INF 560
Calcul Parallèle et Distribué
Cours 1

Eric Goubault et Sylvie Putot

Ecole Polytechnique

8 décembre 2014

Programme

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

Organisation

- Fait suite aux cours INF311/321 et éventuellement INF431 (avec quelques rappels sur les threads JAVA)
- Cours suivis de TPs et de TDs (utilisation des threads JAVA, RMI et CUDA), évaluation sur miniprojet
- `http://www.enseignement.polytechnique.fr/profs/informatique/Sylvie.Putot/INF560/`

Plan de ce cours en particulier

- Introduction: architectures parallèles et modèles du parallélisme, une première approche
- Calcul sur GPU (Graphics Processing Unit), une introduction rapide
- Threads JAVA (rappels, pour la plupart, utiles pour RMI et l'algorithmique PRAM du cours 2!)

Motivations

Pourquoi le calcul parallèle:

- Résoudre plus rapidement un problème donné
- Traiter de plus gros volumes de données
- Problèmes specifiques (bases de données réparties etc)
- Tirer le meilleur parti du matériel récent (systématiquement multi-coeur)

Quelques applications:

- Sciences de l'univers et de l'environnement, prédiction climatique
- Biologie et biotechnologies
- Ingénierie (ex dynamique des fluides, modélisation/simulation), logistique/optimisation
- Finance, science des données
- Cryptographie

- Machine **parallèle** = ensemble de **processeurs** qui **coopèrent** et **communiquent**
- Historiquement, **réseaux** d'ordinateurs (Arpanet 1969-72), machines **vectorielles**:
  - CDC 6600 (Seymour Cray, 1964): premier ordinateur avec un processeur superscalaire (parallélisme d'instructions)
  - ILLIAC IV (1965): processeur vectoriel (échec commercial)
  - CRAY 1 (1976): processeur vectoriel (succès commercial)

Taxonomie de **Flynn** (1966):

Basée sur les notions de flot d'**instructions** et flot de **données**.

Quatre types:

- **SISD** - **S**ingle **I**nstruction (Stream) **S**ingle **D**ata (Stream)
  - Machine de Von Neumann
- **SIMD** - **S**ingle **I**nstruction **M**ultiple **D**ata
  - Calculateurs vectoriels, GPU (SPMD)
- **MISD** - **M**ultiple **I**nstruction **S**ingle **D**ata
  - Peu d'utilité pratique: redondance dans applications critiques, [architectures pipeline ?]
- **MIMD** - **M**ultiple **I**nstruction **M**ultiple **D**ata
  - Multiprocesseurs, clusters

SISD - Single Instruction Single Data

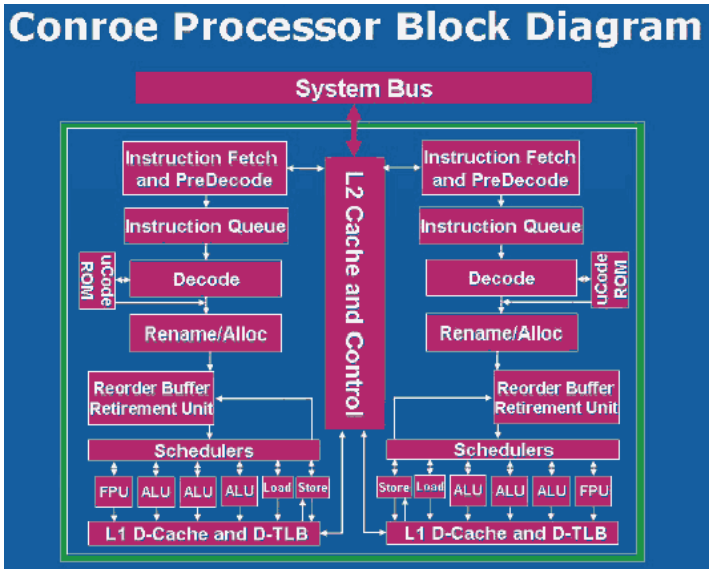
Un ordinateur séquentiel qui n'exploite aucun parallélisme (architecture de Von Neumann)
Par exemple,

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

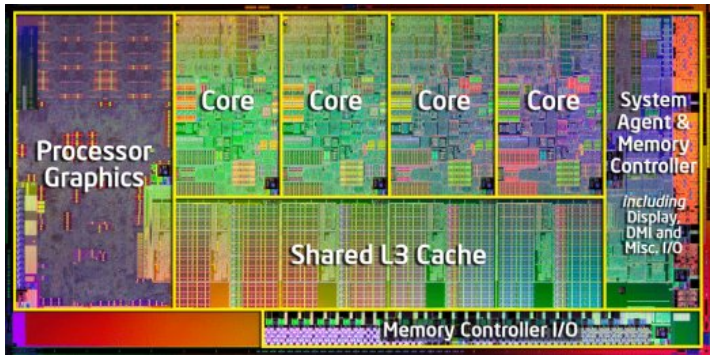
s'exécute sur une machine séquentielle en faisant les additions `A[1]+A[2], etc., A[99]+A[100]` successivement

Mais...

N'existe quasiment plus



(Intel Conroe ∼2006)



(Archi multicoeur Intel)

MISD = Multiple Instruction Single Data

- Plusieurs CPU qui opèrent indépendamment sur le même flot d'instruction
- Peu utilisé - sauf si on considère les architectures pipeline comme MISD (?)

Pipeline

L'exécution d'une instruction complexe telle qu'une addition ou une multiplication entière ou flottante est découpée en blocs ou étages élémentaires, qui peuvent être exécutés par des composantes différentes d'une unité fonctionnelle.

Tous les processeurs modernes sont pipelinés

Pipeline: exemple $C[i] = A[i] \times B[i]$

- **5 étages:** comparaison des exposants (E1), alignement des opérandes (E2), add. des expos. et mult. des mantisses (E3), facteur de normalisation (E4), résultat normalisé (E5).
- Si chaque étape met 1 cycle pour s'exécuter, il faut 5 cycles pour exécuter une instruction, 10 pour 2 instructions:

$A[1], B[1]$ $A[2], B[2]$	$E1$	$E2$	$E3$	$E4$	$E5$	$E1$	$E2$	$E3$	$E4$	$E5$	
resu					$C[1]$					$C[2]$	..

- Avec le pipeline, une instruction par cycle à partir du 5eme:

$A[1], B[1]$	$E1$	$E2$	$E3$	$E4$	$E5$					
$A[2], B[2]$		$E1$	$E2$	$E3$	$E4$	$E5$				
$A[3], B[3]$			$E1$	$E2$	$E3$	$E4$	$E5$			
$A[4], B[4]$				$E1$	$E2$	$E3$	$E4$	$E5$		
$A[5], B[5]$					$E1$	$E2$	$E3$	$E4$	$E5$	
resu					$C[1]$	$C[2]$	$C[3]$	$C[4]$	$C[5]$	...

SIMD = Single Instruction Multiple Data

- Single Instruction: tous les processeurs exécutent la même instruction en même temps
- Multiple Data: les processeurs opèrent sur des éléments différents
- En général exécution synchrone
- Typiquement, processeurs vectoriels:
  - effectuent chaque instruction en parallèle sur un groupe de données (plusieurs pipelines en parallèle)
- Ou encore calcul sur GPU (Graphics Processing Units)
  - SPMD (Single Program Multiple Data) plus souple: structures conditionnelles
- Modèle approprié aux problèmes avec de grands volumes de données régulières (supercalculateurs)
- La plupart des processeurs modernes permettent des opérations vectorielles (jeu d'instructions SSE pour architecture x86)

Exemple calcul vectoriel

En CM-Fortran 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 P_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.

Deuxième révolution : vers le MIMD

- Fin des années 80 - début des années 90 : recul du vectoriel + apparition des microprocesseurs
  - Parallélisme massif SIMD (Connection Machine) (CM-1 65536 processeurs)
  - Apparition des systèmes à mémoire partagée avec un nombre raisonnable de processeurs (Sequent Balance 8000 en 1984: 12 processeurs, Balance 21000 en 1986: 30 processeurs)
  - Apparition des systèmes distribués avec un grand nombre de processeurs, mais sans mémoire partagée (Transputer, CM-5)

MIMD = Multiple Instruction Multiple Data

- Multiple instruction: chaque processeur peut effectuer un flot d'instructions différent
- Multiple data: chaque processeur peut travailler sur un flot de données (ou une espace mémoire) différent
- Synchrone ou asynchrone
- Multi-coeurs, clusters de PC, etc

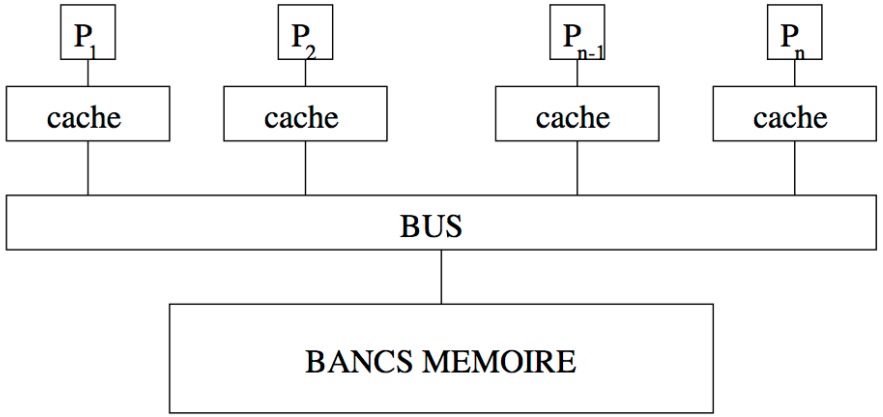
Architecture mémoire

(1) Mémoire partagée

(2) Mémoire distribuée ou Système réparti = mémoire locale + réseau de communication

C'est le cas (2) que l'on va voir plus particulièrement avec RMI.
On pourra également simuler le cas (1) (threads JAVA).

Systèmes à mémoire Partagée



Essentiellement deux classes de systèmes

- UMA (Uniform Memory Access) - Sequent Balance 1984
  - processeurs identiques, accès en temps égal à la mémoire
  - par exemple processeurs vectoriels
  - systèmes aussi appelés SMP (Symmetric Shared-memory Multi-Processeurs)
- NUMA (Non-Uniform Memory Access) - Sequent à partir de 1996
  - les processeurs n'ont pas un temps d'accès égal à toute la mémoire (dépend de la proximité)
  - souvent réalisé en liant physiquement des SMP
  - chaque SMP a accès à la mémoire des autres, mais plus lent

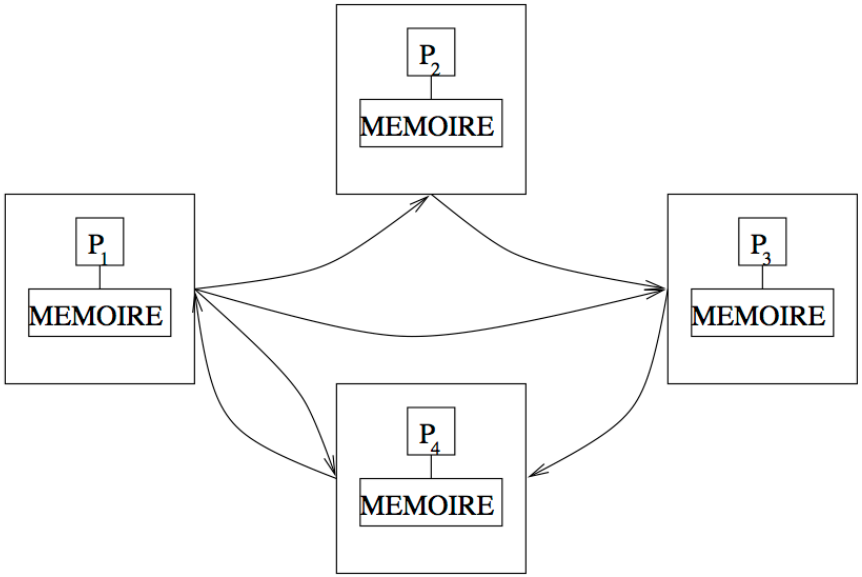
Systèmes à mémoire Partagée

- Tous les processeurs accèdent à toute la mémoire
- La modification de mémoire effectuée par un processeur est visible par tous les autres
- Ils s'exécutent indépendamment mais l'accès mémoire nécessite une synchronisation

Synchronisation

- 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 (wait()/notify()).

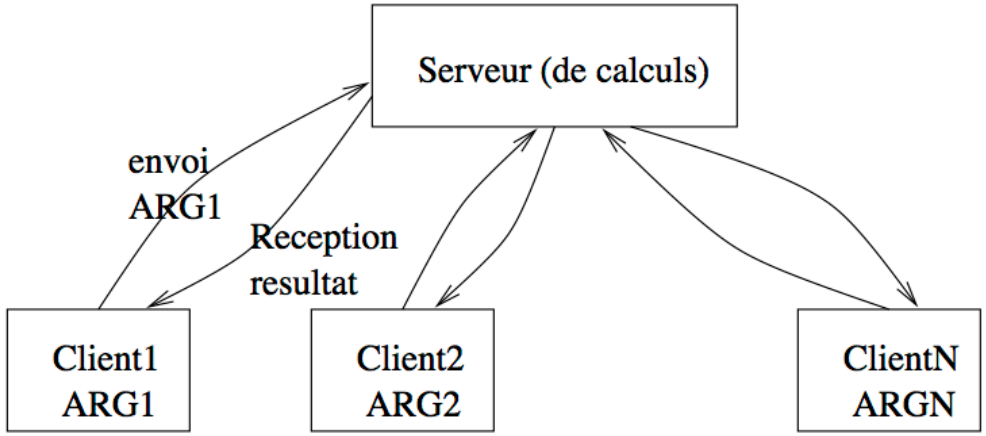
Système à mémoire distribuée



- Les processeurs ont leur propre mémoire, pas de notion de zone d'adressage globale
- Réalisation aisée / peu coûteuse par cluster de machines
- Synchronisation et échange d'information par le programmeur
  - Appel de procédure distribuée (RPC ou RMI en ce qui nous concernera), bloquant ou non bloquant
  - Envoi/réception de message asynchrone (tampon); active polling ou gestion à l'arrivée par une procédure `handler`.
  - Envoi/réception de message synchrone: rendez-vous

Exemple – Clients/Serveurs

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



Remarques

- Architectures parallèles plus ou moins adaptées à certains problèmes.
  - Caractérisées par l'accès à la mémoire : partagée, distribuée, partagée distribuée
- Gain de temps
  - En théorie, gain espéré au plus N (nombre de processeurs)
  - En pratique, selon la conception de l'architecture (accès mémoire, communications entre coeurs, etc.) les performances peuvent être bien en deçà
- Bonnes performances difficiles à atteindre:
  (1) Parallélisation automatique par le compilateur (Fortran): performances limitées
  (2) Constructions parallèles **explicites** (Parallel C, Occam, Java, CUDA...): mise au point difficile

Constructions parallèles explicites

- **Synchronisation**: but = assurer qu'à un moment donné tous les processus ont suffisament avancés leurs calculs.
- **Erreurs** possibles: **Interblocage** (deadlock, livelock), **Famine**,...

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.

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 4 exécutions possibles:

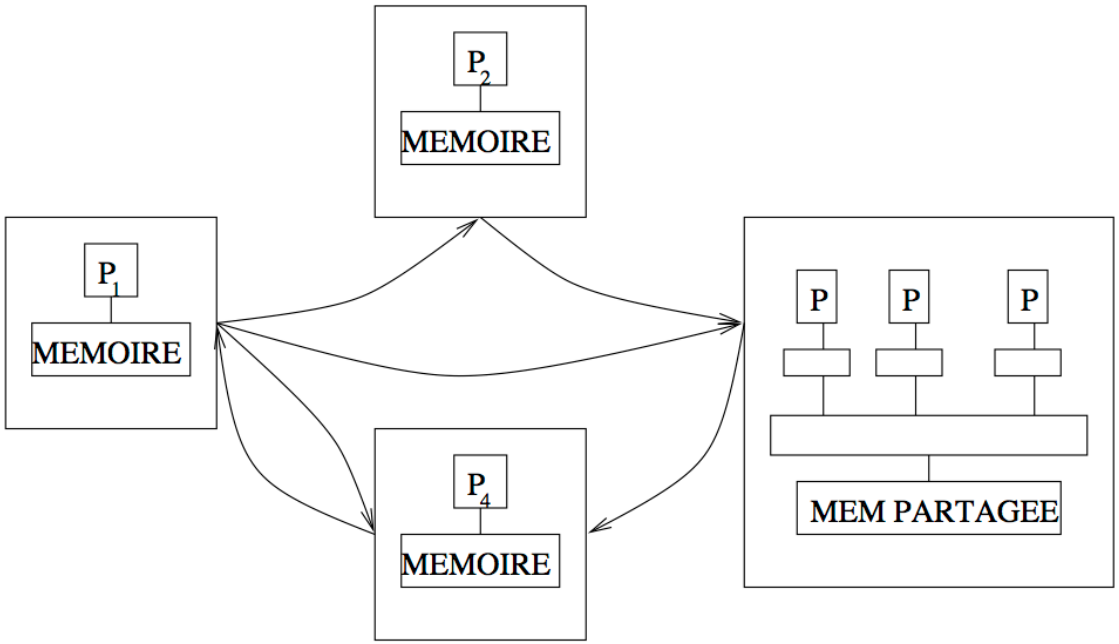
`LD x,R1`	`WT x,1`	`LD x,R1`	`LD x,R1`
`LD x,R2`	`LD x,R1`	`WT x,1`	`LD x,R2`
`WT x,1`	`LD x,R2`	`LD x,R2`	`WT x,R1+R2`
`WT x,R1+R2`	`WT x,R1+R2`	`WT x,R1+R2`	`WT x,1`
Rés x=0	Rés x=2	Rés x=1	Rés x=1

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

Supercalculateurs : tendance actuelle

- On arrive aux limites de la loi de Moore (nombre de transistors par puce double tous les 18 mois)
  - On n'arrive plus à augmenter la fréquence des processeurs "classiques" (gravure trop fine, dissipation thermique explose)
  - L'accès à la mémoire devient pénalisant
- Début des années 2000 : systèmes hybrides à mémoire partagée/distribuée
  - Grappes ou clusters : noeuds (= ensemble de processeurs partageant de la mémoire) interconnectés par un réseau
  - Evolution de la taille des noeuds : de plus en plus gros
- Apparition des processeurs multicores
  - Sur une même puce plusieurs processeurs "classiques" (coeurs) + plusieurs caches, accès concurrents à la mémoire, etc.
  - Reproduction d'un noeud SMP au niveau du processeur
- Apparition d'accélérateurs GPU
  - Meilleur rapport consommation électrique/puissance calcul
  - Difficile à programmer (efficacement)

Architectures hybrides (MIMD)



Combinaison sur un réseau local de stations et de machines multiprocesseurs (elles-même gérant leur parallélisme interne par mémoire partagée ou par bus ou réseau).

- Pas d'horloge globale, mode MIMD, communication par passage de messages,
- Par exemple threads JAVA+RMI
- Ou encore clusters de PC avec GPU

En pratique: quelques clés de la performance d'une application

- Trouver des tâches qui peuvent s'exécuter en parallèle
- Granularité: trouver la bonne taille pour les tâches parallèles
- Localité: les accès mémoire ou communications coûtent plus que le calcul
- Régularité des données
- Equilibrage de charge: ne pas avoir des processeurs qui ne font rien
- Coordination et synchronisation: ne pas bloquer inutilement

Le TOP 500 (http://www.top500.org)

Unités et ordres de grandeur

- Flop/s: Opérations floating point (généralement double précision) par seconde
  - 1 TFlop/s $= 10^{12}$ flop/sec,
  - 1 PFlop/s $= 10^{15}$ flop/sec
  - Machine la plus puissante du TOP500 $\approx 33,8 PFlop/s$
- Bytes: taille des données (un double float = 8 Bytes)

Mesure de performance Top500

- La mesure retenue = performance de Linpack (résolution de système linéaire plein par pivot de Gauss, ultra-optimisée)
- Coût de calcul du pivot de Gauss $\approx n^3/3$
  - pour $n = 10^5$ coût 3.3×10^{14} Flop, $n = 10^6$ coût 3.310^{17} Flop
  - si les transferts de données ne coûtent rien, ce qui est loin d'être le cas (une matrice $10^6 \times 10^6$ a 10^{12} élém = 8 TBytes...)

Le TOP 500 (http://www.top500.org)

TOP 10 Sites for November 2014

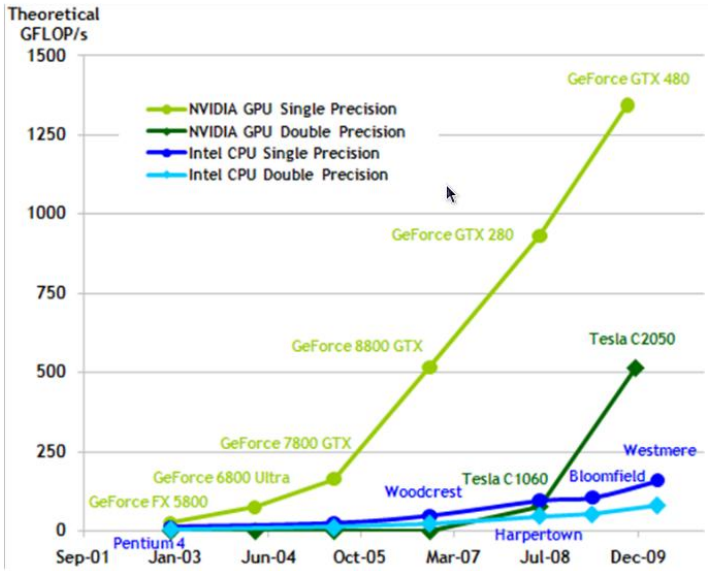
For more information about the sites and systems in the list, click on the links or view the complete list.

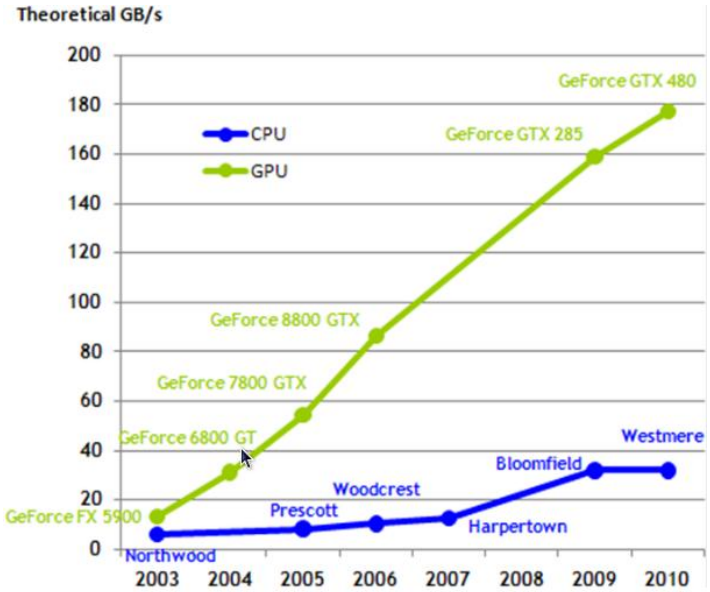
RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
6	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	7,788.9	2,325

Hybrid CPU + GPU

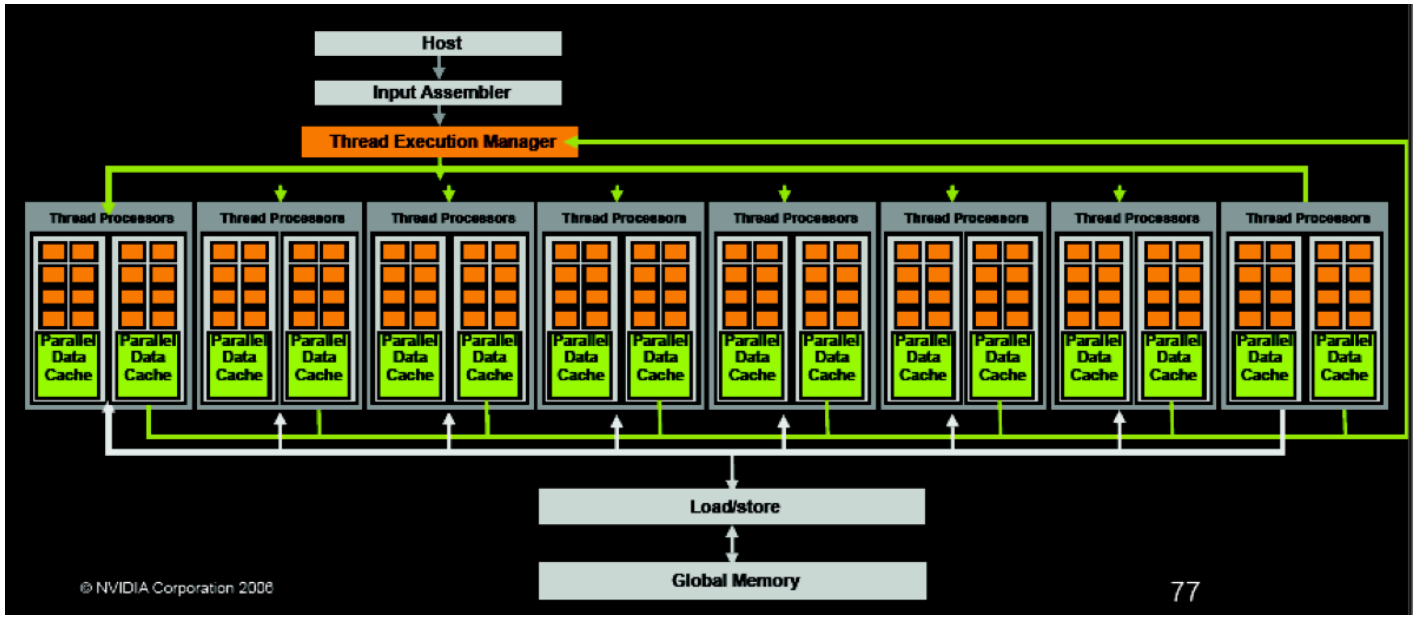
Au LIX, machine bi Tesla K-20m

Calcul sur les cartes graphiques, hautement parallélisées:



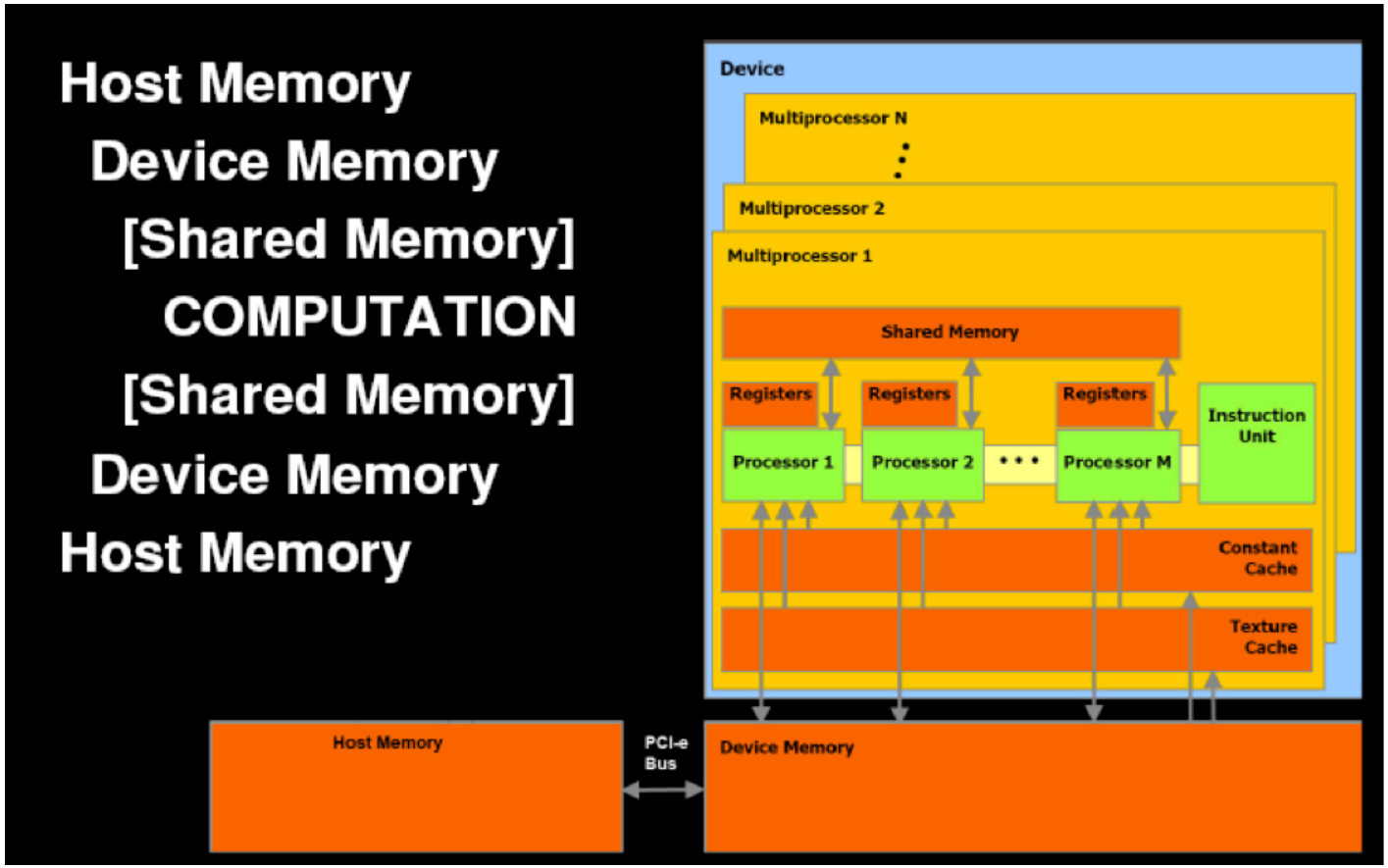


Exemple: architecture G80



(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

Architecture mémoire



Architecture

- Organisés en multiprocesseurs
  - 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)
- TDs en salles Info (salle Info 32)
  - Cartes Quadro K2000 (salles 31, 32, 33, 34): 384 coeurs (2 multiproc de 192 coeurs) à 954MHz (2Go RAM, environ 59Go/s transfert CPU↔GPU) 733 GFlops/s max, cuda capability 3.0

  - GeForce GT 430 (salles 30, 35, 36): 96 (2 multiproc de 48 thread processors) coeurs à 1.40 GHz (1Go RAM, environ 5Go/s transfert CPU↔GPU) 268 GFlops/s max, cuda

Accès possible à une machine du LIX, architecture TESLA K20-m

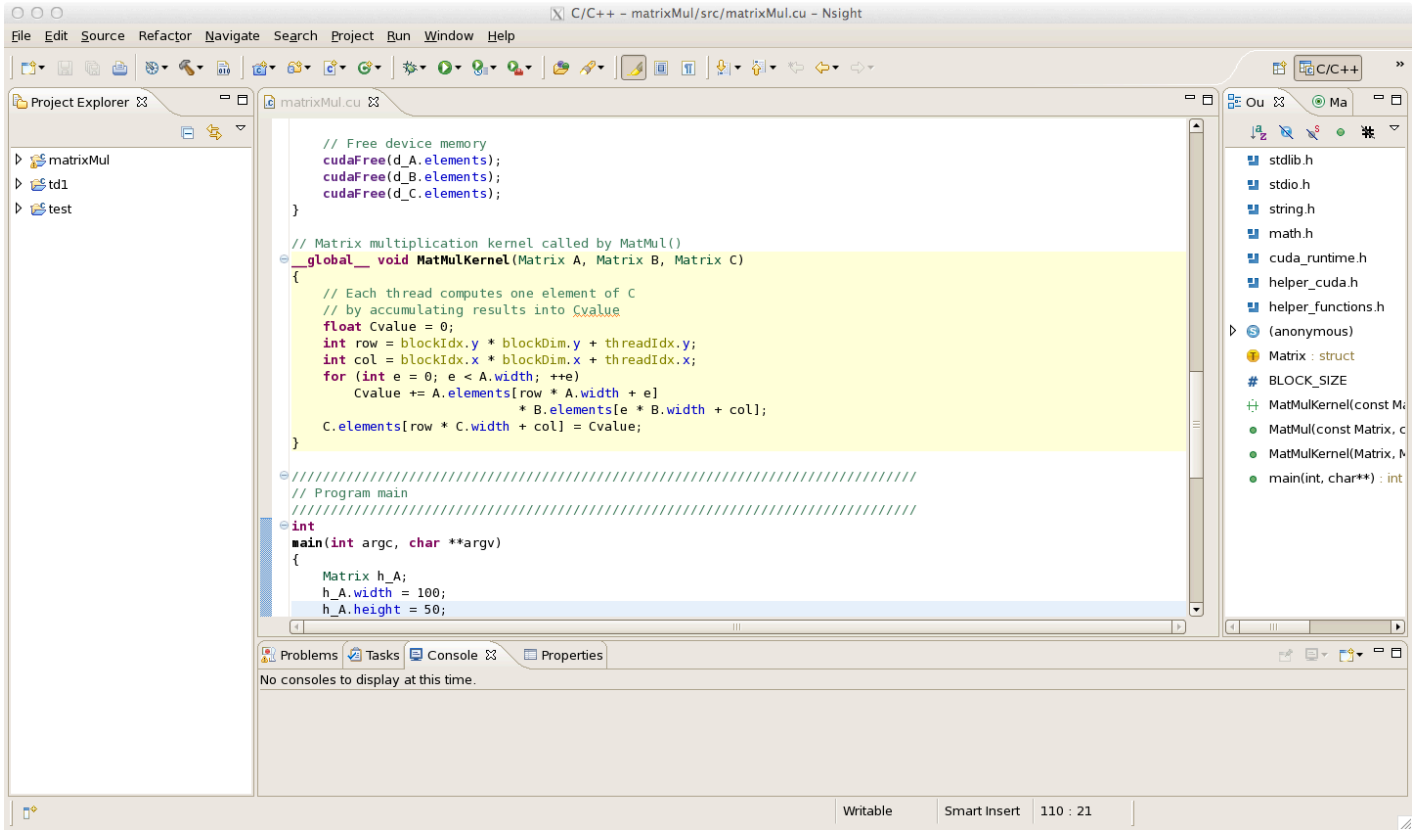
TECHNICAL SPECIFICATIONS	TESLA K10[a]	TESLA K20	TESLA K20X
Peak double precision floating point performance (board)	0.19 teraflops	1.17 teraflops	1.31 teraflops
Peak single precision floating point performance (board)	4.58 teraflops	3.52 teraflops	3.95 teraflops
Number of GPUs	2 x GK104s	1 x GK110	
Number of CUDA cores	2 x 1536	2496	2688
Memory size per board (GDDR5)	8 GB	5 GB	6 GB
Memory bandwidth for board (ECC off)[b]	320 GBytes/sec	208 GBytes/sec	250 GBytes/sec
GPU computing applications	Seismic, image, signal processing, video analytics	CFD, CAE, financial computing, computational chemistry and physics, data analytics, satellite imaging, weather modeling	
Architecture features	SMX	SMX, Dynamic Parallelism, Hyper-Q	
System	Servers only	Servers and Workstations	Servers only

TDs CUDA: portables perso

Si vous avez une machine avec Cartes NVIDIA, par exemple GeForce, vous pouvez télécharger CUDA:

- `https://developer.nvidia.com/category/zone/cuda-zone` sous Windows, Vista, Linux, MacOs etc.
- installer le driver CUDA, le compilateur `nvcc` et la `sdk` (avec de multiples exemples) et nsight
- Attention à la "CUDA capability" et aux calculs simple/double précision!
- Si carte non compatible CUDA, possibilité d'utiliser OpenCL

Nsight



Environnement intégré sous Eclipse de développement, compilation et debug (mais avec 2 cartes graphiques...) en C/C++ - alternative, jcuda: `http://www.jcuda.org/`

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)

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,

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();
   }
}
```

Exécution

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

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

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 dans les versions récentes de Java),
- méthode `sleep(long)` (suspension du thread pendant un certain nombre de nanosecondes)

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

Exemple

```
class Compte implements Runnable {
   int valeur;

   Compte(int val) {
      valeur = val;
   }

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

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

Exemple

```
class Compte3 extends Thread {
   int valeur;

   Compte3(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 Compte3(1).start();
      new Compte3(2000).start();
      printThreads(); } }
```

Exécution

```
% java Compte3
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
%
```

Etats d'un thread

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

- NEW: l'état initial, entre sa création et `start()`.
- RUNNABLE: l'état prêt, après `start`.
- BLOCKED: lorsque l'ordonnanceur place le thread en pause pour en exécuter un autre
- TIMED_WAITING: lorsque le thread est en pause (par la méthode sleep(), par exemple).
- WAITING: lorsque le thread est en attente
- TERMINATED: quand `run()` est terminé. On ne peut plus le relancer

Détermination de l'état par la méthode `getState()`

Exemple

```
class Compte4 extends Thread {
   int valeur;

   Compte4(int val) {
      valeur = val;
   }

   public void run() {
      try {
         for (int i=0;i<=5;i++) {
            System.out.print(valeur + " ");
            sleep(100);
         }
      } catch (InterruptedException e) {
           return; } }

    public static void main(String [] args) {
      Compte4 C1 = new Compte4(1);
      System.out.println("C1 is "+C1.getState()+" ");
      C1.start ();
      Compte4 C2 = new Compte4(2000);
      C2.start ();
      try {
for (;;) {
  System.out.print("C1 is "+C1.getState()+" ");
  System.out.print("C2 is "+C2.getState()+" ");
  sleep (100);
}
      } catch (InterruptedException e) {
return ; }
} }
```

Exécution

```
% java Compte4
C1 is NEW
1 C1 is RUNNABLE C2 is RUNNABLE 2000 C1 is RUNNABLE C2 is RUNNABLE 1 2000
C1 is RUNNABLE 2000 1 C2 is BLOCKED 1 2000 C1 is RUNNABLE C2 is TIMED_WAITING
2000 1 C1 is RUNNABLE C2 is TIMED_WAITING C1 is RUNNABLE 1 2000 C2 is BLOCKED
C1 is RUNNABLE C2 is TERMINATED C1 is TERMINATED ^C
%
```

Etats d'un thread

- 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`.
- `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 et Ordonnancement

- Priorité = nombre entier, qui plus il est grand, plus le processus est prioritaire.
- La priorité peut être maximale: `Thread.MAX_PRIORITY`, normale (par défaut): `Thread.NORM_PRIORITY` (au minimum elle vaut `Thread.MIN_PRIORITY`).
- `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

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.

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'ordonnancer 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 $\longrightarrow$ assure l'*équité*; aucune *famine* (plus tard...)
  - Ordonnancement 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()`)
- Ordonnancé sur plusieurs coeurs/processeurs

La prochaine fois

- TD 1 (Salle Info 32) a 10h15
- Cours 2 en Amphi Lagarrigue lundi 15/12 08h30
  - Modèles et algorithmique PRAM
  - Introduction à la programmation CUDA