

## Parallélisme

ERIC GOUBAULT  
COMMISSARIAT À L'ÉNERGIE ATOMIQUE, SACLAY & CHAIRE  
ÉCOLE POLYTECHNIQUE/THALÈS

1

---

## PROGRAMME

- Les “threads” en JAVA (rappel, et simulation modèles PRAM etc.)
- Les “nouveaux” problèmes liés au parallélisme (problème de coordination, exclusion mutuelle, points morts etc.)
- L’algorithmique parallèle de base (PRAM, complexité, réseaux de tri, algorithmes sur anneau etc.)
- La programmation massivement parallèle (CUDA sur cartes NVIDIA)
- L’ordonnancement de tâches
- Au delà du multitâche (“clusters” de stations, distribution, RMI)
- La tolérance aux pannes

## ORGANISATION

- Complémentaire du cours système, faisant suite au cours d’informatique fondamentale (avec quelques rappels)
- Cours suivis de TPs et de TDs (utilisation des threads JAVA), évaluation sur miniprojet
- <http://www.enseignement.polytechnique.fr/profs/informatique/Eric.Goubault/ParaI11.html>
- Pour d’autres cours (MPRI, PVM/MPI etc.):  
<http://www.enseignement.polytechnique.fr/profs/informatique/Eric.Goubault>

3

---

## PLAN DE CE COURS EN PARTICULIER

- Introduction: modèles du parallélisme, une première approche
- Threads JAVA (rappels)

## INTRODUCTION

Machine **parallèle** = ensemble de **processeurs** qui **coopèrent** et **communiquent**

Historiquement, **réseaux** d'ordinateurs, machines **vectérielles** et faiblement parallèles (années 70 - IBM 360-90 vectoriel, IRIS 80 triprocesseurs, CRAY 1 vectoriel...)

5

## TAXONOMIE DE Tanenbaum:

Quatre types: **SISD**, **SIMD**, **MISD** et **MIMD**.

Basée sur les notions de flot de **contrôle** et flot de **données**.

## TAXONOMIE DE Tanenbaum (1):

Machine **SISD** = **S**ingle **I**nstruction **S**ingle **D**ata  
= Machine Von Neuman

7

## EXEMPLE

Le code suivant,

```
int A[100];  
...  
for (i=1;100>i;i++)  
    A[i]=A[i]+A[i+1];
```

s'exécute sur une machine séquentielle en faisant les additions  $A[1]+A[2]$ ,  $A[2]+A[3]$ , etc.,  $A[99]+A[100]$  à la suite les unes des autres.

TAXONOMIE DE **Tanenbaum** (2):

Machine **SIMD** = **S**ingle **I**nstruction **M**ultiple **D**ata

(1) Parallèle (Connection Machine)

(2) Systolique

En général exécution **synchrone**

9

EXEMPLE

En CM-Fortran sur la Connection Machine-5 avec 32 processeurs,

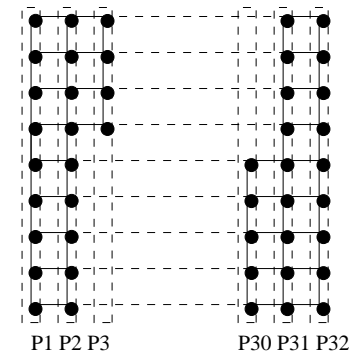
```

    INTEGER I,A(32,1000)
CMF$  LAYOUT A(:NEWS,:SERIAL)
    ...
    FORALL (I=1:32,J=1:1000)
    $    A(I:I,J:J)=A(I:I,J:J)+A(I:I,(J+1):(J+1))

```

Chaque processeur  $i$ ,  $1 \leq i \leq 32$  a en sa mémoire locale une tranche du tableau **A**:  $A(i,1)$ ,  $A(i,2)$ , ...,  $A(i,1000)$ . Il n'y a pas d'interférence dans le calcul de la boucle entre les différentes tranches: tous les processeurs exécutent la même boucle sur sa propre tranche en même temps.

EXEMPLE



11

TAXONOMIE DE **Tanenbaum** (3):

Machine **MISD** = **M**ultiple **I**nstruction **S**ingle **D**ata

Processeurs vectoriels, architectures pipelines

EXEMPLE

Pipelining d'une addition vectorielle,

```
FOR i:=1 to n DO
  R(a+b*i):=A(a'+b'*i)+B(a''+b''*i);
```

A, B et R sont placés dans des registres vectoriels qui se remplissent au fur et à mesure du calcul,

EXEMPLE

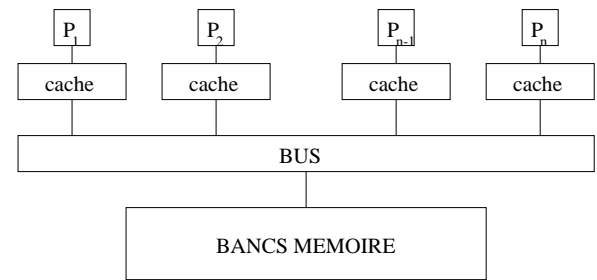
Temps	A ( i )	B ( i )	R ( i )
1	1 . . .	1 . . .	. . . .
2	2 1 . .	2 1 . .	. . . .
3	3 2 1 .	3 2 1 .	. . . .
4	4 3 2 1	4 3 2 1	. . . .
5	5 4 3 2	5 4 3 2	1 . . .
6	6 5 4 3	6 5 4 3	2 1 . .
<i>etc.</i>			

TAXONOMIE DE **Tanenbaum** (4):

Machine **MIMD** = **M**ultiple **I**nstruction **M**ultiple **D**ata

- (1) Mémoire partagée (Sequent)
  - (2) Mémoire locale + réseau de communication (Transputer, Connection Machine, local, par réseau d'interconnexion) - Système réparti
- C'est le cas (2) que l'on va voir plus particulièrement avec RMI. On pourra également simuler le cas (1) (que l'on verra plus avec les threads JAVA).

MÉMOIRE PARTAGÉE



## MÉMOIRE PARTAGÉE

Synchronisation par,

- Barrières de synchronisation,
- Sémaphores : deux opérations P et V.
- Verrou (mutex lock) : sémaphore binaire qui sert à protéger une section critique.
- Moniteurs : construction de haut niveau, verrou implicite.

17

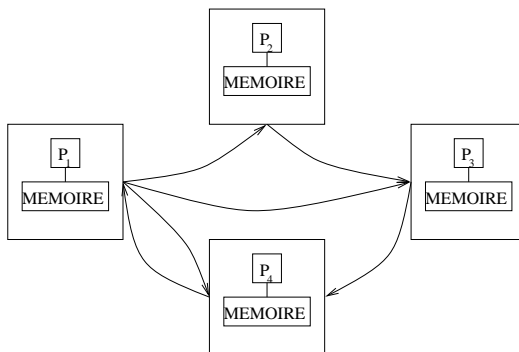
## MACHINE DISTRIBUÉE

Synchronisation et échange d'information par,

- Appel de procédure distribuée (RPC ou RMI en ce qui nous concernera) :
  - réponse synchrone
  - réponse asynchrone
- Envoi/Réception de message asynchrone (tampon de taille limitée ou non); active polling ou gestion à l'arrivée par une procédure **handler**.
- Envoi/Réception de message synchrone : rendez-vous (TD6/CCS)
- Mélange des deux derniers cas.

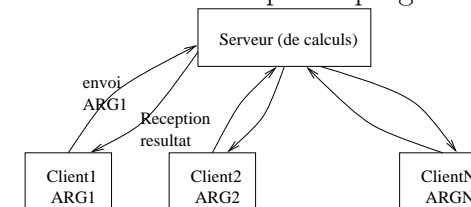
19

## MACHINE DISTRIBUÉE



## EXEMPLE – CLIENTS/SERVEURS

Le protocole RPC (“Remote Procedure Call”) entre machines UNIX avec réseau Ethernet est un exemple de programmation MIMD:



## REMARQUES

**Architectures** plus ou moins adaptées à certains problèmes.

**Gain de temps** espéré: au plus  $N$  (nombre de processeurs).

En pratique, parallélisme difficile à contrôler

21

## PROGRAMMATION PARALLÈLE:

- (1) Langage séquentiel: le compilateur **parallélise** (Fortran...)
- (2) Constructions parallèles **explicites** (Parallel C, Occam...)

## CONSTRUCTIONS PARALLÈLES EXPLICITES

- Création de parallélisme,
  - Itération simultanée sur tous les processeurs (**FOR** parallèle de Fortran),
  - Définition d'un ensemble de **processus** (**COBEGIN**),
  - Création de **processus** (**FORK** de Unix)
- Contrôle du parallélisme,
  - Synchronisation (**Rendez-vous** d'Ada),
  - Passage de messages (synchrones/asynchrones, **broadcast**,...),
  - Section critiques (gestion de la mémoire partagée),
  - Stopper une exécution parallèle (**COEND**, **WAIT** d'Unix)

23

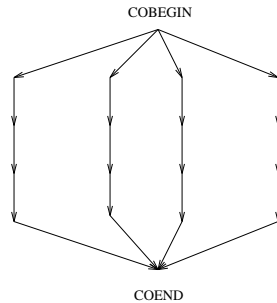
## PROCESSUS (MIMD)

Un **Processus** est une unité de programme **séquentielle** (abstraction d'un processeur) avec,

- sa propre pile de données,
- son compteur ordinal,
- éventuellement sa mémoire locale



## CONSTRUCTIONS PARALLÈLES EXPLICITES (SUITE):



25

## CONSTRUCTIONS PARALLÈLES EXPLICITES (SUITE):

**Synchronisation:** but = assurer qu'à un moment donné tous les processus ont suffisamment avancés leurs calculs.

**Erreurs** possibles: **Interblocage** (deadlock, livelock), **Famine**,... (cours 2 et 3)

### **Exemples:**

- X dit à Y: "Donne moi ton nom et je te dirai le mien". Y dit la même chose à X. Ceci est une situation d'interblocage.
- X et Y veulent tous deux utiliser un objet. X est plus rapide qu'Y et obtient régulièrement l'utilisation de cet objet avant Y qui ne l'obtient jamais. On dit que Y est en situation de famine.

## CONSTRUCTIONS PARALLÈLES EXPLICITES (SUITE):

**MIMD** mémoire partagée,

**Erreurs** possibles: **incohérence** des données.

Partant de  $x = 0$ , on exécute  $x := x + x$  en parallèle avec  $x := 1$ . Par exemple, on a ces trois exécutions possibles:

LOAD x,R1	WRITE x,1	LOAD x,R1	LOAD x,R1
LOAD x,R2	LOAD x,R1	WRITE x,1	LOAD x,R2
WRITE x,1	LOAD x,R2	LOAD x,R2	WRITE x,R1+R2
WRITE x,R1+R2	WRITE x,R1+R2	WRITE x,R1+R2	WRITE x,1
Résultat x=0	Résultat x=2	Résultat x=1	Résultat x=1

Pour y remédier, **sections critiques** et **exclusion mutuelle**.

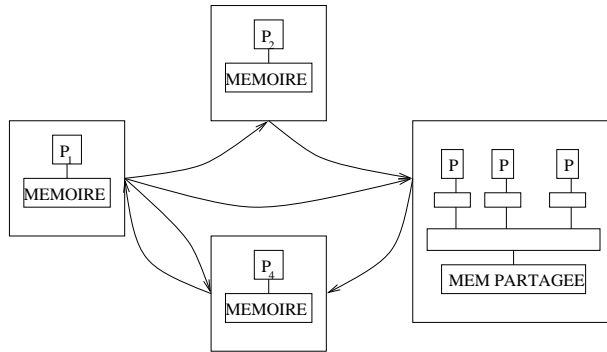
27

RÉSEAUX DE STATIONS

Pas d'horloge globale, mode MIMD, communication par passage de messages, typiquement gérés par PVM, MPI dans le monde numérique, voir,

- <http://www-unix.mcs.anl.gov/mpi/>
  - [http://www.csm.ornl.gov/pvm/pvm\\_home.html](http://www.csm.ornl.gov/pvm/pvm_home.html)
- sur un réseau local ou d'interconnexion.

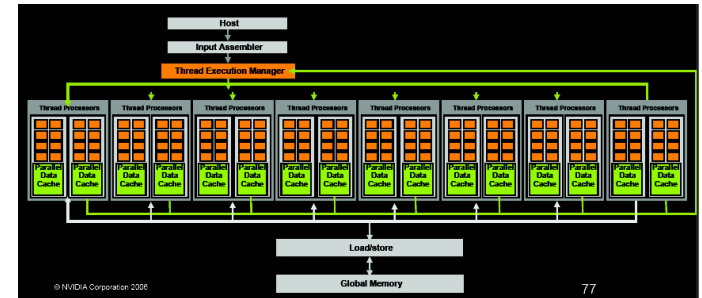
## ARCHITECTURES HYBRIDES



Combinaison sur un réseau local de stations et de machines multi-processeurs (elles-même gérant leur parallélisme interne par mémoire partagée ou par bus ou réseau). Par exemple threads JAVA+RMI.

29

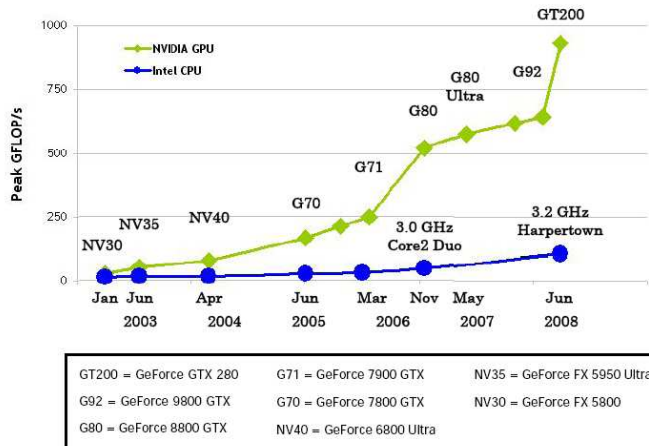
## EXEMPLE: ARCHITECTURE G80



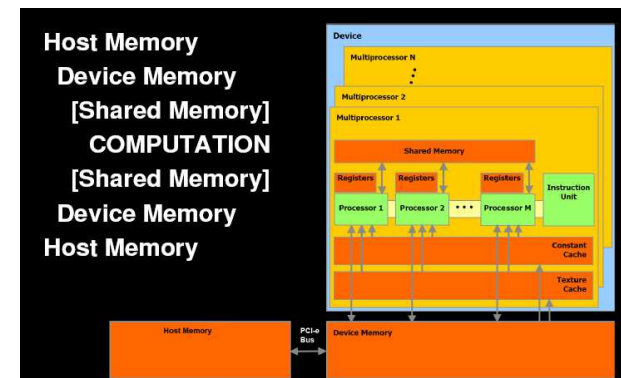
(ici 128 threads procs. pour 4 multi-procs de 32 coeurs) L'hôte peut charger des données sur le GPU, et calculer en parallèle avec le GPU

31

## CARTES NVIDIA



## ARCHITECTURE MÉMOIRE





## ARCHITECTURE

- Organisés en multiprocesseurs (ex. GeForce GTX 260: 6 multi-proc de 32 coeurs=192 coeurs, à 1.242GHz)
- registres 32 bits par multi-proc.
- mémoire partagées par multi-proc.
- un cache à lecture seule (ainsi qu'un cache de textures à lecture seule)

33

## TDs CUDA

- On dispose de cartes GeForce GTX 260 avec 192 coeurs pour les TDs (salles 30 [pays], 35 [os], 36 [voitures])
  - on peut en théorie atteindre 600 GFlops crête!
  - 896 Mo de mémoire
- Ceux qui ont des machines pour jeux vidéos... par exemple GeForce série 8 peuvent télécharger CUDA:
  - <http://www.nvidia.com/cuda> accessible sous windows, vista, linux etc.
  - installer le driver CUDA, le compilateur nvcc et le sdk (avec de multiples exemples)
  - cf. par exemple mon portable avec GeForce 9400M (démon)

## THREADS JAVA

- Concept répandu mais annexe dans d'autres langages de programmation (C ou C++ par exemple),
- Concept intégré dans JAVA (mais aussi ADA)
- Thread = "Thread of Control", processus léger etc.
- un Thread est un programme séquentiel, avec sa propre mémoire locale, s'exécutant en même temps, et sur la même machine virtuelle JAVA, que d'autres threads
- communication à travers une mémoire partagée (ainsi que des méthodes de synchronisations)

35

## UTILITÉ DES THREADS

- Plus grande facilité d'écriture de certains programmes (extension de la notion de fonction)
- Systèmes d'exploitation
- Interfaces graphiques
- Ordinateurs multi-processeurs (bi-pentium etc.)

## CRÉATION

Pour créer un thread, on crée une instance de la classe **Thread**,  
`Thread Proc = new Thread();`

- on peut configurer **Proc**, par exemple lui associer une priorité,
- on peut l'exécuter en invoquant la méthode **start** du thread,

37

## EXÉCUTION

- **start()** va à son tour invoquer la méthode **run** du thread (qui ne fait rien par défaut)
- C'est possible par exemple si on définit **Proc** comme une instance d'une sous-classe de **Thread**, dans laquelle on redéfinit la méthode **run**.
- Les variables locales sont des champs de cette sous-classe.

## EXEMPLE

```
class Compte extends Thread {
    int valeur;

    Compte(int val) {
        valeur = val;
    }

    public void run() {
        try {

            for (;;) {
                System.out.print(valeur + " ");
                sleep(100);
            }
        } catch (InterruptedException e) {
            return;
        }
    }

    public static void main(String[] args) {
        new Compte(1).start();
        new Compte(2000).start();
    }
}
```

39

## EXÉCUTION

L'exécution du programme `main` donne quelque chose comme,

```
> java Compte
1 2000 1 2000 1 1 2000 1
^C
>
```

41

## REMARQUES

- Les entiers `valeur` sont distincts dans les deux threads qui s'exécutent. C'est une variable locale au thread.
- Si on avait déclaré `static int valeur`, cette variable aurait été partagée par ces deux threads (durée de vie=celle des threads).
- Si on avait déclaré `Integer valeur` ("classe enveloppante" de `int`), cette classe aurait été partagée par tous les threads et aurait une durée de vie éventuellement supérieure à celle des threads

## ARRÊT

- méthode `stop()` (dangereux et n'existe plus en JAVA 2),
- méthode `sleep(long)` (suspension du thread pendant un certain nombre de nanosecondes)

43

## MAIS...

- Pas d'héritage multiple en JAVA, donc on ne peut pas hériter de `Thread` et d'une autre classe en même temps,
- Il est alors préférable d'utiliser l'interface `Runnable`.

## L'INTERFACE Runnable

- L'interface **Runnable** représente du code exécutable,
- Elle ne possède qu'une méthode, `public void run();`
- Par exemple la classe **Thread** implémente l'interface **Runnable**.
- Mieux encore, on peut construire une instance de **Thread** à partir d'un objet qui implémente l'interface **Runnable** par le constructeur: `public Thread(Runnable target);`

45

## EXEMPLE

```
class Compte implements Runnable {
    int valeur;

    Compte(int val) {
        valeur = val;
    }

    public void run() {
        try {
            for (;;) {
                System.out.print(valeur + " ");
                Thread.sleep(100);
            }
        }
    }
}
```

## EXEMPLE

```
} catch (InterruptedException e) {
    return;
}
}

public static void main(String[] args) {
    Runnable compte1 = new Compte(1);
    Runnable compte2 = new Compte(2000);
    new Thread(compte1).start();
    new Thread(compte2).start();
}
}
```

47

## NOMMER LES THREADS

- `void setName(String name)` nomme un thread: utile essentiellement pour le debugging (`toString()` renvoie ce nom)
- `String getName()` renvoie le nom du thread.

## INFORMATIONS SUR LES THREADS

- `static Thread currentThread()` renvoie la référence au Thread courant c'est-à-dire celui qui exécute `currentThread()`,
- `static int enumerate(Thread[] threadArray)` place tous les threads existant (y compris le `main()` mais pas le thread ramasse-miettes) dans le tableau `threadArray` et renvoie leur nombre.
- `static int activeCount()` renvoie le nombre de threads.

49

### EXEMPLE

```
class Compt3 extends Thread {
    int valeur;

    Compt3(int val) {
        valeur = val;
    }

    public void run() {
        try {
            for (;;) {
                System.out.print(valeur + " ");
                sleep(100);
            }
        }
    }
}
```

```
} catch (InterruptedException e) {
    return; } }
```

```
public static void printThreads() {
    Thread[] ta = new Thread[Thread.activeCount()];
    int n = Thread.enumerate(ta);
    for (int i=0; i<n; i++) {
        System.out.println("Le thread "+ i + " est
            " + ta[i].getName());
    } }
```

```
public static void main(String[] args) {
    new Compt3(1).start();
    new Compt3(2000).start();
    printThreads(); } }
```

51

## EXÉCUTION

```
% java Compt3
1 2000 Le thread 0 est main
Le thread 1 est Thread-2
Le thread 2 est Thread-3
1 2000 1 2000 1 2000 1 2000 2000
1 1 2000 2000 1 1 2000 2000 1 1
2000 2000 1 1 2000 2000 1 1 2000
2000 1 1 2000 2000 1 1 ^C
%
```

## COMPLÉMENT: PRIORITÉS ET ORDONNANCEMENT

- `void setPriority(int priority)` assigne une priorité au thread donné
- `int getPriority()` renvoie la priorité d'un thread donné
- `static void yield()`: le thread courant rend la main, ce qui permet à la machine virtuelle JAVA de rendre actif un autre thread de même priorité

53

## ETATS D'UN THREAD

Un thread peut être dans l'un des 4 cas suivants:

- l'état initial, entre sa création et `start()`.
- l'état prêt, immédiatement après `start`.
- l'état bloqué: thread en attente, d'un verrou, d'un socket, quand il est suspendu (`sleep(long)`)
- l'état terminaison, quand `run()` se termine ou quand on invoque `stop()` (à éviter si possible... n'existe plus en JAVA 2).

## ETATS D'UN THREAD

- Détermination de l'état: `isAlive()`, qui renvoie un booléen indiquant si un thread est encore vivant, c'est à dire s'il est prêt ou bloqué
- On peut aussi interrompre l'exécution d'un thread qui est prêt (passant ainsi dans l'état bloqué). `void interrupt()` envoie une interruption au thread spécifié; si pendant `sleep`, `wait` ou `join`, lèvent une exception `InterruptedException`.

55

## ETATS D'UN THREAD

- `void join()` attend terminaison d'un thread. Egalement: `void join(long timeout)` attend au maximum `timeout` millisecondes.

```
public class ... extends Thread {
    ...
    public void stop() {
        t.shouldRun=false;
        try {
            t.join();
        } catch (InterruptedException e) {} } }
```

## PRIORITÉS

- Priorité = nombre entier, qui plus il est grand, plus le processus est prioritaire.
- `void setPriority(int priority)` assigne une priorité et `int getPriority()` renvoie la priorité.
- La priorité peut être maximale: `Thread.MAX_PRIORITY`, normale (par défaut): `Thread.NORM_PRIORITY` (au minimum elle vaut 0).

57

## PROCESSUS DÉMONS

On peut également déclarer un processus comme étant un démon ou pas:

```
setDaemon(Boolean on);  
boolean isDaemon();
```

“support” aux autres (horloge, ramasse-miettes...). Il est détruit quand il n’y a plus aucun processus utilisateur (non-démon) restant

## ORDONNANCEMENT TÂCHES (CAS 1 PROCESSEUR)

- Le choix du thread JAVA à exécuter (partiellement): parmi les threads qui sont prêts.
- Ordonnanceur JAVA: ordonnanceur préemptif basé sur la priorité des processus.
- “Basé sur la priorité”: essaie de rendre actif le(s) thread(s) prêt(s) de plus haute priorité.
- “Préemptif”: interrompt le thread courant de priorité moindre, qui reste néanmoins prêt.

59

## ORDONNANCEMENT DE TÂCHES

- Un thread actif qui devient bloqué, ou qui termine rend la main à un autre thread, actif, même s’il est de priorité moindre.
- La spécification de JAVA ne définit pas précisément la façon d’ordonner des threads de même priorité, par exemple:
  - “round-robin” (ou “tourniquet”): un compteur interne fait alterner l’un après l’autre (pendant des périodes de temps prédéfinies) les processus prêts de même priorité → assure l’équité; aucune *famine* (plus tard...)
  - Ordonnement plus classique (mais pas équitable) thread actif ne peut pas être préempté par un thread prêt de même priorité. Il faut que ce dernier passe en mode bloqué. (par `sleep()` ou plutôt `static void yield()`)

EN FAIT...

L'ordonnement dépend aussi bien de l'implémentation de la JVM que du système d'exploitation sous-jacent; 2 modèles principaux:

- “green thread”, c'est la JVM qui implémente l'ordonnement des threads qui lui sont associés (donc pas d'exploitation multi-processeur - souvent UNIX par défaut).
- “threads natifs”, c'est le système d'exploitation hôte de la JVM qui effectue l'ordonnement des threads JAVA (par défaut Windows).