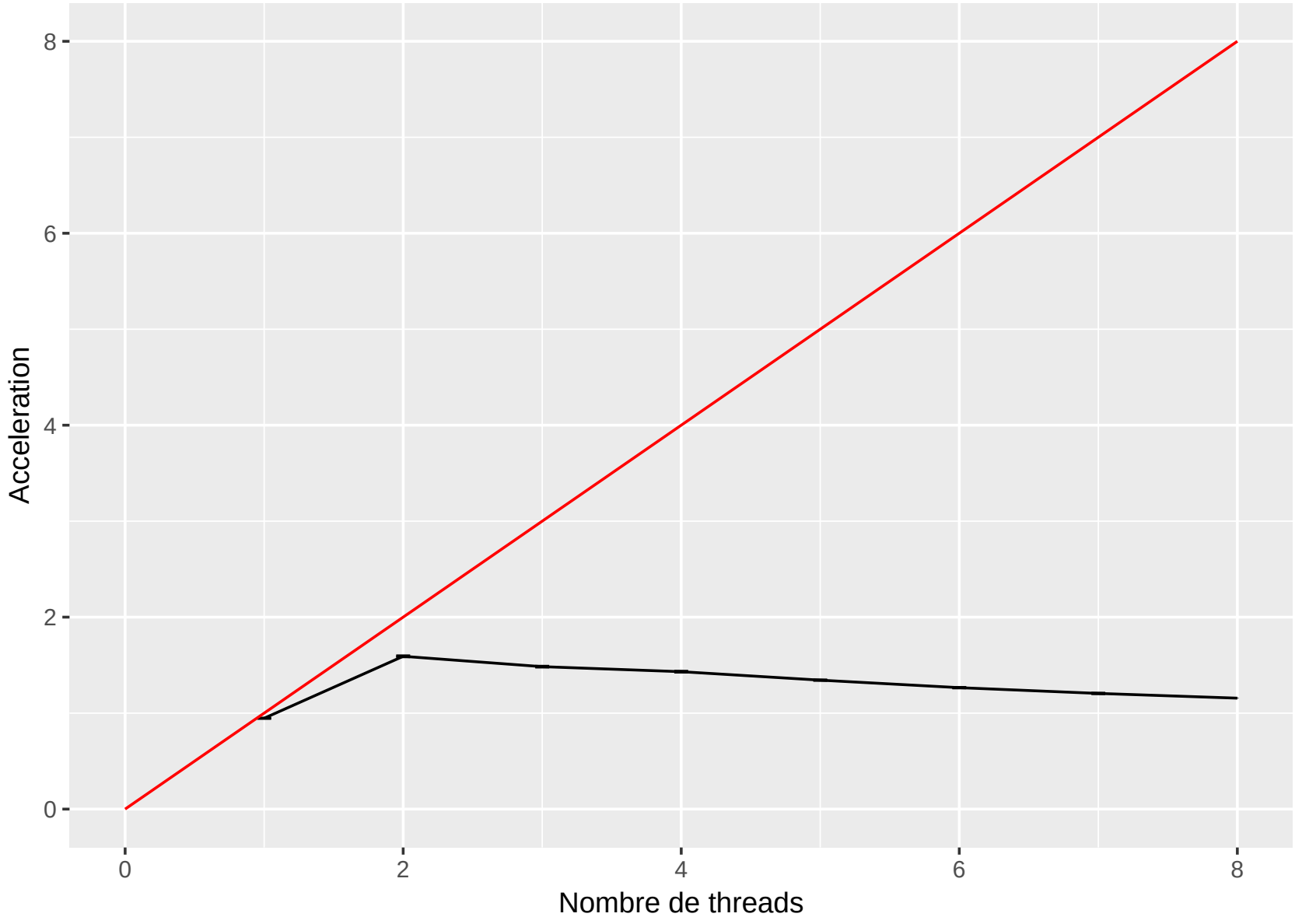
Xeon E3-1230v2, 8Go RAM, Debian 8.7, OpenJDK 2.6.8, tirage uniforme

Tri de 10^8 entiers : 1 cœur avec hyperthreading



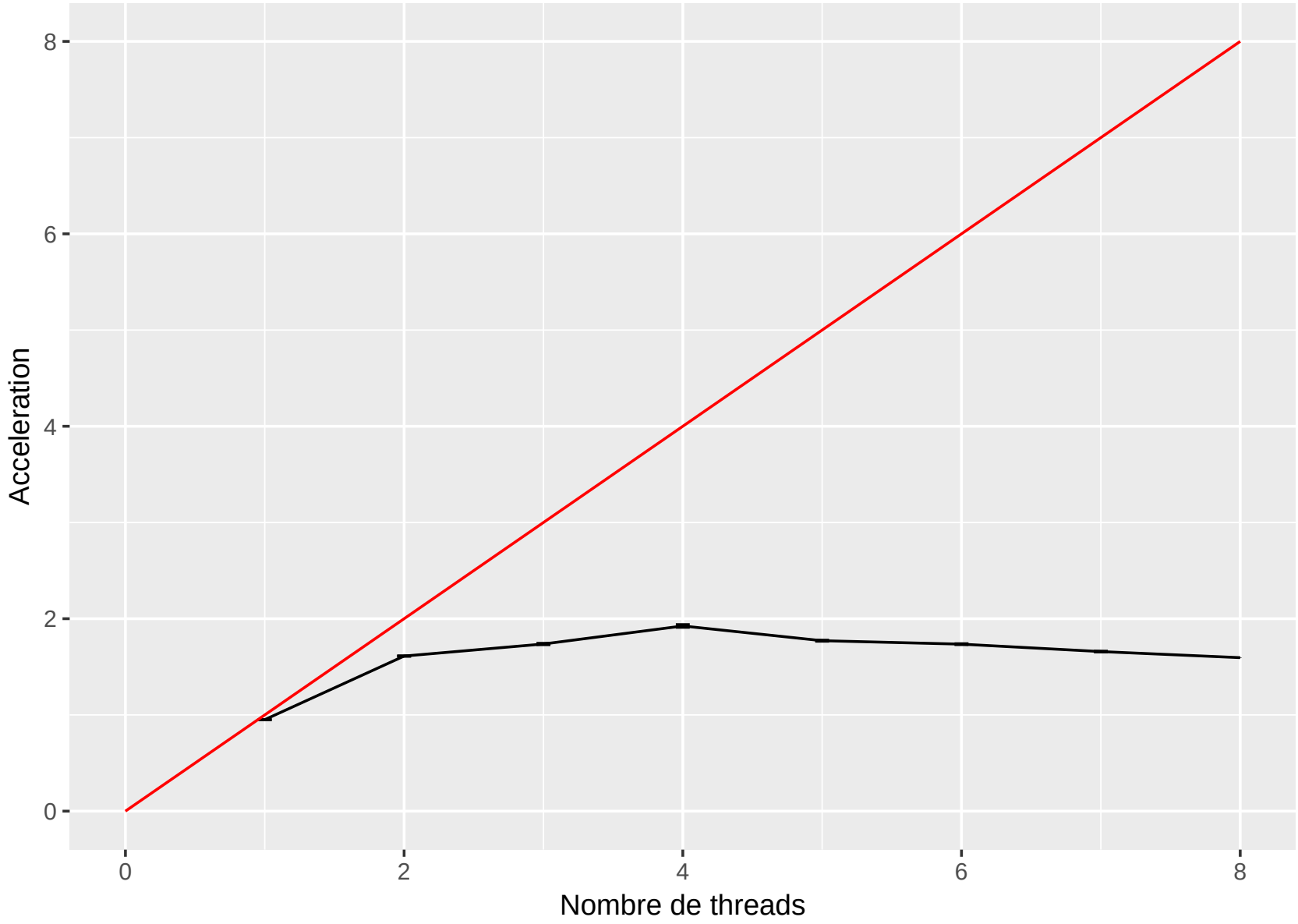
Xeon E3-1230v2, 8Go RAM, Debian 8.7, OpenJDK 2.6.8, tirage uniforme

Tri de 10^8 entiers : 2 cœurs sans hyperthreading



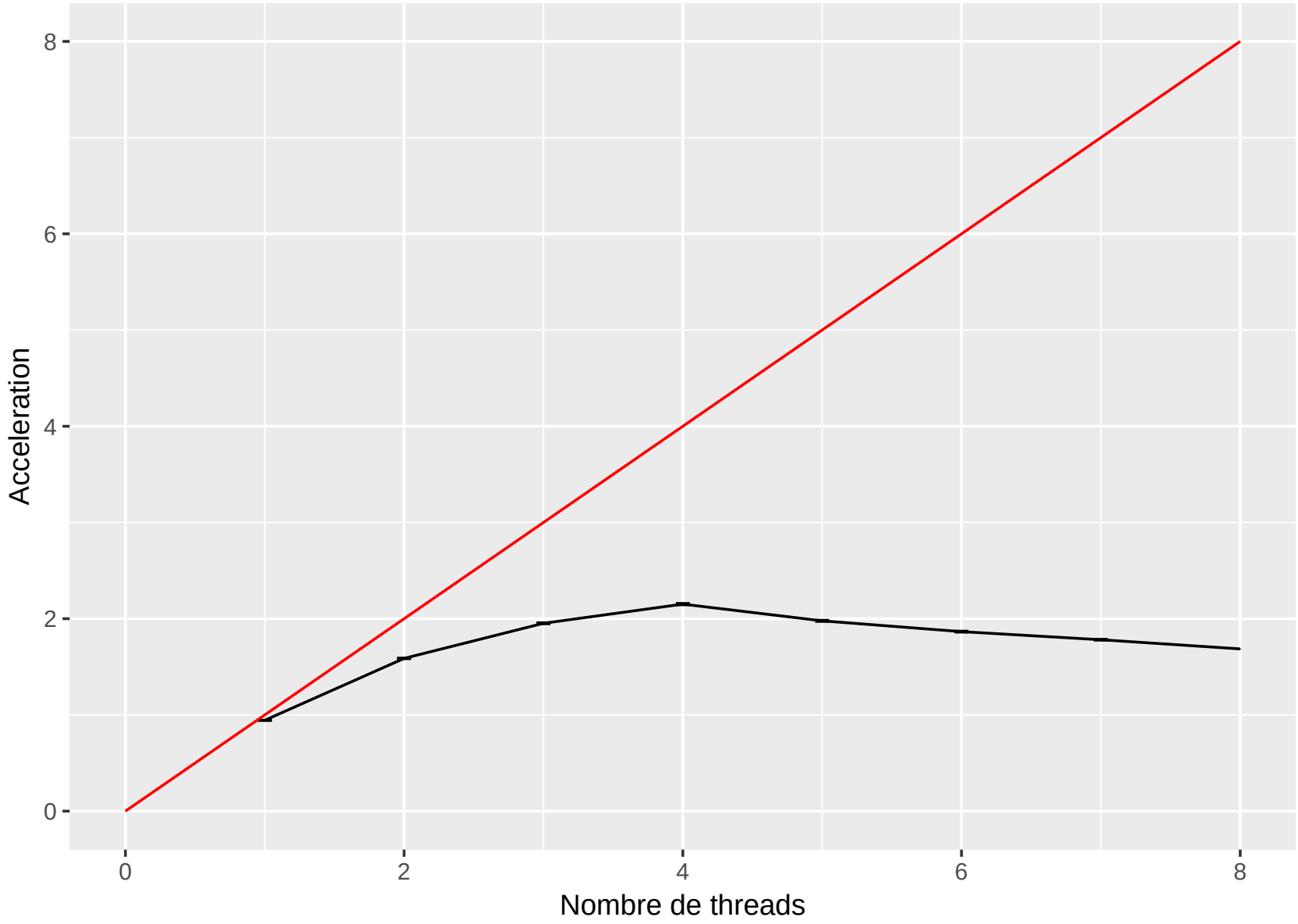
Xeon E3-1230v2, 8Go RAM, Debian 8.7, OpenJDK 2.6.8, tirage uniforme

Tri de 10^8 entiers : 2 cœurs avec hyperthreading



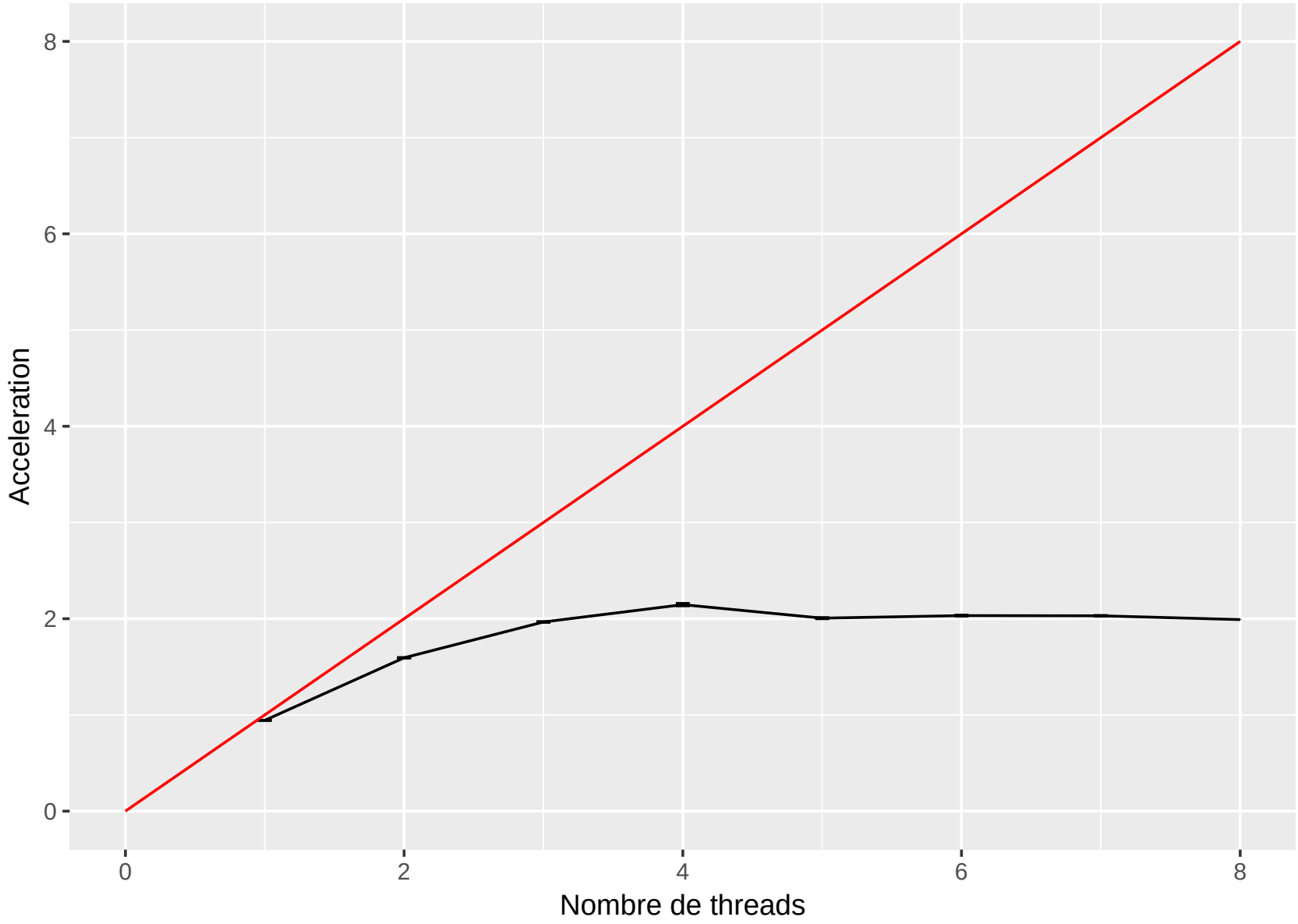
Xeon E3-1230v2, 8Go RAM, Debian 8.7, OpenJDK 2.6.8, tirage uniforme

Tri de 10^8 entiers : 4 cœurs sans hyperthreading



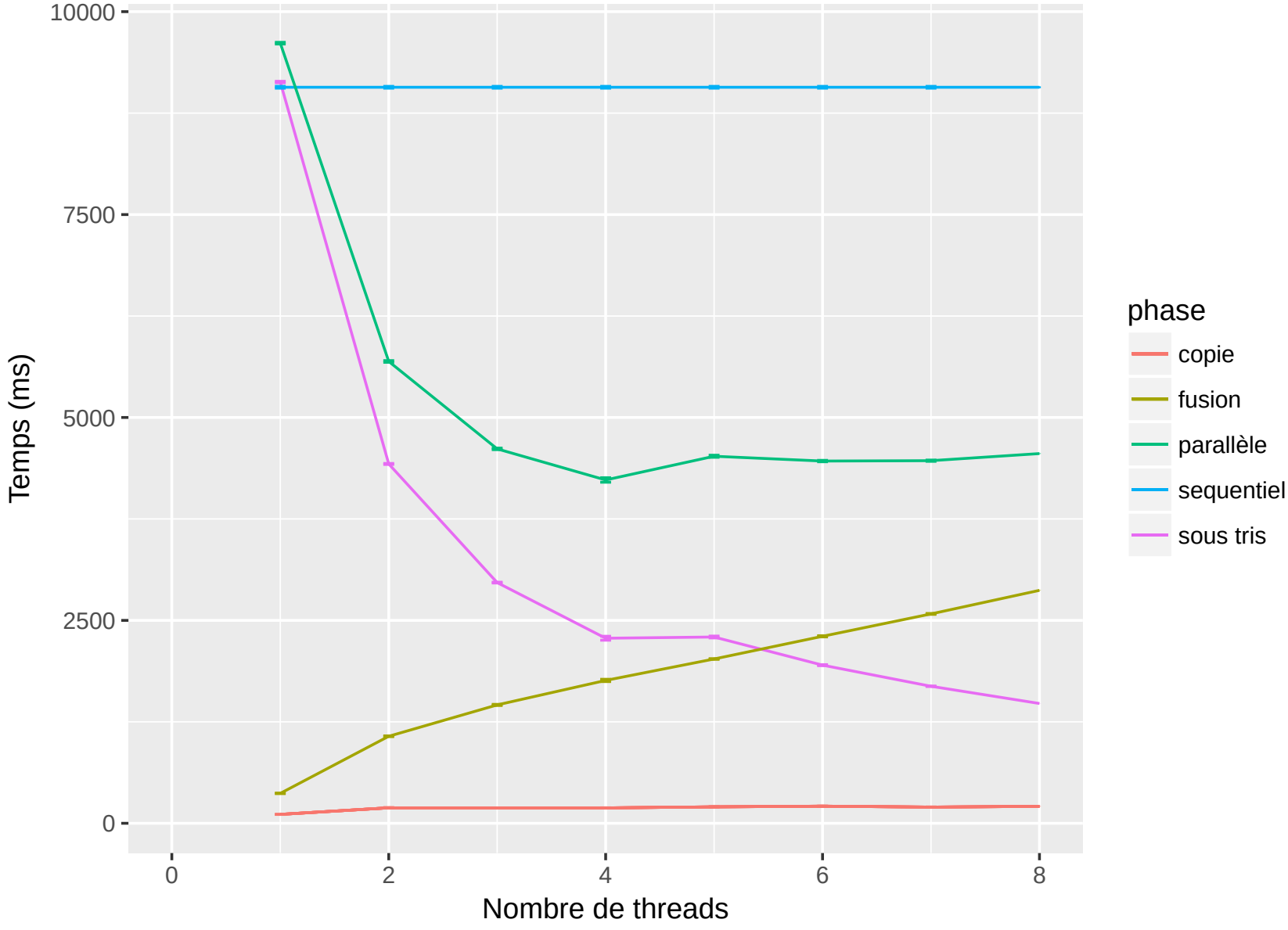
Xeon E3-1230v2, 8Go RAM, Debian 8.7, OpenJDK 2.6.8, tirage uniforme

Tri de 10^8 entiers : 4 cœurs avec hyperthreading



Xeon E3-1230v2, 8Go RAM, Debian 8.7, OpenJDK 2.6.8, tirage uniforme

Tri de 10^8 entiers : 4 cœurs avec hyperthreading



Durée des différentes phases du calcul

La partie parallèle ne décroit pas linéairement

- Moins de travail en parallèle

$$n \times log_2(n) > k \times n/k \times log_2(n/k)$$

  échanges, mais...
- la constante ne change rien asymptotiquement
- surcoût matériel
  - les hyperthreads partagent tout : tous les niveaux de cache, le tampon de réordonnancement, les unités fonctionnelles, ...
  - les cœurs partagent le cache L3 et le contrôleur mémoire
- limite théorique du matériel partagé
  - $10^8 \times log_2(10^8/8) \times 4 \times 2$ octets lus, environ $17,5 Go$.
  - DDR3-1600 dual channel, $25,6 Go/s$ théoriques (latence recouverte, pas de rafraîchissement, cache ignoré)
    $\rightarrow$ 686 ms minimum

La performance dépend de beaucoup de facteurs

Programme

- schéma de parallélisation (ex. éviter les copies)
- travail supplémentaire (ex. fusion en place de Huang et Langston)

Système

- gestion des *threads* (ordonnancement, changement de contexte)
- coût des primitives de synchronisation

Matériel

- nombre de cœurs de calcul
- gestion matérielle de plusieurs *threads*
- capacité du bus mémoire
- cache dédié / partagé
- ...

Problèmes de cohérence mémoire

Les *threads* partagent le même espace mémoire

- des accès concurrents à un même objet sont possibles
- si l'un des accès modifie l'objet, une incohérence est possible

Exemple

```
class Compteur {
    int i;

    Compteur () {
        i=0;
    }

    void incremente() {
        i++;
    }
}
```

`Compteur c;`

Thread 0	Thread 1
`c.incremente();`	`c.incremente();`

Problèmes de cohérence mémoire

Les *threads* partagent le même espace mémoire

- des accès concurrents à un même objet sont possibles
- si l'un des accès modifie l'objet, une incohérence est possible

Exemple

```
class Compteur {
    int i;

    Compteur () {
        i=0;
    }

    void incremente() {
        i++;
    }
}
```

```
        int i = 0;
Thread 0   Thread 1
-------------------
i++;       i++;

i devrait valoir 2
```

Problèmes de cohérence mémoire

Les *threads* partagent le même espace mémoire

- des accès concurrents à un même objet sont possibles
- si l'un des accès modifie l'objet, une incohérence est possible

Exemple

```
class Compteur {
    int i;

    Compteur () {
        i=0;
    }

    void incremente() {
        i++;
    }
}
```

`int i = 0;`

Thread 0	Thread 1
`load r0, =i`	`load r0, =i`
`load r1, [r0]`	`load r1, [r0]`
`add r1, r1, #1`	`add r1, r1, #1`
`str r1, [r0]`	`str r1, [r0]`

`i devrait valoir 2`

Problèmes de cohérence mémoire

Les *threads* partagent le même espace mémoire

- des accès concurrents à un même objet sont possibles
- si l'un des accès modifie l'objet, une incohérence est possible

Exemple

```
class Compteur {
    int i;

    Compteur () {
        i=0;
    }

    void incremente() {
        i++;
    }
}
```

`int i = 0;`

Thread 0	Thread 1
`load r0, =i`	
`load r1, [r0]`	
`(r1 vaut 0)`	`load r0, =i`
	`load r1, [r0]`
	`(r1 vaut 0)`
	`add r1, r1, #1`
	`str r1, [r0]`
`add r1, r1, #1`	`(on stocke 1)`
`str r1, [r0]`	
`(on stocke 1)`	

`i devrait valoir 2`

Pourquoi est-ce que c'est encore plus compliqué ?

Au niveau matériel

- les coeurs de calcul réordonnent les instructions
- le contrôleur mémoire réordonne les accès

Au niveau logiciel

- le compilateur élimine les accès localement inutiles
- le système préempte les *threads* n'importe quand

Ce qu'il faut retenir

- deux accès concurrents au même objet, dont un au moins modifie l'objet, peuvent poser problème
- le système offre des primitives de synchronisation qui s'occupent des détails de plus bas niveau

Les sections critiques

Définition

- portions de code ne devant pas être exécutées en concurrence
- on appelle, par opposition, sections restantes le reste du code

Pour garantir l'exclusion mutuelle entre section critiques

- java utilise un modèle proche de celui des moniteurs et fournit les opérations associées
- le programmeur a la charge de les identifier et de les protéger

Synchronized

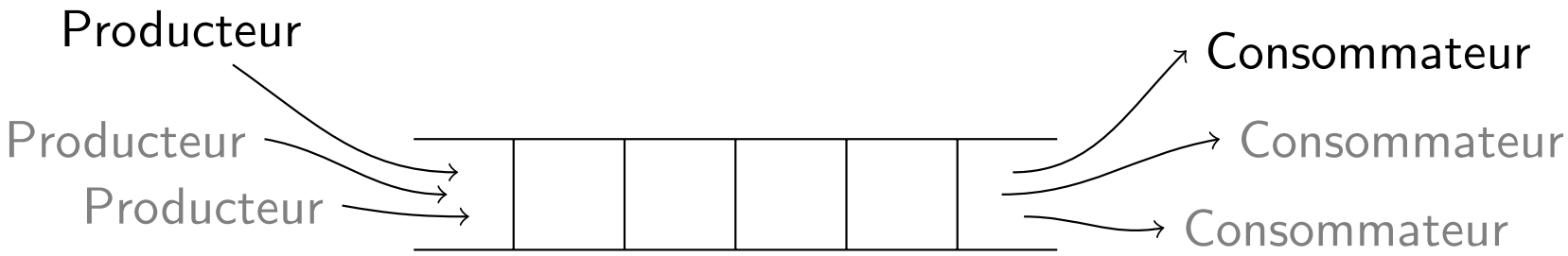
Le mot clé `synchronized` qualifie une méthode qui

- prend un verrou sur l'objet dans lequel elle est appelée
- s'exécute normalement
- libère le verrou pris avant l'exécution

Si le verrou est déjà pris

- le *thread* qui appelle la méthode est bloqué
- lorsque le verrou se libère, un des *threads* bloqués est débloqué et prend le verrou
- l'ordre dans lequel les *threads* prennent le verrou dépend de la machine virtuelle

Le problème du producteur / consommateur



File d'attente partagée

- un ou plusieurs producteurs insèrent des éléments dans la file
- un ou plusieurs consommateurs extraient des éléments de la file
- lorsque la file est vide, les extractions bloquent les consommateurs

Wait / notify

`void wait()`

- libère le verrou sur l'objet et met le *thread* en attente dans cet objet
- lorsqu'une notification arrive
  - libère le *thread* de son attente
  - le bloque sur la prise de verrou, attention
    - le verrou est tenu par le *thread* qui appelle le `notify`
    - rien ne garantit que le *thread* ayant reçu la notification sera le prochain à le récupérer

`void notify()`

- envoie une notification à un *thread* en attente dans l'objet
  - le *thread* est choisi par la machine virtuelle
- ne fait rien si aucun *thread* n'est en attente

`void notifyAll()`

- envoie une notification à tous les *threads* en attente dans l'objet
- ne fait rien si aucun *thread* n'est en attente

Problèmes de synchronisation usuels

Un *thread* arrête de s'exécuter sans libérer les verrous qu'il tient

- impossible dans le modèle recommandé par java
- c'est pour ça que les méthodes `suspend` et `stop` sont dépréciées

Interblocage

- chaque *thread* d'un ensemble
  - détient au moins un verrou et bloque sur un verrou detenu par un autre *thread* de l'ensemble
  - attend une notification et doit envoyer une notification à un autre *thread* de l'ensemble

  le tout formant un cycle d'attente
- difficile à diagnostiquer / débugger
  - souvent non déterministe
  - les traces de debug peuvent changer l'ordonnancement

Approche moins sujette aux incohérences

Tester l'état de l'objet synchronisé en sortant d'un wait

- le *thread* a reçu une notification lui indiquant le changement d'état attendu, mais
- rien ne garantit qu'un autre *thread* ne changera pas cet état entre la notification et la reprise de verrou

Avec l'exemple du producteur / consommateur

1. un consommateur C1 tombe sur une file vide et attend
2. un producteur P produit un objet
3. un autre consommateur C2 arrive et bloque sur le verrou
4. P notifie C1
5. C1 termine son attente et bloque sur le verrou
6. P libère le verrou
7. C2 prend le verrou et consomme l'objet
8. C1 prend le verrou et se retrouve sur une file vide !

Approche moins sujette aux interblocages

Notifier tous les *threads* lors d'un changement d'état

- il n'est pas possible de choisir le *thread* qui reçoit une notification
- s'il y a plusieurs types de *threads*, une notification peut cibler le mauvais destinataire

Bien entendu, on perd en efficacité, les synchronisations coûtent cher...

Avec l'exemple du producteur / consommateur et une file partagée à capacité limitée N

1. 2N consommateurs tombent sur une file vide et attendent
2. N producteurs produisent un objet et notifient N consommateurs
3. N autres producteurs, tombent sur la file pleine et attendent
4. les N consommateurs débloqués consomment un objet et notifient les N consommateurs restants
5. ces derniers N consommateurs tombent sur une file vide et attendent
6. il reste N consommateurs et N producteurs bloqués