Programmation Parallèle

Fathi El Yafi

Directeur de Projets de Développement

Simulation Numérique

1ère Partie: Généralités

Nous Allons Faire:

- Introduction
- Architecture des Calculateurs Parallèles
- Modèles de Programmation Parallèle

Introduction:
Qu'est-ce que le calcul parallèle?

Le calcul parallèle

L'ensemble des techniques **logicielles** et **matérielles** permettant l'exécution simultanée de séquences d'instructions **indépendantes** sur des processus et/ou cœurs différents

- Techniques matérielles: les différentes architectures de calculateur parallèle
- Techniques logicielles: les différents modèles de programmation parallèle

Introduction:
Qu'est-ce que le calcul parallèle?

Faire coopérer plusieurs processeurs pour réaliser un calcul ou une tache.

Avantages:

- Rapidité:

  Pour N processeurs, temps de calcul divisé par N, en théorie…

- Taille mémoire:

  Pour N processeurs, on dispose de N fois plus de mémoire, en général

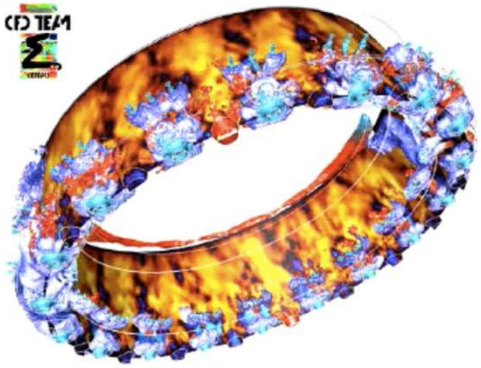
Difficultés:

- Il faut gérer le partage des tâches
- Il faut gérer l'échange d'information. tâche non-indépendantes

Introduction: Objectifs du calcul parallèle

- Exécution plus rapide d'un programme en distribuant le travail
- Exécution de problèmes plus gros en utilisant plus de ressources matérielles, notamment la mémoire

Example:



Allumage d'une chambre de combustion d'hélicoptère réalisé avec le code AVBP

Temps de calcul sur 112 processeurs Intel Xeon hexa-cœurs cadencé à 2,67 Ghz :

78 heures

Temps de calcul sur 1 processeur de même type :

près de 1 an

Architectures des Calculateurs Parallèles

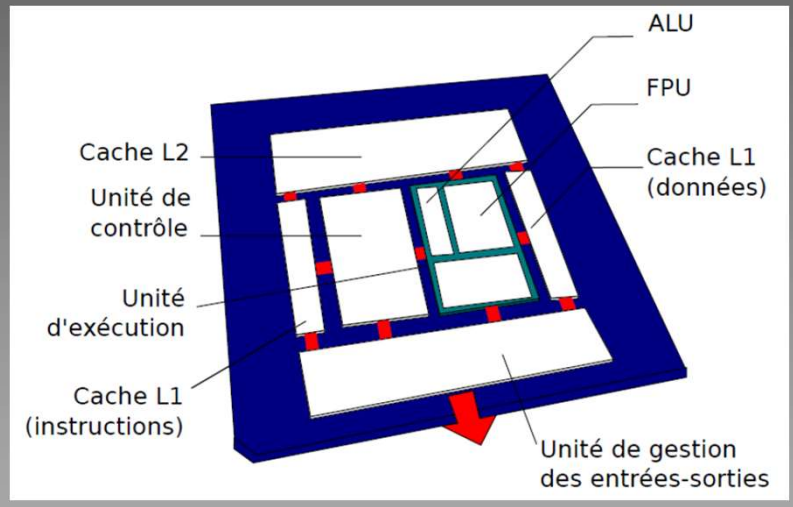
Les composants d'un supercalculateur

- Processeur: fournit la puissance de calcul
- Nœud: contenant plusieurs processeurs partageant une même mémoire
- Mémoire: dont l'accès depuis le processeur n'est pas uniforme
- Réseau:
  - Relient les nœuds entre eux
  - De plusieurs types (calcul, administration, E/S)
- Accélérateur:
  - Processeur graphique (GPU) utilisé pour le calcul
  - Processeur MIC (Many Integrated Core) d'Intel (plusieurs cœurs sur la même puce).
  - Stockage

Architectures des Calculateurs Parallèles

Les composants d'un processeur (CPU, Central Processing Unit)

- 1 unité de gestion des entrée-sorties.
- 1 unité de commande (Control Unit) :
  lit les instructions arrivant, les décode
  puis les envoie à l'unité d'exécution.
- 1 unité d'exécution
  - 1 ou plusieurs ALU (Arithmetical and Logical Unit) :
    en charge notamment des fonctions basiques
    de calcul arithmétique.
  - 1 ou plusieurs FPU (Floating Point Unit) :
    en charge des calculs sur les nombres flottants.
- Registres et caches : mémoires de petites tailles.



Architectures des Calculateurs Parallèles

Principales caractéristiques d'un processeur (1/2)

- Le jeu d'instructions (ISA, Instruction Set Architecture) constitue l'ensemble des opérations élémentaires qu'un processeur peut accomplir.
  - CISC (Complex Instruction Set Computer):
    Choix d'instructions aussi proches que possible d'un langage de haut niveau.
  - RISC (Reduced Instruction Set Computer):
    Choix d'instructions plus simples et d'une structure permettant une exécution très rapide.
- La finesse de gravure des transistors dont dépend la fréquence du processeur et sa consommation (de l'ordre de 18 nm en 2017).

Architectures des Calculateurs Parallèles

Principales caractéristiques d'un processeur (2/2)

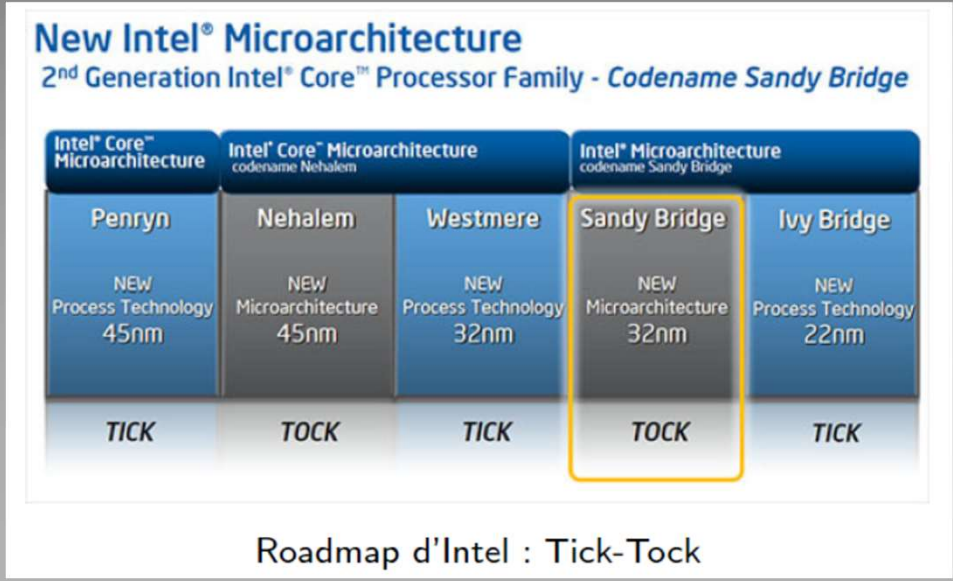
- La fréquence de l'horloge (en MHz ou GHz)
  - Elle détermine la durée d'un cycle (un processeur exécute chaque instruction en un ou plusieurs cycles).
  - Plus elle augmente, plus le processeur exécute d'instructions par seconde.
  - Actuellement, supérieur à 3 GHz (temps de cycle ≈ 0.33 ns).
  - Croissance de plus en plus limitée.
  - Principales limites à sa croissance : la consommation électrique et la dissipation thermique du processeur.

Architectures des Calculateurs Parallèles

Augmenter les performances des processeurs

- Augmenter la fréquence de l'horloge
  mais limites techniques et solution coûteuse.

- Introduire de nouvelles technologies afin d'optimiser l'architecture générale du processeur.



Roadmap d'Intel : Tick-Tock

Architectures des Calculateurs Parallèles

Parallélisme au niveau de l'architecture

Ce parallélisme peut être qualifié d'implicite:
il est transparent pour l'utilisateur.

- Parallélisme d'instruction (ILP: Instruction Level Parallelism).
  - Pipeline et architectures superscalaire.
- Parallélisme de données (DLP: Data Level Parallelism)
  - SIMD (Single Instruction Multiple Data): SSE, AVX…
  - Processeur vectoriel.
- Parallélisme au niveau des threads (TLP: Thread Level Parallelism)
  - SMT (Simultaneous Multithreading).
    Hyperthreading chez Intel.

Architectures des Calculateurs Parallèles

Les différentes étapes d'exécution d'une opération:

Une opération s'exécute en plusieurs étapes indépendantes par des éléments différents du processeur :

- IF : Instruction Fecht
- ID : Instruction Decode/Register Fecht
- EX : Execution/Effective Address
- MA : Memory Access/ Cache Access
- WB : Write-Back

Sur un processeur sans pipeline les instructions sont exécutées les unes après les autres

Architectures des Calculateurs Parallèles

- Parallélisme d'instruction (ILP Pipeline):

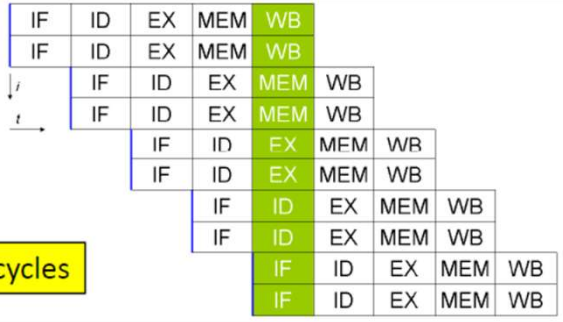
Les différentes étapes d'une opération s'exécutent simultanément sur des données distinctes (parallélisme d'instructions).

5 instructions en parallèle en 9 cycles (25 cycles en séquentiel)
➔ permet de multiplier le débit avec lequel les instructions sont exécutées par le processeur.

- Parallélisme d'instruction (Architecture Superscalaire):

Principe

Duplication de composants (FMA, FPU) et exécution simultanée de plusieurs instructions



10 instructions en 9 cycles

Architectures des Calculateurs Parallèles

Parallélisme de données (DLP: Data Level Parallelism)

- SIMD (Single Instruction Multiple Data)
  Ce mode permet d'appliquer la même instruction simultanément à plusieurs données (stockées dans un registre) pour produire plusieurs résultats.

Exemples de jeux d'instructions SIMD

MMX : permettent d'accélérer certaines opérations répétitives dans certains domaines :traitement de l'image 2D, son et communications.
SSE (SSE2,SSE3, SSE4, SSE5 annoncé par AMD) : par ex instructions pour additionner et multiplier plusieurs valeurs stockées dans un seul registre
3DNow : jeu d'instructions multimédia développé par AMD
AVX (Advanced Vector Extensions) : nouvel ensemble d'instructions sur les proc SandyBridge intel, unité SIMD de largeur 256 ou 512 bit

- Certaines architectures disposent d'instructions vectorielles
  - L'instruction vectorielle est décodée une seule fois, les éléments du vecteur sont ensuite soumis un à un à l'unité de traitement
  - Les pipelines des unités de traitement sont pleinement alimentés
    Ex: NEC SX, Fujitsu VPP, Altivec…

Architectures des Calculateurs Parallèles

Parallélisme au niveau des threads (TLP: Thread Level Parallelism)

L'idée consiste à mixer deux flux d'instructions arrivant au processeur pour optimiser l'utilisation simultanée de toutes les ressources (remplir les cycles perdus – cash miss, load …).

- Le SMT (Simultaneous Multithreading) partage le pipeline du processeur entre plusieurs threads (d'un même programme ou de deux programmes différents). Les registres et les caches sont aussi partagés.
- Hyperthreading chez Intel.

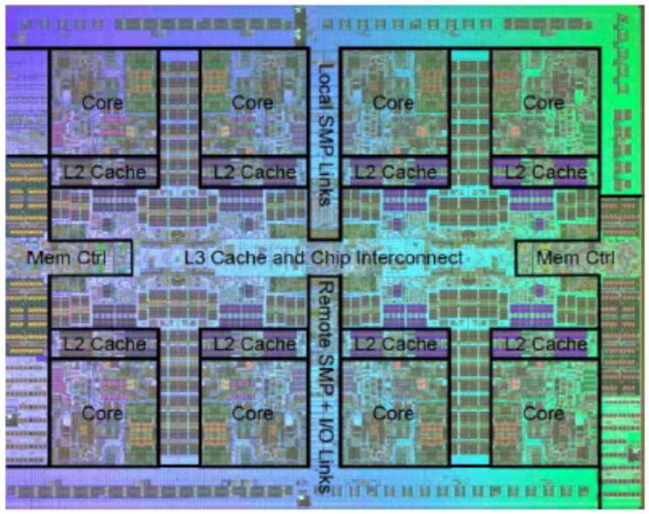
Le multithreading dégrade les performance individuelles des threads mais améliore les performances de l'ensemble.

Architectures des Calculateurs Parallèles

Performance des processeurs: multicœurs

Plusieurs instances de calcul sur un même *socket*(1)

- Connectique inchangée par rapport à un mono-cœur.
- Certains éléments (caches L2/L3…) peuvent être partagés.
- Exemple:
  Intel Westmere jusqu'à 10 cœurs par socket,
  Power 7 jusqu'à 10 cœurs par socket..



IBM Power7 (octo-cœurs)

(1) socket : connecteur utilisé pour interfacer un processeur avec une carte mère (les multiprocesseurs utilisent plusieurs sockets)

Architectures des Calculateurs Parallèles

Intérêt du multicœur

- Diminuer la fréquence des cœurs pour réduire la consommation électrique et la dissipation thermique, mais en mettre plus pour gagner en puissance de calcul.

	mono-cœur	bi-cœurs
Fréquence	F	0.75F
Consommation par processeur	W	0.84 W
Performance par processeur	P	1.5P

Un bicœurs avec une fréquence moindre de 25% par rapport à un monocœur offre 50% de performance en plus et 20% environ de consommation en moins qu'un mono-cœurs.

Architectures des Calculateurs Parallèles

Puissance crête d'une machine

- Puissance crête (Peak performance) $\mathbf{R_{peak}}$ exprimée en Flops (Floating point Operations Per second).

R_{peak} (pour des flottants DP ou SP)

$$R_{peak} = N_{FLOP}/cycle \times Freq \times N_c$$

Freq : Fréquence du processeur.
NFLOP/cycle : Nombre d'opérations flottantes (SP ou DP) effectuées par cycle.
Nc : Nombre de cœurs du processeur ou de la machine.
Exemple : un processeur hexacœurs à 2:66 GHz (Intel Westmere) est capable d'effectuer 4 opérations flottantes par cycle :
Rpeak (DP) = 4 2:66 109 6 = 63:8 Gigaflops.

Architectures des Calculateurs Parallèles

Puissance de calcul d'une machine

Quelques ordres de grandeur:

- PC de bureau: de quelques dizaines à une centaine de Gigaflops (1)
  Exemple pour une machine équipée de 2 processeurs Westmere hexacœurs à 2.66 GHz: $\mathbf{R_{peak}}$ = 127.6 Gigaflops (1)

- Calculateurs de mesocentre: quelques dizaines de Teraflops (1).

- Premiers supercalculateurs du Top 500: quelques Petaflops (1)
  (2018: Exaflops (1))!!

(1) : Giga/Tera/Peta/Exaflops=10^9/10^{12}/10^{15}/10^{18} Flops.

Architectures des Calculateurs Parallèles

Puissance maximale d'une machine

- La puissance crête $\mathbf{R_{peak}}$ est une valeur théorique maximale !

$$\mathbf{R_{max}} < \mathbf{R_{peak}} \text{ pour un benchmark donné}$$

- Dans la pratique, la puissance maximale $\mathbf{R_{max}}$ dépend de nombreux paramètres et en particulier:
  - De la charge de la machine
  - Du système de fichiers pour le IOs importantes
  - Des accès mémoire (code de calcul «*memory-bound*»)

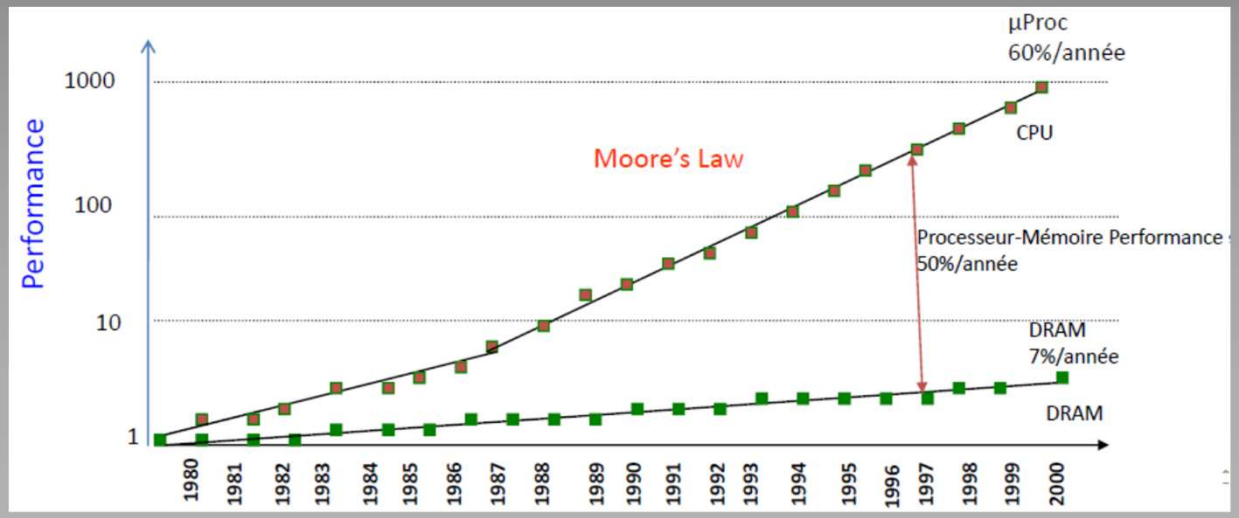
Architectures des Calculateurs Parallèles

Performance: mémoire versus µprocesseur

Cas idéal: Mémoire la plus grande et la plus rapide possible.

La performance des ordinateurs est limitée par la latence et la bande passante de la mémoire:

- Latence = temps pour un seul accès.
  Temps d'accès mémoire >> temps cycle processeur
- Bande passante = nombre d'accès par unité de temps



Architectures des Calculateurs Parallèles

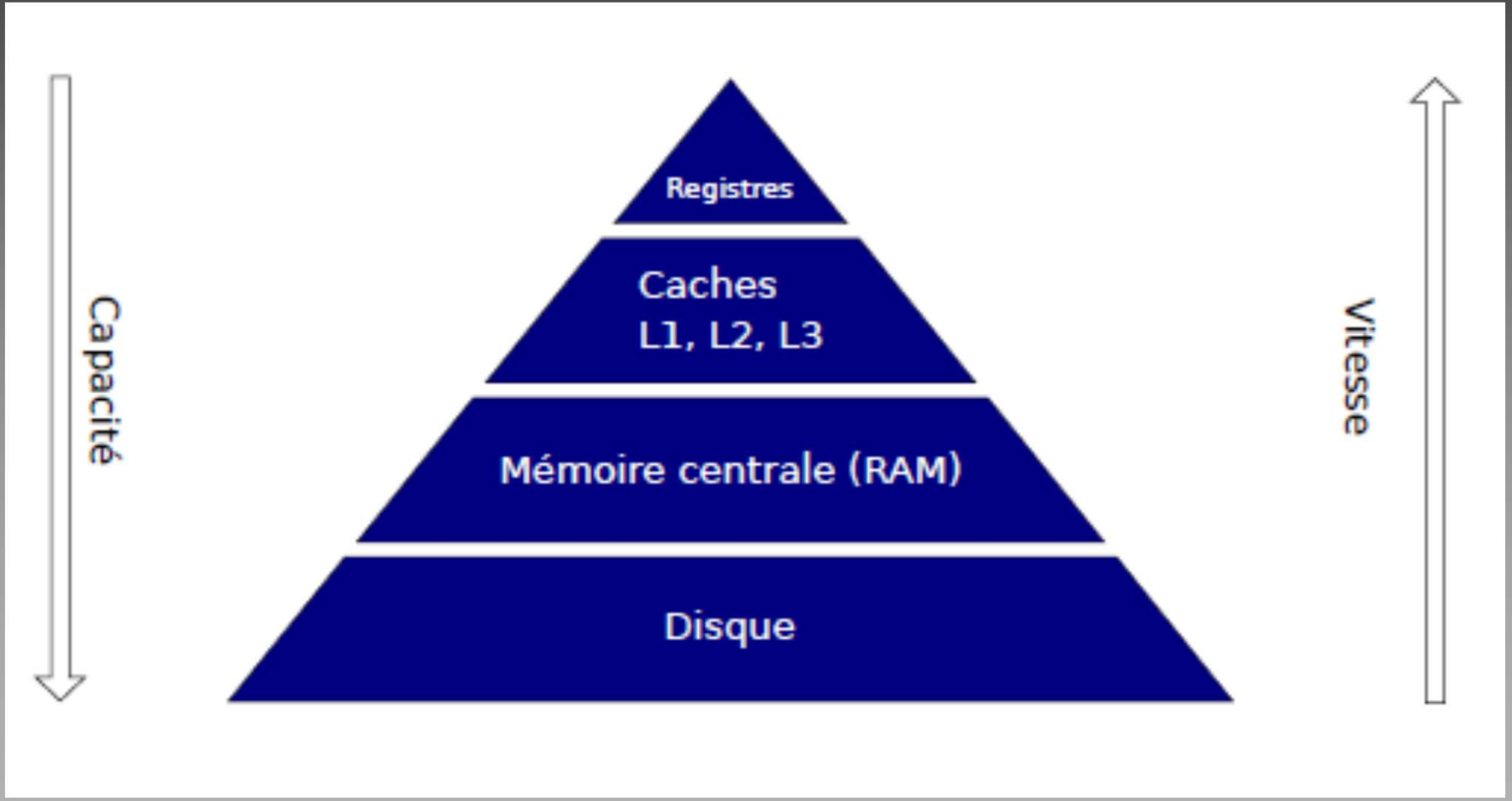
Mémoire

Principales caractéristiques:

- Type (SRAM, SDRAM, DDR-SDRAM…)
- Capacité: mémoire ou disque
- Latence: le temps passé entre une requête et le début de la réponse correspondante, exprimée en seconde (cycle ≈ .033 ns pour un processeur à 3 GHz).
- Type d'accès (séquentiel, direct…)

Architectures des Calculateurs Parallèles

Hiérarchie Mémoire(1/3)



Architectures des Calculateurs Parallèles

Hiérarchie Mémoire(2/3)

- Registres: mémoires intégrées au processeur.
  Accès rapide: en général, 1 cycle.
- Caches: mémoires intermédiaires (SDRAM) entre le processeur et la mémoire centrale.
  Les caches sont hiérarchisées (L1, L2 et L3).
  Ils peuvent être privés ou partagés entre les cœurs.
  L1: accédé en quelques cycles, L2: latence 2 à 10 fois supérieure.

Exemple, processeur Nehalem : 64 *KiB* de cache L1 par cœurs, 256 *KiB* de cache L2 par cœurs et 8 *MiB* de cache L3 partagé par tous les cœurs.

Architectures des Calculateurs Parallèles

Hiérarchie Mémoire (3/3)

- Mémoire centrale (RAM).
  De la dizaine à quelques centaines de GB.
  Latence de quelques centaines de cycles.
- Disques.
  Généralement plusieurs TB.
  Latence de plusieurs millions de cycles.

Architectures des Calculateurs Parallèles

Caches (1/2)

- Le cache est divisé en lignes (ou blocs) de mots.
- Les lignes présentes dans le cache ne sont que des copies temporaires des lignes de la mémoire centrale.
- Le transfert des données de la mémoire centrale vers le cache se fait par lignes.

Architectures des Calculateurs Parallèles

Caches (2/2)

- Lorsque le processeur doit accéder à une donnée:
  - Soit la donnée se trouve dans le cache («cache hit»): le transfert de la donnée vers les registres se réalise immédiatement.
  - Soit la donnée ne se trouve pas dans le cache («cache miss»): une nouvelle ligne est chargée dans le cache depuis la mémoire centrale.
    Pour ce faire, il faut libérer de la place dans le cache et donc renvoyer en mémoire centrale une des lignes, de préférence celle dont la durée d'inactivité est la plus longue.
    Le processeur est en attente pendant toute la durée de chargement de la ligne.

Architectures des Calculateurs Parallèles

Localité

Améliorer le taux de succès du cache (éviter les «cache miss») en utilisant les principes de localité temporelle et spatiale.

- Localité temporelle:
  Lorsqu'un programme accède à une donnée, il est probable qu'il y accédera à nouveau dans un futur proche.
- Localité spatiale:
  Lorsqu'un programme accède à une donnée, il est probable qu'il y accédera ensuite aux données voisines.
- Ces principes sont utilisés par:
  - Les constructeurs via l'architecture du processeur
  - Les programmeurs

Architectures des Calculateurs Parallèles

Localité Spatiale

Exemple: accès aux éléments d'une matrice.

Fortran : accès non optimal

```
do i = 1, n
  do j = 1, n
    y(i) = a(i,j) * x(j)
  end do
end do
```

C : accès non optimal

```
for (j=0; j<n; ++j)
{
  for (i=0; i<n; ++i)
  {
     y[i] += a[i][j] * x[j];
  }
}
```

Fortran : accès optimal

```
do j = 1, n
  do i = 1, n
    y(i) = a(i,j) * x(j)
  end do
end do
```

C : accès optimal

```
for (i=0; i<n; ++i)
{
  for (j=0; j<n; ++j)
  {
     y[i] += a[i][j] * x[j];
  }
}
```

Temps sans optimisation du compilateur

- Implémentation non optimale en C: 53.26 s
- Implémentation optimale en C: 7.66 s

Architectures des Calculateurs Parallèles

Localité Temporelle

```
DO I = 1,10 0000
    S1 = A(I)
    S2 = A(I+K) # S2 sera réutilisé à l'itération I+K
END DO
```

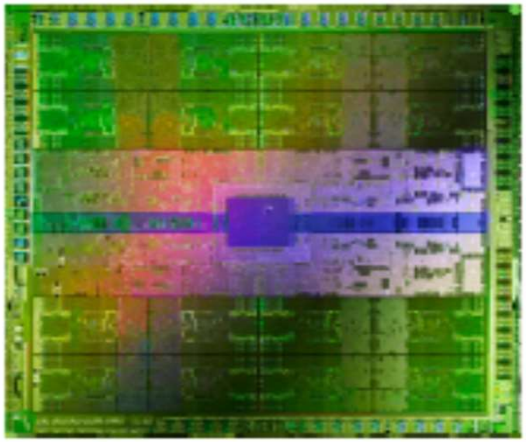
- L'idée de base est de conserver A(I+K) (lecture de la référence à l'itération I, déclaration S2) en mémoire rapide jusqu'à sa prochaine utilisation à l'itération I+K, déclaration S1.
- Cependant dans le cas générale, on aura certainement besoin de stocker d'autres données.

L'évaluation précise de la proximité temporelle est très complexe

Architectures des Calculateurs Parallèles

Accélérateurs GPU

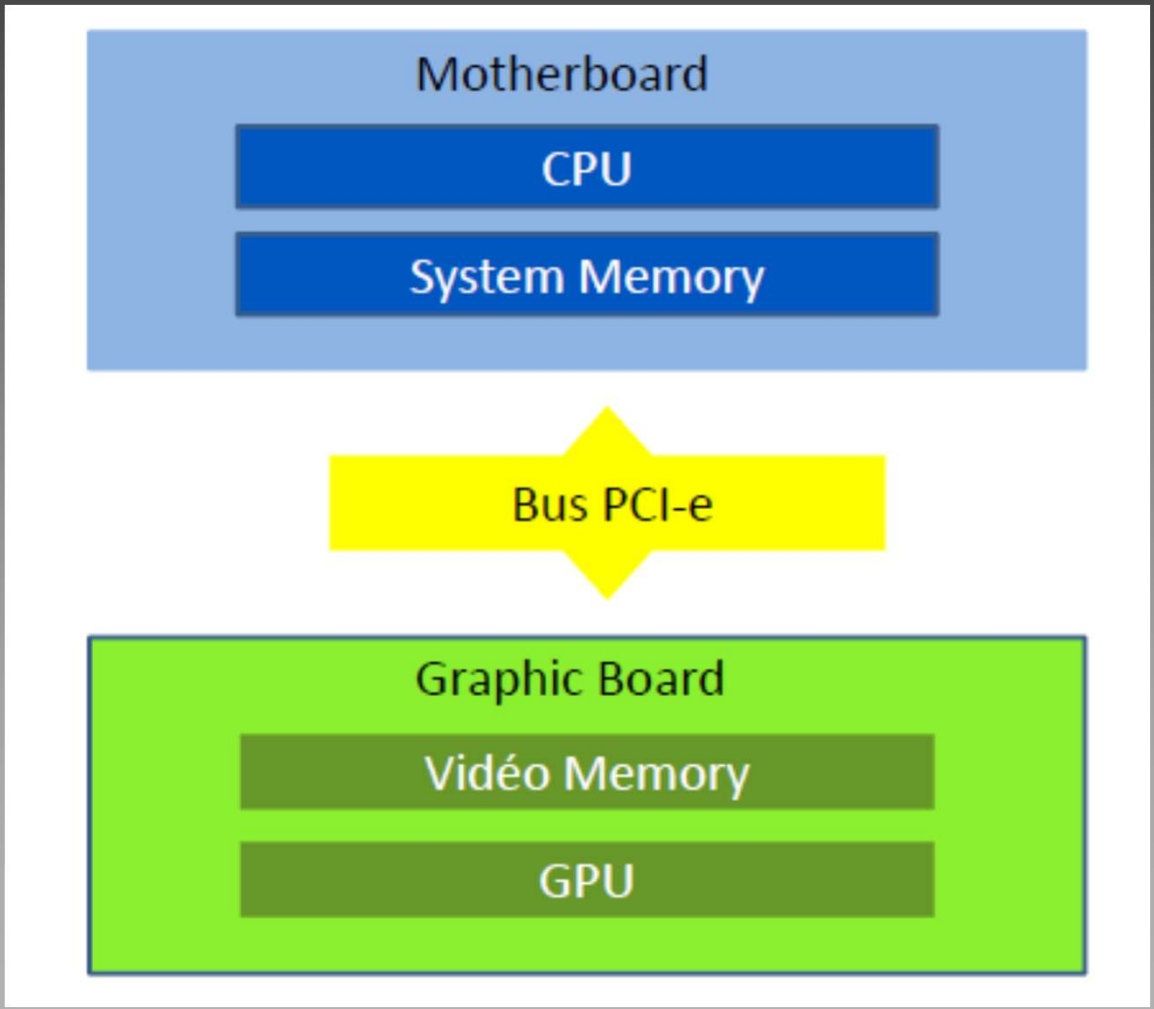
- GPU: *Graphic Processing Unit*:
  processeur **massivement parallèle, disposant de sa propre mémoire** assurant les fonctions de calcul de l'affichage.



Puce NVidia Fermi et carte Tesla

Architectures des Calculateurs Parallèles

Architecture de PC classique



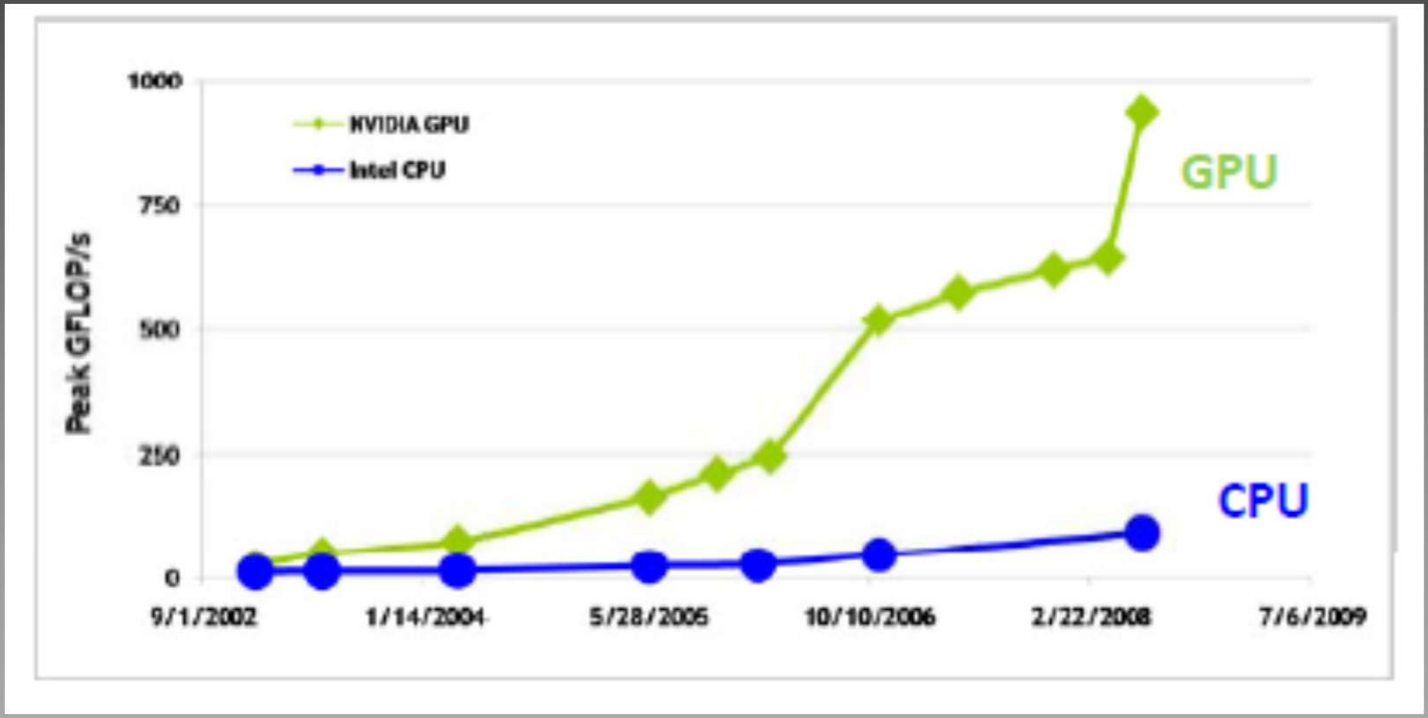
Architectures des Calculateurs Parallèles

CPU vs GPU

- CPU: optimisé pour exécuter rapidement une série de tâches de tout type.
- GPU: optimisé pour exécuter des opérations simples sur de très gros volumes de données.
  => Beaucoup plus de transistors dédiés au calcul sur les GPUs et moins de contrôle

Architectures des Calculateurs Parallèles

Evolution des performances CPU vs GPU

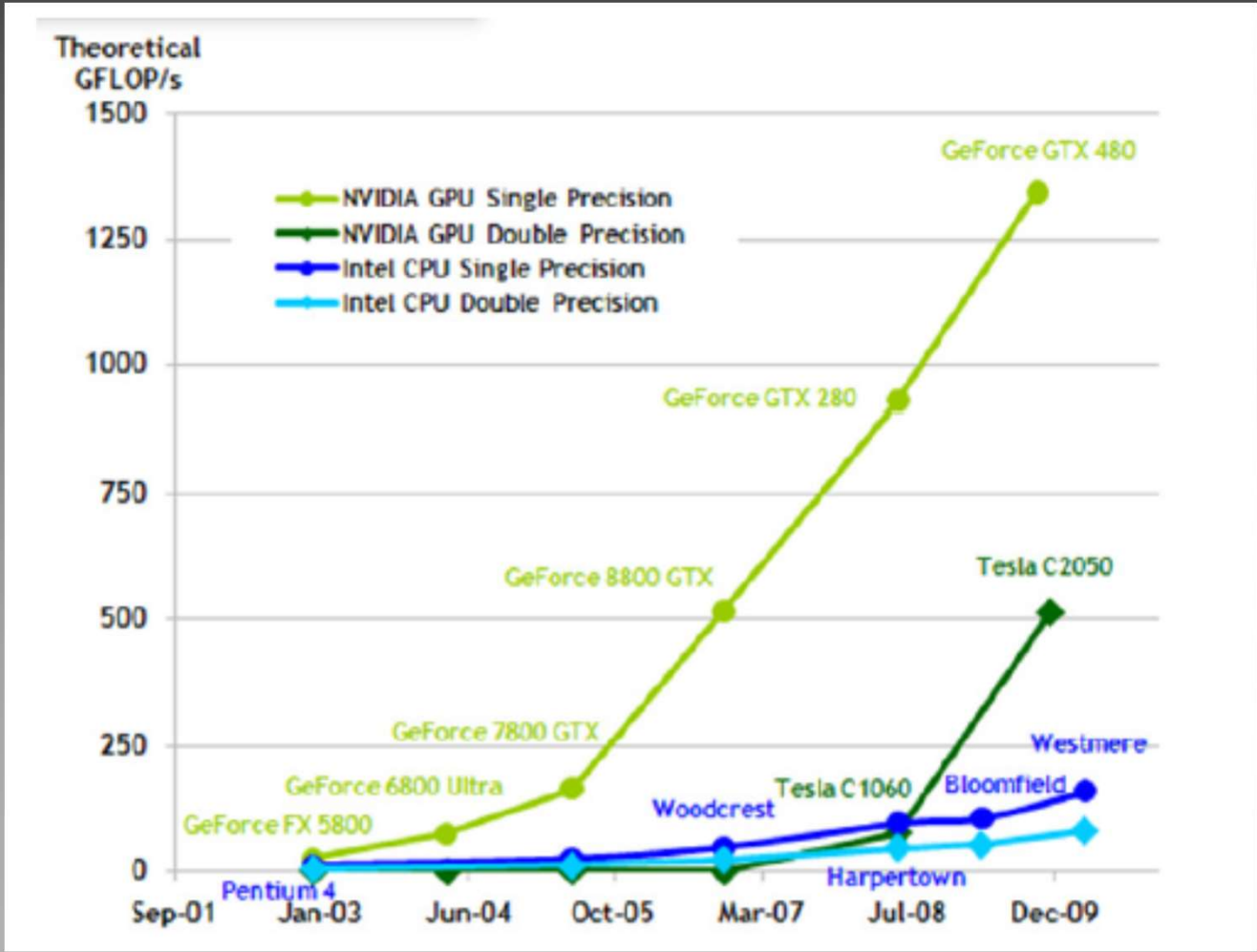


Vitesse des processeurs classique * 2 tous les 16 mois

Vitesse des processeurs graphiques * 2 tous les 8 mois

Architectures des Calculateurs Parallèles

Pourquoi utiliser des GPUs pour le HPC?



Architectures des Calculateurs Parallèles

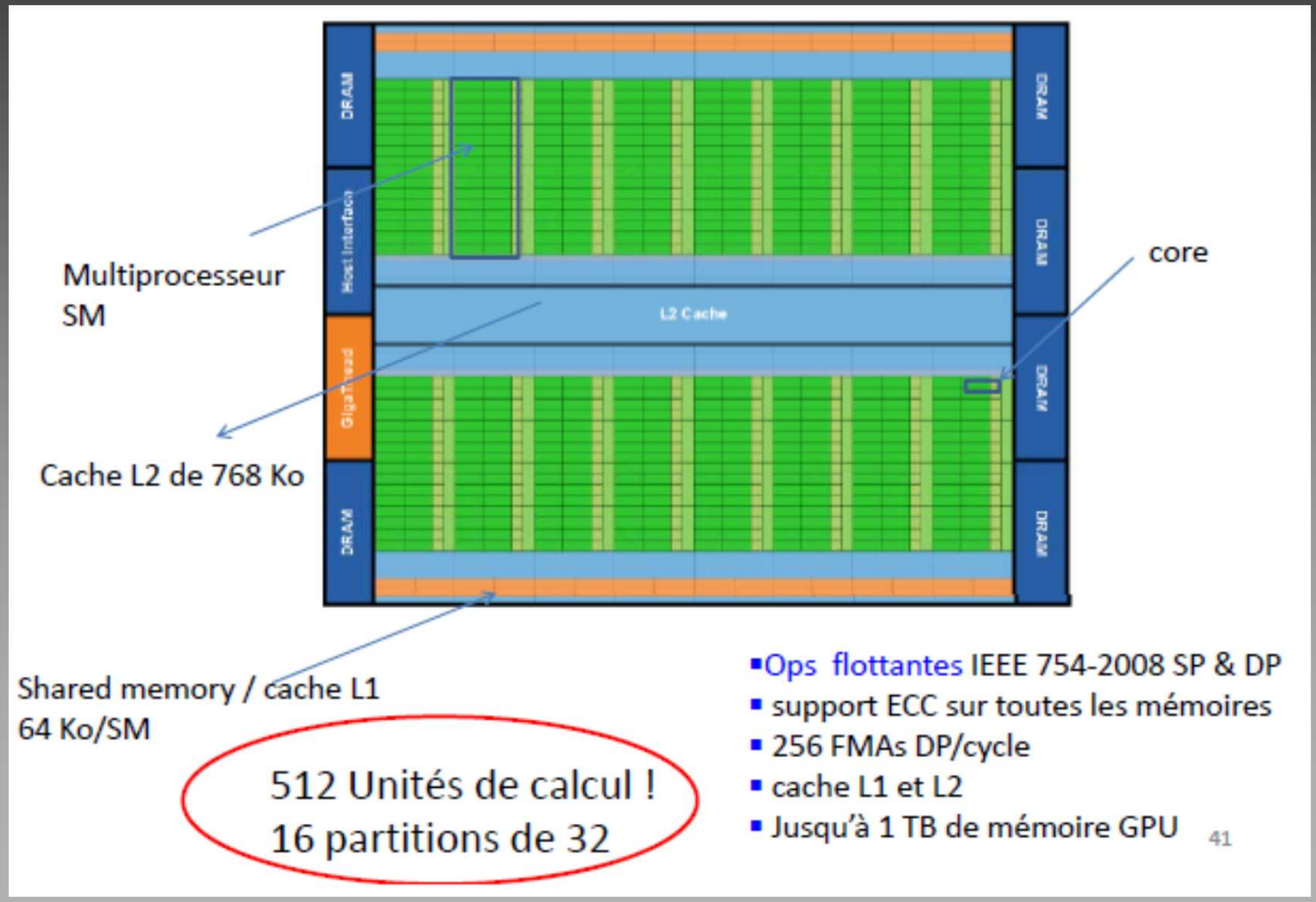
Pourquoi utiliser des GPUs pour le HPC?

A performance théorique égale, les GPUs sont, par rapport aux CPUs:

- Des solutions plus denses (9 x moins de place)
- Moins consommateurs d'électricité (7 x moins)
- Moins chers (6 x moins)

Architectures des Calculateurs Parallèles

Architecture du processeur Fermi de NVIDIA



Architectures des Calculateurs Parallèles

Types des architectures

Les différentes architectures se distinguent selon l'organisation de la mémoire:

- Architectures à mémoire partagée
- Architectures à mémoire distribuée
- Architectures mixtes
- Architectures hybrides (GPU)

Architectures des Calculateurs Parallèles

Architectures à mémoire partagée

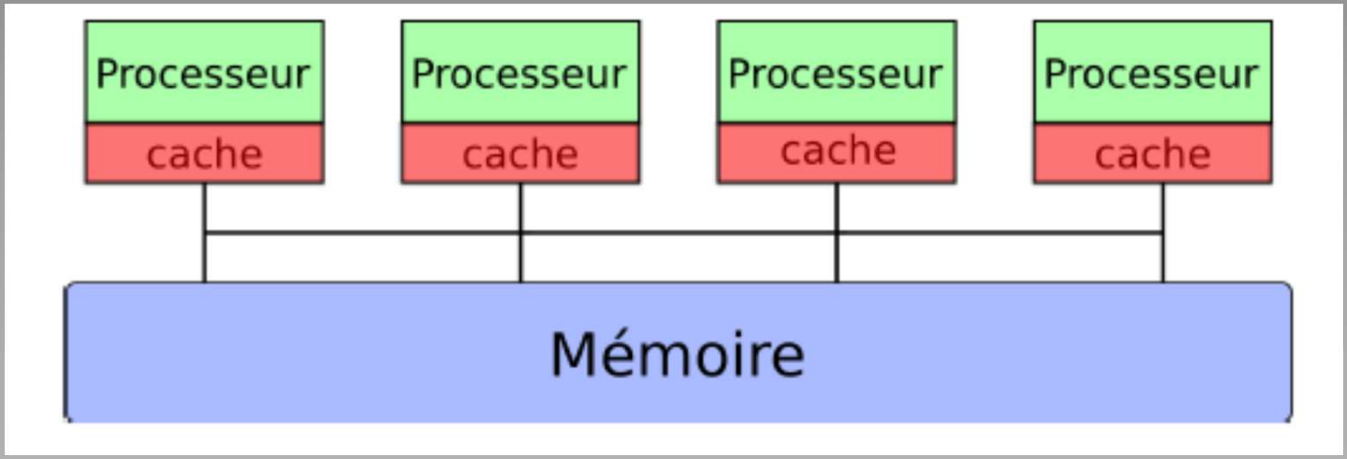
- Un espace mémoire global visible par tous les processeurs.
- Les processeurs ont leur propre mémoire cache dans laquelle est copiée une partie de la mémoire globale.
- On distingue deux classes d'architecture à mémoire partagée:
  - UMA: *Uniform Memory Access*
  - NUMA: *Non-Uniform Memory Access*

Architectures des Calculateurs Parallèles

Architectures à mémoire partagée: UMA

Architecture UMA (*Uniform Memory Access*)

- Une seule mémoire centralisée.
- Le temps d'accès à un emplacement quelconque en mémoire est le même pour tous les processeurs.

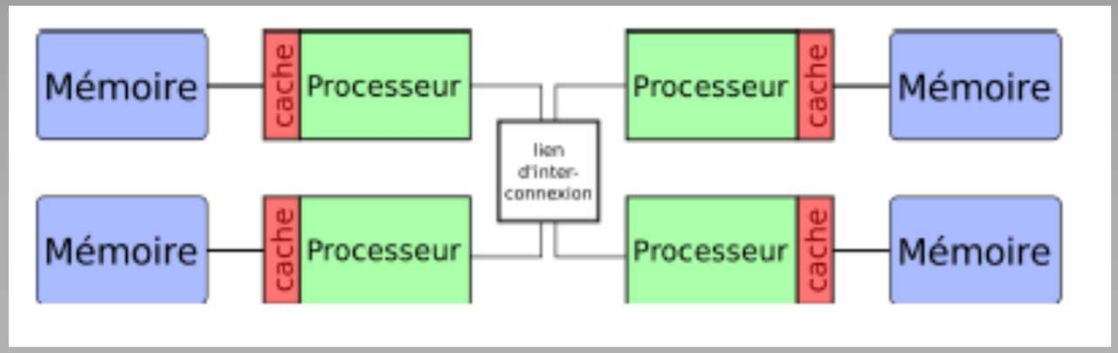


Architectures des Calculateurs Parallèles

Architectures à mémoire partagée: NUMA

Architecture NUMA (Non-*Uniform Memory Access*)

- Une mémoire centrale par processeur (ou par bloc UMA de processeurs).
- Les processeurs sont capables d'accéder à la totalité de la mémoire du système mais les temps d'accès sont non uniformes.
- Le temps d'accès via le lien d'interconnexion est plus lent que le temps d'accès direct vers la mémoire centrale.



Architectures des Calculateurs Parallèles

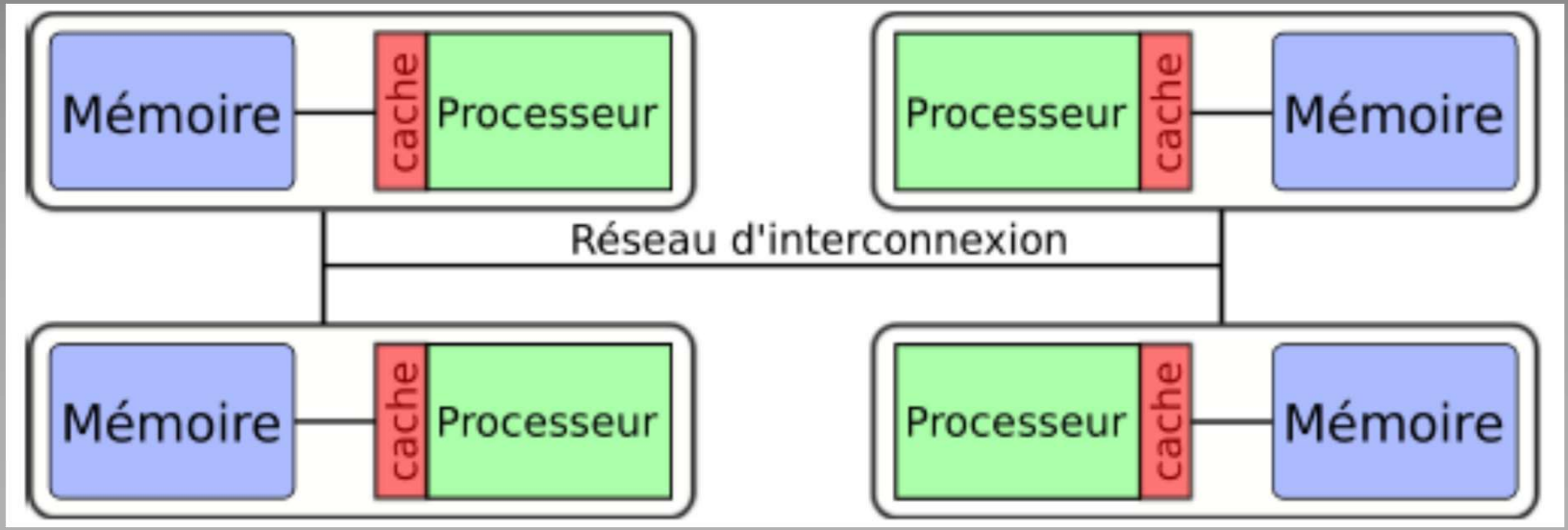
Architectures à mémoire partagée: remarques

- La gestion de la cohérence de cache est le plus souvent transparente pour l'utilisateur (CC-UMA, CC-NUMA).
- L'espace d'adresse global, visible de tous les processeurs, facilite la programmation parallèle.
- L'échange des données entre les tâches attribuées aux processeurs est transparent et rapide.
- Le nombre des processeurs par machine est limité.
- Les performances décroissent vite avec le nombre de processeurs et/ou cœurs utilisés par une application à cause du trafic sur le chemin d'accès des données en mémoire.
- Machine couteuse lorsque le nombre de processeurs devient important.

Architectures des Calculateurs Parallèles

Architectures à mémoire distribuée

- Un espace mémoire est associé à chaque processeur.
- Les processeurs sont connectés entre eux à travers un réseau.
- L'accès à la mémoire du processeur voisin se fait explicitement par échange de messages à travers le réseau d'interconnexion entre les processeurs.



Architectures des Calculateurs Parallèles

Architectures à mémoire distribuée: Réseau d'interconnexion

Le réseau d'interconnexion est important sur ces architectures car il détermine la vitesse d'accès aux données d'un processeur voisin. Les caractéristiques du réseau sont:

- Sa latence: temps pour initier une communication.
- Sa bande passante: vitesse de transfert des données à travers le réseau.
- Sa topologie: architecture physique du réseau.

Topologie en arbre

Topologie en hypercube

Architectures des Calculateurs Parallèles

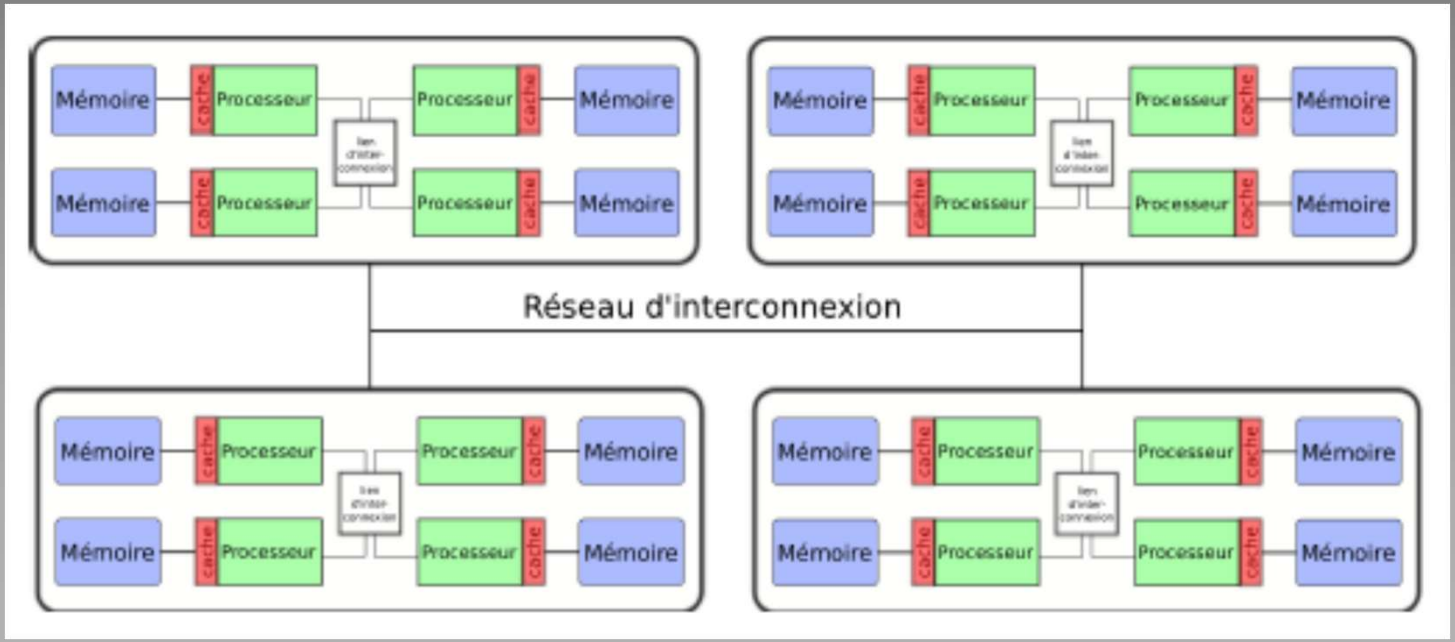
Architectures à mémoire distribuée: remarques

- Accès rapide à la mémoire locale.
- L'architecture permet facilement d'obtenir des machines avec un grand nombre de processeurs et ce, pour un coût réduit par rapport à l'architecture à mémoire partagée.
- L'échange des données entre processeurs doit être géré par le programmeur. Un effort de développement est nécessaire pour utiliser ce type d'architecture.
- Les performances sont dépendantes de la qualité du réseau d'interconnexion (Infiniband, Gigabit, Myrinet…).

Architectures des Calculateurs Parallèles

Architectures mixte

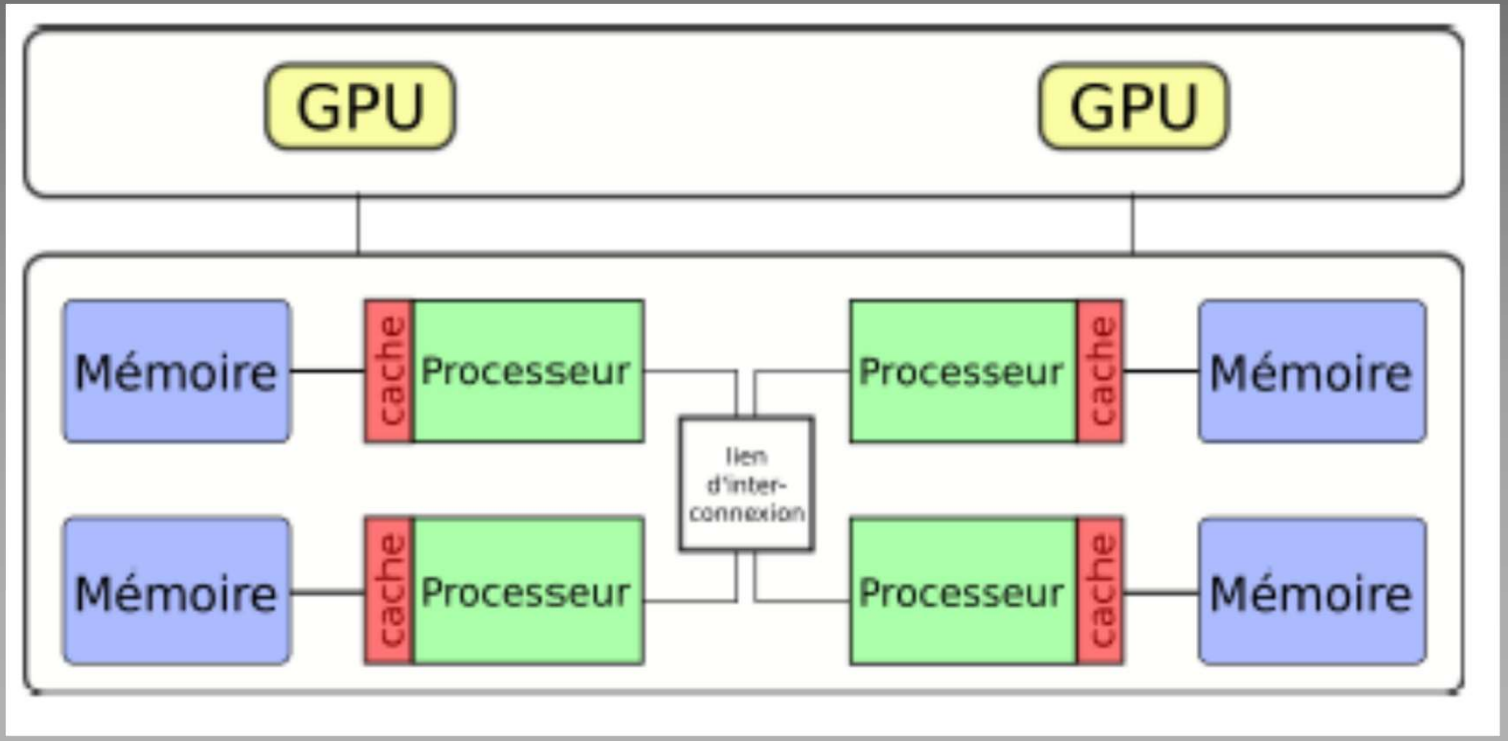
- Aujourd'hui, la plupart des calculateurs combinent les architectures à mémoire partagée et à mémoire distribuée.
- Ces machines sont alors constituées de machines à mémoire partagée (nœud de calcul) reliées entre elles par un réseau d'interconnexion.



Architectures des Calculateurs Parallèles

Architectures hybride

- Avec l'essor des accélérateurs graphiques, de plus en plus de calculateur possèdent des nœuds de calcul équipés de GPU.



Modèles de Programmation Parallèle

A chaque architecture de machine parallèle correspond un modèle de programmation:

- Modèle de programmation à mémoire partagée.
- Modèle de programmation à mémoire distribuée.
- Modèle de programmation mixte: utilisation combinée des deux modèle précédents
- Modèle de programmation hybride: utilisation des GPUs

Modèles de Programmation Parallèle

Programmation à mémoire partagée (1/2): *multithreading*

Pour ce type de modèle, on utilise le plus souvent le multithreading:

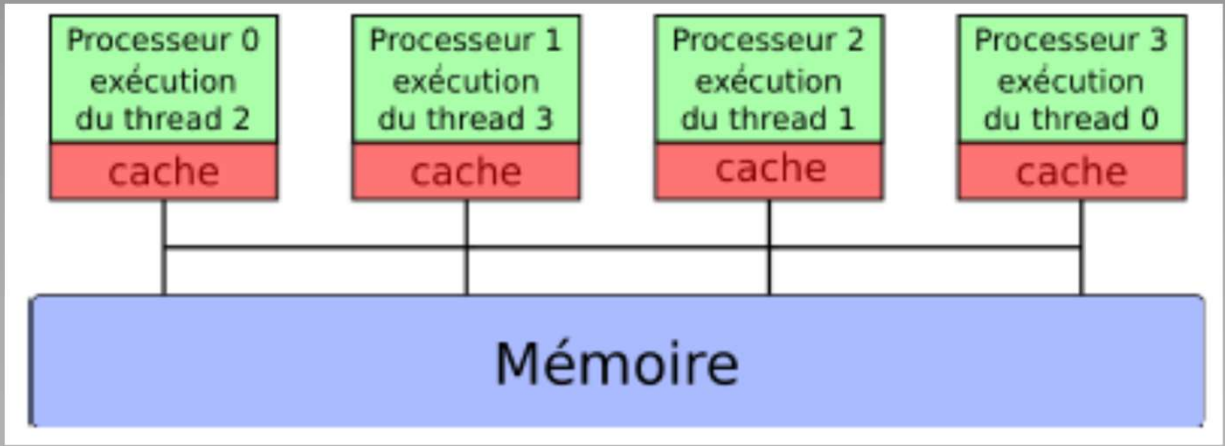
- Un programme multithreading s'exécute dans un processus unique.
- Ce processus active plusieurs processus légers (appelés également *threads*) capables de s'exécuter de manière concurrente.
- L'exécution de ces processus légers se réalise dans l'espace mémoire du processus d'origine.
- Chaque *thread* possède un espace mémoire local invisible des autres *threads*.

Modèles de Programmation Parallèle

Programmation à mémoire partagée (2/2): *multithreading*

- C'est l'exécution concurrente des *threads* sur plusieurs processus ou cœurs qui permet l'exécution parallèle du programme.
- Le système d'exploitation distribue les *threads* sur les différents processeurs ou cœurs de la machine à mémoire partagée.

Exemple: distribution de 4 threads sur une machine à mémoire partagée de 4 processeurs de type UMA.



Modèles de Programmation Parallèle

Programmation à mémoire partagée: *OpenMP*

OpenMP (*Open Multi-Processing*) est une **interface de programmation** (API) pour générer un **programme multithreads**.

- L'interface de programmation fournit:
  - Des directives de compilation.
  - Des sous-programmes.
  - Des variables d'environnement.
- Cette API est:
  - Disponible pour le Fortran, le C et le C++.
  - Supportée par de nombreux systèmes et compilateurs (gnu, intel…).

Modèles de Programmation Parallèle

Programmation à mémoire partagée: *MPI*

MPI (*Message Passing Interface*) est une **interface de programmation** (API) pour générer un **programme par échange de messages**.

- L'interface de programmation fournit des fonctions pour gérer:
  - Un environnement d'exécution.
  - Les communications point à point.
  - Les communications collectives.
  - Des groupes de processus.
- Il existe plusieurs implémentations de cette API:
  - Développées par les constructeurs pour leur plate-formes.
  - Disponible en open source (OpenMPI, Mpich2,…).
- Supportée par de nombreux systèmes et compilateurs (gnu, intel…).

Modèles de Programmation Parallèle

Programmation à mémoire distribuée: échange de messages

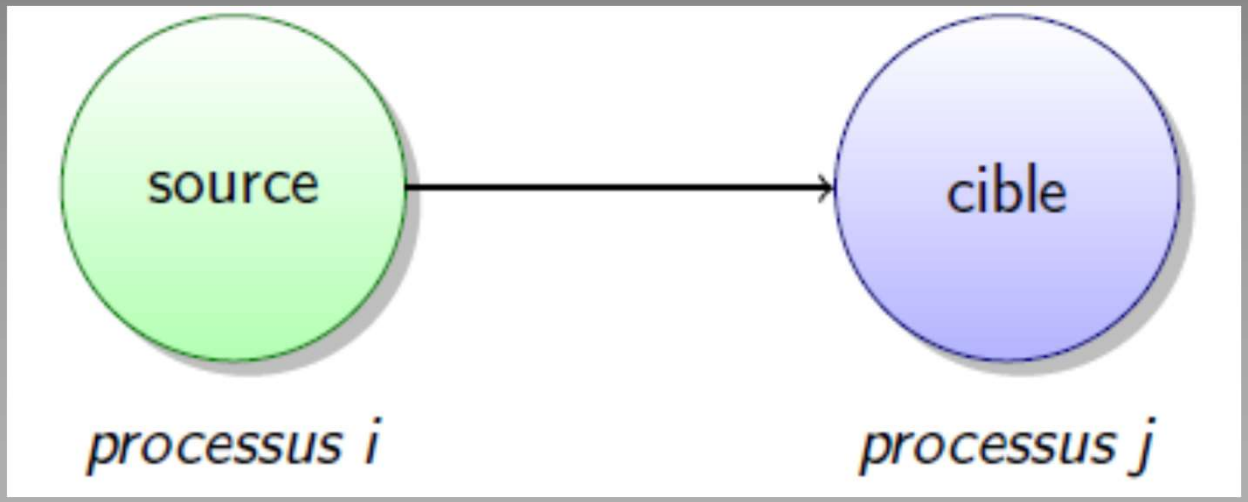
Pour ce type de modèle, on utilise le plus souvent le modèle **échange de messages**:

- Chaque processus exécute un programme sur des données différentes.
- Chaque processus dispose de ses propres données, sans accès direct à celle des autres.
- Les processus peuvent s'échanger des messages entre eux pour:
  - Transférer des données
  - Se synchroniser

Modèles de Programmation Parallèle

Concepts d'échange de messages

- Un processus i envoie des données source au processus j.
- Le processus j reçoit des données du processus i et les met dans cible.



Modèles de Programmation Parallèle

Modèle de programmation mixte

- Ce modèle consiste à utiliser deux niveaux de parallélisme:
  - Le modèle de parallèlisation pour architecture à mémoire distribuée pour réaliser des tâches peu dépendantes en parallèle sur les différents nœuds du calculateur.
  - Le modèle de parallèlisation pour architecture à mémoire partagée sur chaque nœud du calculateur.
- Le principe consiste alors du multithreading avec OpenMP à l'intérieur de chaque processus MPI.

Modèles de Programmation Parallèle

Modèle de programmation hybride

- Ce modèle consiste à confier au GPUs une partie des calculs
- Outils pour utiliser les GPUs
  - CUDA (Nvidia API).
  - OpenCL (*Open Computing Language*).
  - HMPP (*Hybrid Multicore Parallel Programming*) CAPS enterprise.
  - AMP (*Accelerated Massive Parallelism*) API Microsoft.

2ème Partie:
Programmation Parallèle en C#

Nous Allons Faire:

- Généralités
- Delegates
- Threads
- Tasks
- Parallel.For()

Programmation Parallèle en C#

Qu'est-ce que la programmation parallèle?

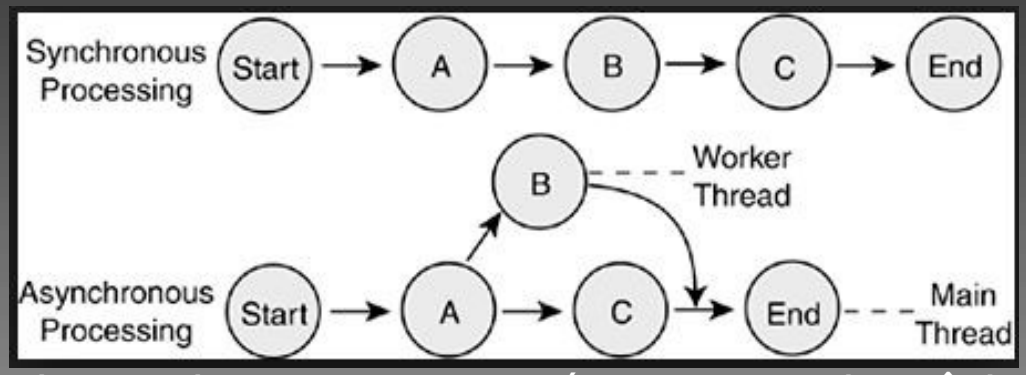
La programmation parallèle est une technique de programmation dans laquelle le flux d'exécution de l'application est décomposé en morceaux qui seront exécutés en même temps (simultanément) par plusieurs cœurs, processeurs ou ordinateurs pour obtenir les meilleures performances.

Avant de discuter de la programmation parallèle, comprenons deux concepts importants.

- Programmation Synchrone (Synchronous programming)
- Programmation Asynchrone (Asynchronous programming)

Programmation Parallèle en C#

Synchrone vs Asynchrone (1/4)



En exécution synchrone, le programme exécute toutes les tâches dans l'ordre.

Cela signifie que lorsque chaque tâche est déclenchée, le programme attend qu'il termine avant de commencer la suivante.

Dans l'exécution asynchrone, le programme n'exécute pas toutes les tâches dans l'ordre. Cela signifie qu'il déclenche les tâches simultanément, puis attend leur fin.

Alors maintenant, la question est, si l'exécution asynchrone prend moins de temps à terminer que l'exécution synchrone, pourquoi quelqu'un choisirait-il l'exécution synchrone?

Programmation Parallèle en C#

Synchrone vs Asynchrone (2/4)

Mode synchrone: chaque tâche s'exécute en séquence, il est donc plus facile de programmer. C'est ainsi que nous le faisons depuis des années.

Inconvénients:

- Il faut plus de temps pour terminer.
- Il peut arrêter le thread d'interface utilisateur (UI)
- Il n'utilise pas l'architecture multicœur des nouveaux processeurs. Peu importe si votre programme fonctionne sur un processeur 1 ou 64, il s'exécutera aussi vite (ou lentement) sur les deux.

Mode asynchrone: La programmation asynchrone élimine ces inconvénients

Toutefois, avec l'exécution asynchrone, vous avez quelques défis:

- Vous devez synchroniser les tâches.
- Vous devez résoudre les problèmes de concurrence.
- Il n'y a plus de séquence logique, vous n'avez pas le contrôle de celui qui finit le premier.

Programmation Parallèle en C#

Synchrone vs Asynchrone (3/4)

Mode synchrone: chaque tâche s'exécute en séquence, il est donc plus facile de programmer. C'est ainsi que nous le faisons depuis des années.

Inconvénients:

- Il faut plus de temps pour terminer.
- Il peut arrêter le thread d'interface utilisateur (UI)
- Il n'utilise pas l'architecture multicœur des nouveaux processeurs. Peu importe si votre programme fonctionne sur un processeur 1 ou 64, il s'exécutera aussi vite (ou lentement) sur les deux.

Mode asynchrone: La programmation asynchrone élimine ces inconvénients

Toutefois, avec l'exécution asynchrone, vous avez quelques défis:

- Vous devez synchroniser les tâches.
- Vous devez résoudre les problèmes de concurrence.
- Il n'y a plus de séquence logique, vous n'avez pas le contrôle de celui qui finit le premier.

Programmation Parallèle en C#

Synchrone vs Asynchrone (4/4)

Alors, choisissez-vous une programmation plus facile ou une meilleure utilisation des ressources?

Heureusement, vous n'avez pas à prendre cette décision. Microsoft a créé plusieurs façons de minimiser les difficultés de programmation pour l'exécution asynchrone.

Les Delegates en C#

Les bases des delegates !

Qu’est-ce qu’un delegate?

- Un delegate est un concept abstrait du C#. Jusqu'à maintenant, une variable pouvait contenir de nombreuses choses. On traite, par exemple, les objets comme des variables. Elles permettent aussi de mettre en mémoire des données, comme du texte, des nombres entiers ou flottants, des booléens. Ces cas ne sont que des exemples.
- Un delegate est en fait une variable un peu spéciale... Elle ne sert qu'à donner une **référence vers une méthode ou fonction**. Elle est donc de type référence, comme un objet !
- L'utilité sera bien évidement d'envoyer ces delegates en paramètres !
- Pensez-y, une méthode générique unique pourrait s'occuper de lancer un nombre infini d'opérations déterminées par vos bons soins.

Les Delegates en C#

Exercices:

- Exemple simple: **Lesson1_ExpSimple**

On voit clairement une situation très usuelle ici. Vous appelez **Test** deux fois à partir du **Main**.

- Exemple d'un delegate: **Lesson2_ExpDelegate**

Dans cet exemple, l'utilisation d'un delegate est carrément inutile, mais ils deviendront rapidement indispensables, surtout lors de la programmation réseau.

Quand vous vous abonnez à un *événement*, vous refilez tout simplement une méthode à une liste de delegate. Quand on invoque un événement, on appelle toutes les méthodes abonnées.

Les Delegates en C#

Les signatures des méthodes:

Toute méthode possède une signature. Une signature de méthode, comme dans le monde réel, sert à identifier une méthode de façon unique.
La signature de méthode résout le problème des noms uniques dans le cas de surcharge d'une méthode. Dans les langages autorisant la surcharge de méthode, on se réfère à la signature plutôt qu'au nom de la méthode pour l'appeler. Pour chaque méthode doit correspondre une signature différente des autres.
Donc, la signature d'une méthode contient les informations suivantes :

- L'identificateur de la méthode
- La séquence des types de la liste des paramètres

Le type de retour ne fait pas partie de la signature !

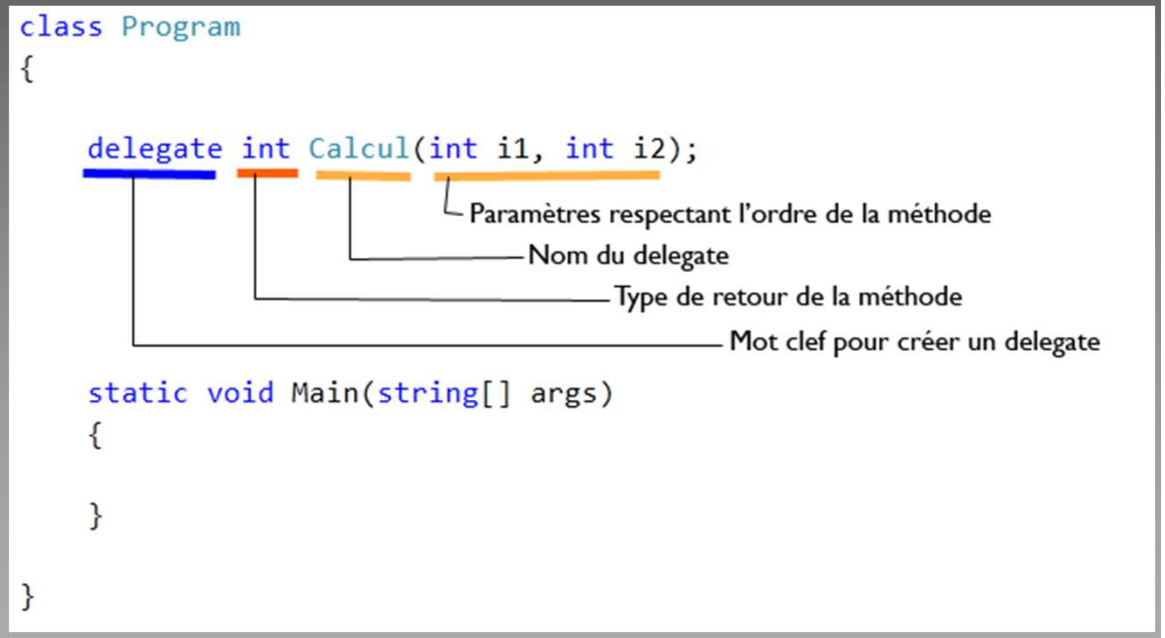
Ce qui fait que notre méthode:
Static public int Test(string test) a la signature suivante: Test (String);

Les Delegates en C#

L’autopsie d’un delegate!

Analysons maintenant comment créer ce fameux *delegate*. Tout d'abord, sachez que le rôle principal d'un *delegate* est de passer, croyez-le ou non, une méthode en paramètre à une autre méthode

La définition d’un delegate:



On fait généralement cela dans le même espace où l’on déclare les *variables globales*. Ensuite, on utilise le *delegate* comme un **objet**.

On peut alors l'utiliser dans les **paramètres** ou ailleurs si nécessaire. Ce sera plus clair pour vous avec l'exemple qui suit.

Les Delegates en C#

Exercices:

- Exemple d'un delegate: **Lesson3_ExpDelegate**

C'est pas du chinois, mais c'est pas simple, hein ?

En effet, ça arrache une grimace la première fois qu'on voit ça, mais en fait, c'est très simple. Comme toutes nos méthodes répondent à la même ***définition*** que notre *delegate*, nous sommes en mesure de toutes les utiliser à l'aide du même. C'est un peu comme de dire qu'un *int* peut égaler 0 ou bien 12154, car les deux répondent à la même définition.

Cela conclut notre introduction aux *delegates*. Il y en a plus que ça à savoir et à comprendre, mais si vous maîtrisez cette partie, c'est excellent.

Si vous avez fait du C ou du C++, cette fonctionnalité ressemble étrangement aux pointeurs sur fonction.

Expression Lambda en C#

Qu'est ce qu'une expression Lambda?

Une expression lambda est une fonction anonyme qui peut contenir des expressions et des instructions. Elle peut ainsi être utilisée pour créer des délégués ou des types d'arborescence d'expression.

Toutes les expressions utilisent l'opérateur lambda **=>** (se lit « conduit à »).

Une expression est toujours constituée de deux parties :

- Le côté gauche donne les paramètres d'entrées (s'il y en a),
- Le côté droit donne les instructions de la méthode anonyme.

Les expressions lambda sont utilisées de deux manières :

- Pour créer des lambda-expression : il s'agit d'expression qui sont utilisée pour créer des arborescences d'expressions (surtout utilisé pour exécuter des instructions dynamiquement, idéal pour créer des filtres de données avec Linq par exemple)
- Pour créer des lambda-instructions : il s'agit de l'équivalent d'un delegate en C# avec une syntaxe différente.

Expression Lambda en C#

Exemples

Voici un exemple de Lambda-expression:

```
Delegate int DelegateType(int i);
Public void Main()
{
   DelegateType  myDelegate = x =>{ x*x;}
    Int square = myDelegate(5); // square vaut 25
}
```

Voici un exemple de Lambda-instruction:

```
Delegate void DelegateInstruction(string s);
Public void Main()
{
    DelegateInstruction myFunc = str =>
                                        {Console.WriteLine(«Entree = {0}», str;}
    myFunct(«Hello»); // affiche «Entree = Hello»
}
```

Multithreading en C#

Les threads, enfin !

Les threads sont des exécutions que l'on sépare de l'exécution principale pour les raisons suivantes :

- Tâche exigeante (gros calculs, gros traitements, etc.)...
- Tâche détachée (impression, recherche, etc.)...
- Tâche bloquante (ça sent le réseau ici !)...

Multithreading en C#

Qu'est-ce qu'un Thread?

Un thread est la plus petite unité de code à laquelle un système d'exploitation alloue le temps CPU.

En multithreading, un seul processus a plusieurs threads d'exécution. Si le système a plusieurs processeurs, il peut fonctionner en parallèle.

Avantages de la programmation multithread ou asynchrone:

- Supposons que vous avez un programme qui vérifie des douzaines de sites Web pour obtenir des informations sur les prix d'un produit. Donc dans ce cas, si le programme fait tout en un seul thread:
  - Le programme principal est bloqué jusqu'à la fin de la recherche sur le Web.
  - L'interface utilisateur est bloquée et l'utilisateur ne peut rien faire.

  Mais si vous exécutez l'interface Web et effectuez une recherche dans différents threads, le programme peut rester réactif même lorsque la recherche est en cours.
- Le multithreading peut également simplifier votre code

Multithreading en C#

Différentes méthodes pour le multithreading:

Le framework .net fournit plusieurs méthodes pour le multithreading.

- PLINK
- Background worker
  - Ce composant exécute le code sur un thread distinct. Il utilise des événements pour communiquer avec le thread d'interface utilisateur principal.
- Task Parallel Library (TPL)
  - Ces outils vous permettent d'exécuter facilement plusieurs méthodes dans différents threads ou d'exécuter plusieurs instances de la même méthode avec des paramètres différents.
- Tasks
  - La classe task vous permet de créer et d'exécuter des threads.
- Threads
  - La classe Thread vous donne un accès de niveau inférieur aux threads.

Multithreading en C#

Threads en C# (1/2):

Le framework .net fournit le namespace “System.Threading.Thread” pour travailler avec les Threads en C#.

- Properties of Thread:
  - **IsAlive**: Returns True when the thread is started but not stopped
  - **IsBackground**: Returns whether the Thread is a Background Thread or not
  - **Priority**: Determines threads priority, i.e. Highest, Normal, Lowest etc.
  - **ThreadState**: Returns the thread state, i.e. Aborted, Running, Stopped, Suspended, Background etc.

Multithreading en C#

Threads en C# (2/2):

- Methods in Thread Class:
  - **Start**: Starts the Thread
  - **Sleep**: Suspends the Thread for specified amount of time.
  - **Abort**: To Terminate/Stop the Thread
  - **ResetAbort**: Cancels an Abort for current thread.
  - **Join**: It is called on the main thread to let it wait until the other thread finishes.
  - **Yield**: Yields execution to another thread if one is ready to run.

Multithreading en C#

Comment créer un Thread en C# (1/2):

```
using System;
using System.Threading;
```

Déclaration et instanciation du thread:

```
Thread t = new Thread(new ThreadStart(ThreadMethod));
```

C'est ici que l'on crée notre objet qui va représenter notre tâche. Le constructeur de la classe *Thread* introduit un concept spécifique de .NET, les délégués. Pour résumer, considérons un délégué comme un pointeur de méthode. Le délégué ***ThreadStart*** prend comme argument le nom d'une méthode que le thread va exécuter lorsqu'il sera lancé. La méthode passée en argument sera appelée lorsque l'on invoquera la méthode Start de notre thread. C'est dans cette méthode ***ThreadMethod*** que le travail que l'on veut paralléliser devra être introduit.

Multithreading en C#

Comment créer un Thread en C# (2/2):

Lancement du thread: **t.Start();**

La méthode Start permet de lancer le thread, et d'exécuter la méthode Déléguée par *ThreadStart*.

Destruction du thread: **t.Abort();**

A tout moment, on peut être amené à vouloir explicitement détruire des threads, par exemple lors de la fermeture du programme, histoire de faire les chose proprement. La méthode la plus sèche pour stopper un thread se nomme Abort.

Celle-ci tue le thread et lève une exception du type ThreadAbortException.

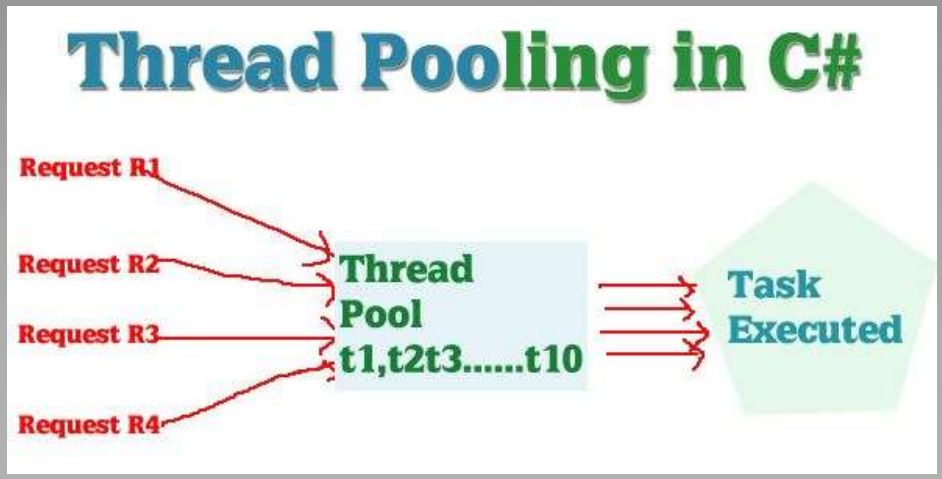
Suspend le thread: **t.Suspend();**

Le thread reprend son activité: **t.Resume();**

Multithreading en C#

Exercices (1/2):

- Création d'un thread 1: Lesson4_MTCreate
- Création d'un thread 2: Lesson5_MTCreate
- Création d'un Background thread: Lesson6_MTBackGrd
- Comment stopper un thread: Lesson7_MTStop
- Un Pool de threads: Lesson8_MTPool



Multithreading en C#

Exercices (2/2):

Création d'un thread avec paramètres: **Lesson9_MTParam**

Attention:

la plus fréquente source d'erreur lors de l'utilisation de ce genre de thread est le paramètre. En effet, ce type de delegate a sa propre définition et il requiert un seul et unique paramètre qui sera de type object. Faîtes bien attention à transtyper (cast) vos variables correctement !

On voit très bien que dans ce cas-ci, le Thread B a terminé en premier, ce qui prouve que le même code peut générer des résultats différents d'une exécution à l'autre, s'il est codé avec des threads !

Multithreading en C#

Cas particuliers:

Même si un thread s'exécute en deçà de votre programme principal, il reste que la méthode qu'il exécute fait partie de la classe à laquelle la méthode appartient. Cela signifie que l'accès aux variables globales et membres de votre classe lui seront accessibles sans problème.

Là où le problème se pose, c'est lorsque plusieurs threads devront accéder à la même variable, y faire des changements et des tests.

Il est important de bien synchroniser ses threads, surtout si l'on aspire à commercialiser le produit. Ainsi, plusieurs structures de synchronisation existent, et nous allons en survoler quelques unes.

Multithreading en C#

Les mécanismes de synchronisation

- Les variables de contrôle
- Le lock
- Les Mutex
- SemaphoreSlim
- Join

Multithreading en C#

Les variables de contrôle

Une variable de contrôle est en réalité une variable globale que seul le thread principal modifiera et que les threads enfants contrôleront. Ce concept est particulièrement efficace dans le cas où le thread effectue une boucle infinie. Ça sent la programmation réseau ici.

Exercice : Lesson10_MTVarControle

Ce qu'il faut comprendre de cet exemple, c'est que les variables de contrôle sont une bonne méthode afin d'influencer le comportement d'un thread, mais généralement seulement lorsque celui-ci est en boucle. Aussi, il est TRÈS important de retenir que seul le thread principal doit modifier la valeur de cette variable. Sinon, on pourrait retrouver des threads à toutes les sauces.

Avez-vous remarqué un bout de code qui ne vous semblait pas thread-safe ?

```
int id = ++_identificateur;
```

Ce bout de code n'est pas thread-safe, car on ne sait pas si un autre processus pourrait prendre le contrôle au mauvais moment. Si l'ordre de lancement est très important, cette ligne pourrait ne pas s'exécuter à temps.

Multithreading en C#

Le lock

L'instruction *lock* permet de verrouiller efficacement une ressource tant et aussi longtemps qu'un bloc d'instruction est en cours. Cela signifie que si d'autres threads tentent d'accéder à la même ressource en même temps, ils ne pourront pas. Cela ne signifie pas qu'ils planteront et se termineront, mais plutôt qu'ils passeront le jeton à un autre thread et attendront patiemment leur tour afin d'accéder à cette ressource.

Exercice : Lesson11_MTLock

Donc, dans cet exemple, on voit que tout est bien protégé. Aucun thread ne peut venir interférer avec les autres. Remarquez la création d'une instance d'un objet de Type **Object** à la ligne 12. Cela est notre témoin de verrouillage. En réalité, n'importe quel objet qui se passe en référence peut servir de témoin de verrouillage. Comme nous avons travaillé avec des **int** dans cet exemple et que ce type est passé par valeur, nous avons eu à créer cette variable.

Multithreading en C#

Les Mutex

Les Mutex sont excessivement similaires aux **lock**. Cependant, si vous désirez créer de nombreuses sections critiques indépendantes, les Mutex ont l'avantage d'être sous forme d'objets plutôt que d'instructions. Un petit exemple vous éclaira sur l'utilisation des Mutex.

Exercice : Lesson12_MTMutex

Si vous avez fait un peu de programmation en Win32 (langage C), vous pouvez voir la lignée directe des Mutex du .NET et des CRITICAL_SECTION du Win32. Sinon, vous voyez que les Mutex ont la même fonction que l'instruction **lock** en un peu plus verbeux. Je tiens cependant à vous avertir que de ne pas relâcher un Mutex peut faire planter votre application, donc faîtes attention à cela.

Multithreading en C#

SemaphoreSlim

Le SemaphoreSlim sert à contrôler l'accès d'une ressource limitée. Jusqu'à maintenant, les mécanismes de synchronisation dont nous avons parlé ont surtout servi à limiter une ressource à un accès mutuellement exclusif entre des threads concurrents. Qu'en est-il si l'on veut partager une ressource, mais à travers plusieurs threads simultanément ? Cependant, on aimerait garder un nombre maximal d'accès concurrent à la ressource. Les sémaphores existent pour cette raison. Ils sont plutôt utilisé lors de courtes durées d'attente.

Exercice : **Lesson13_MTSemaphoreSlim**

Multithreading en C#

Le Join()

C'est le dernier mécanisme de synchronisation dont je parlerai. Il s'agit très simplement d'attendre la fin d'un autre thread afin de continuer le thread dans lequel le *Join()* est défini. Cela en fait une méthode bloquante qui pourrait vous causer des problèmes en Windows Forms.

Exercice : **Lesson14_MTJoin**

Après avoir vu comment bien synchroniser ses threads, voyons ce que vous ne devez PAS faire !!!

Le Abort()

Exercice : **Lesson15_MTAbort**

cela semble assez facile à faire, et semble sans grandes conséquences. Si vous étiez en train de faire quelque chose de vraiment important, et que le thread principal choisirait ce moment pour arrêter le thread, l'application en entier pourrait devenir instable et planter. Puisqu'on ne veut pas cela, il vaut mieux utiliser le *Join()* et une variable de contrôle, comme dans l'exemple suivant :

Exercice : **Lesson16_MTAbort**

Tasks en C#

Qu'est-ce que Task en C# ?

.net Framework fournit la class: ***System.Threading.Tasks.Task***

- Pour vous permettre de créer des threads et les exécuter de manière asynchrone.
- Task est un objet qui représente un travail qui devrait être accompli.
- Task peut vous dire si le travail est terminé et si l'opération renvoie un résultat, peut aussi vous donner le résultat.

Création et exécution explicites de tâches

Exercice : **Lesson17_MTTaskCreate**

Création et exécution implicites de tâches

Exercice : **Lesson18_MTTaskCreate**

Tasks en C#

Task Parallel Library (TPL) en C# (1/5)

La bibliothèque parallèle de tâches (TPL) est basée sur le concept de *tâche*, qui représente une opération asynchrone. À certains égards, une tâche ressemble à un thread ou à un élément de travail **ThreadPool**, mais à un niveau d'abstraction supérieur. Le terme *parallélisme des tâches* fait référence à une ou plusieurs tâches indépendantes qui s'exécutent simultanément. Les tâches présentent deux grands avantages:

- Une utilisation plus efficace et évolutive des ressources système.
- Davantage de contrôle par programmation qu'avec un thread ou un élément de travail.

Pour ces raisons, la bibliothèque parallèle de tâches est l'API privilégiée pour l'écriture du code multithread, asynchrone et parallèle dans .NET Framework.

Tasks en C#

Task Parallel Library (TPL) en C#(2/5)

Vous pouvez utiliser la méthode **TaskFactory.StartNew** pour créer et lancer une tâche dans une opération. Utilisez cette méthode quand la création et la planification n'ont pas besoin d'être séparées.

Exercice : **Lesson19_MTTaskFactory**

L'exemple suivant montre comment utiliser le type **System.Threading.Tasks.Task<TResult>** pour retourner une valeur à partir de la propriété **Result.**

Exercice : **Lesson20_MTTaskFactory**

La propriété **Result** bloque le thread appelant jusqu'à ce que la tâche se termine.

Tasks en C#

Task Parallel Library (TPL) en C#(3/5)

l'exemple suivant, parce que les tâches sont de Type **System.Threading.Tasks.Task<TResult>**, chacune d'elle a une propriété **Task<TResult>.Result** publique contenant le résultat du calcul. Les tâches sont exécutées de façon asynchrone et peuvent se terminer dans n'importe quel ordre. Si la propriété Result est accessible avant la fin du calcul, la propriété bloque le thread appelant jusqu'à ce que la valeur soit disponible.

Exercice : **Lesson21_MTTaskFactory**

Tasks en C#

Task Parallel Library (TPL) en C#(4/5)

Lorsque vous utilisez une expression lambda pour créer le délégué d'une tâche, vous avez accès à toutes les variables qui sont visibles à ce stade dans votre code source. Toutefois, dans certains cas, notamment dans les boucles, une expression lambda ne capture pas la variable comme prévu. Elle capture uniquement la valeur finale, pas la valeur qui change au cours de chaque itération. L'exemple de code suivant illustre le problème. Il passe un compteur de boucles à une expression lambda qui instancie un objet **CustomData** et utilise le compteur de boucles comme identificateur de l'objet.

Exercice : **Lesson22_MTTaskFactory**

Tasks en C#

Task Parallel Library (TPL) en C#(5/5)

Vous pouvez accéder à la valeur de chaque itération en fournissant un objet d'état à une tâche via son constructeur. L'exemple suivant modifie l'exemple précédent en utilisant le compteur de boucles lors de la création de l'objet **CustomData** qui, à son tour, est passé à l'expression lambda. Comme le montre la sortie de l'exemple, chaque objet **CustomData** possède maintenant un identificateur unique basé sur la valeur du compteur de boucle au moment où l'objet a été instancié.

Exercice : **Lesson23_MTTaskFactory**

L'exemple suivant est une variante de l'exemple précédent. Il utilise la propriété **AsyncState** pour afficher des informations sur les objets **CustomData** passés à l'expression lambda.

Exercice : **Lesson24_MTTaskFactory**

Parallel.for en C#

Parallel Class: Divise le travail entre les processeurs en C#

La classe Parallel est utilisée dans le:

namespace System.Threading.Tasks

pour affecter des tâches à planifier automatiquement et attendre qu'elles soient terminées.

La classe Parallel s'adapte automatiquement au nombre de processeurs.

Traitement des données en parallèle:

Lorsque vous disposez d'un ensemble de données pouvant être réparties sur plusieurs processeurs et traitées indépendamment, vous pouvez utiliser des constructeurs tels que:

Parallel.For() & Parallel.ForEach()

Parallel.for en C#

Parallel.For()

Boucle **for** séquentielle :

```
int n = ...
for (int i = 0; i < n; i++) {
   // ...
}
```

Boucle **Parallel.For** :

```
int n = ...
Parallel.For(0, n, i => {
   // ...
});
```

Signature de la méthode **Parallel.For** :

```
Parallel.For(int fromInclusive,
             int toExclusive,
             Action<int> body);
```

Dans l'exemple, les deux premiers arguments spécifient les limites d'itération. Le premier argument est l'indice le plus bas de la boucle. Le deuxième argument est la limite supérieure exclusive ou l'index le plus grand plus un. Le troisième argument est une action invoquée une fois par itération. L'action prend l'index de l'itération comme argument et exécute le corps de la boucle une fois pour chaque index.

Parallel.for en C#

Parallel.ForEach()

Boucle **foreach** séquentielle:

```
IEnumerable<MyObject> myEnum = ...
foreach (var obj in myEnum) {
   // ...
}
```

Boucle **Parallel.ForEach**:

```
IEnumerable<MyObject> myEnum = ...
Parallel.ForEach(myEnum, obj => {
   // ...
});
```

Signature de la méthode **Parallel.ForEach**:

```
Parallel.ForEach(IEnumerable<TSource> source, Action< TSource >
body);
```

Parallel.for en C#

Parallel.For()

Cette rubrique contient trois exemples qui illustrent la méthode.

- Le premier exemple, **Lesson25_MTParallelFor**, calcule la taille des fichiers dans un répertoire unique.
- Le deuxième, **Lesson26_MTMultMatrices**, calcule le produit de deux matrices.
- Le troisième, **Lesson27_MTPrimeNumbers**, calcule des nombres premiers.

Parallel.for en C#

Lesson25_MTParallelFor:

Cet exemple est un utilitaire en ligne de commande simple qui calcule la taille totale des fichiers d'un répertoire. Il attend un chemin d'accès de répertoire unique en tant qu'argument et indique le nombre et la taille totale des fichiers contenus dans ce répertoire. Après avoir vérifié que le répertoire existe, il utilise la méthode Parallel.For() pour énumérer les fichiers dans le répertoire et déterminer leur taille. Chaque taille de fichier est ensuite ajoutée à la variable totalSize.

Notez que l'addition est effectuée en appelant la méthode Interlocked.Add, pour être exécutée comme une opération atomique. Dans le cas contraire, plusieurs tâches pourraient essayer de mettre à jour la variable totalSize simultanément.

Parallel.for en C#

Lesson26_MTMultMatrices:

Cet exemple utilise la méthode **Parallel.For()** pour calculer le produit de deux matrices. Il montre également comment utiliser la classe **System.Diagnostics.Stopwatch** pour comparer les performances d'une boucle parallèle avec une boucle non parallèle. Étant donné qu'il peut générer un important volume de sortie, l'exemple permet de rediriger la sortie vers un fichier.

Parallel.for en C#

Lesson27_MTPrimeNumbers:

Un **nombre premier** est un **entier naturel** qui admet exactement deux **diviseurs** distincts entiers et **positifs** (qui sont alors 1 et lui-même). Ainsi, 1 n'est pas premier car il n'a qu'un seul diviseur entier positif ; 0 non plus car il est divisible par tous les entiers positifs.

Nombre	Diviseurs
6	6·3·2·1
7	7·1
8	8·4·2·1
9	9·3·1

Parallel.for en C#

Parallel.ForEach()

Cette rubrique contient deux exemples qui illustrent la méthode.

- Le premier exemple, **Lesson28_MTParallelForEach,** Cet exemple indique comment utiliser une boucle Parallel.ForEach() pour permettre le parallélisme des données sur toute source de données **System.Collections.Ienumerable** ou **System.Collections.Generic.Ienumerable<T>**

Une boucle **ForEach** fonctionne comme une boucle **For**. La collection source est partitionnée et le travail est planifié sur plusieurs threads en fonction de l'environnement système. Plus les processeurs du système sont nombreux, plus l'exécution de la méthode parallèle est rapide. Pour certaines collections source, une boucle séquentielle peut être plus rapide, en fonction de la taille de la source et du genre de travail exécuté.

Parallel.for en C#

Parallel.ForEach()

- Le deuxième exemple, **Lesson29_MTParallelForEach**:

Cet exemple montre comment écrire une méthode **ForEach** qui utilise des variables de thread local. Quand une boucle **ForEach** s'exécute, elle divise sa collection source en plusieurs partitions. Chaque partition obtient sa propre copie de la variable « thread local ».

Pour utiliser une variable de thread local dans une boucle **ForEach**, vous devez appeler l'une des surcharges de méthode qui accepte deux paramètres de type. Le premier paramètre de type, **Tsource**, spécifie le type d'élément source, tandis que le second, **TLocal**, spécifie le type de la variable de thread local.

Parallel.for en C#

Remarque importante sur Parallel.For()

Lors de la parallélisation du code, y compris des boucles, un objectif important consiste à utiliser les processeurs le plus possible sans surparalléliser jusqu'au point où la surcharge de traitement en parallèle réduit les performances. Dans cet exemple, seule la boucle externe est parallélisée, car peu de travail est effectué dans la boucle interne. La combinaison d'une petite quantité de travail et des effets indésirables du cache peut entraîner une dégradation des performances dans les boucles parallèles imbriquées. Par conséquent, paralléliser uniquement la boucle externe est la meilleure façon d'optimiser les avantages offerts par l'accès concurrentiel sur la plupart des systèmes.

3ème Partie:
Communication Interprocessus en C#

- En général, plusieurs processus (threads) coopèrent pour réaliser une tâche complexe.
- Ces processus s'exécutent en parallèle sur un même ordinateur (monoprocesseur ou multiprocesseurs) ou sur des ordinateurs différents.
- Ils doivent s'échanger des informations (communication interprocessus).
- Il existe plusieurs mécanismes de communication interprocessus. Nous pouvons citer :
  - les données communes (variables, fichiers, segments de données),
  - les signaux,
  - les messages

Communication Interprocessus en C#

Echange des messages (1/3)

L'**échange de messages** est un autre moyen pour faire communiquer deux processus. Avec celui-ci, les processus s'échangent des blocs de données de taille fixe, appelés **messages**. Ce message contient des données, qui peuvent être de tout type simple (entier positif ou négatif, flottant positif ou négatif, caractère isolé ou chaine de caractères) ou une combinaison de ces éléments (plusieurs nombres, des nombres et une chaine de caractères, …). De plus, il contient un type, un entier positif, choisi par le programmeur, qui indique quel est le contenu du message. Il n'y a que trois restrictions sur le contenu du message :

- le type doit être présent.
- le format des données doit être connu.
- la taille des données doit être connue pour pouvoir être stockée par le processus qui recevra le message.

Communication Interprocessus en C#

Echange des messages (2/3)

Sur certaines implémentations, les messages sont envoyés directement, sans aucune mise en attente. Le processus destinataire du message est alors prévenu qu'on souhaite communiquer avec lui et doit arrêter ce qu'il fait pour échanger avec l'émetteur du message. Dit autrement, les deux processus doivent se synchroniser pour l'échange. On parle alors tout simplement de **passage de message**. Cela fonctionne bien quand les deux processus ne sont pas sur le même ordinateur et communiquent entre eux via un réseau local ou par Internet.

Avec les **canaux** et les **files de messages** (*message queue* en anglais ou MSQ en abrégé), le processus destinataire d'un message et son émetteur n'ont pas besoin de se synchroniser. Il est dès lors possible d'envoyer un message sans avoir à prévenir le processus destinataire. Pour cela, les messages que s'envoient les processus sont mis en attente dans une file d'attente (tant que celle-ci n'est pas remplie, auquel cas le processus émetteur se bloque). Les processus destinataires peuvent consulter cette file de message quand ils le souhaitent, notamment pour voir si on leur a envoyé des données (si la file d'attente est vide, ils se bloquent en attendant des données qui leur sont destinés).

Communication Interprocessus en C#

Echange des messages (3/3)

Dans tous les cas, les données envoyées via la file d'attente doivent de préférence être récupérées dans l'ordre d'envoi. Pour cela, le processus récupère systématiquement le message le plus ancien. Ainsi, les messages sont triés dans leur ordre d'envoi et sont retirés dans cet ordre par le processus destinataire. Notez que j'ai bien dit « retiré » et non pas « lus » : une fois qu'un message est consulté, il est retiré de la file. C'est ce qu'on appelle un fonctionnement « FIFO » pour *First In First Out*, premier entré, premier sorti.

Communication Interprocessus en C#

Les Canaux

Le premier type de file d'attente est celui des canaux. La différence entre les deux tient dans le fait que les canaux permettent à seulement deux processus de communiquer, qui plus est dans un seul sens, là où une file de message ne souffre pas de ces limites.

Il faut donc deux canaux pour que les deux processus s'échangent des informations dans les deux sens : l'un fonctionnant dans un sens et le second dans l'autre.

Il existe deux types de canaux :

- les canaux anonymes
- les canaux nommés.

Communication Interprocessus en C#

Les Canaux anonymes(1/2)

Les canaux anonymes fournissent une communication entre processus sur un ordinateur local. Ils offrent moins de fonctionnalités que les canaux nommés, mais nécessitent également moins de surcharge. Vous pouvez utiliser les canaux anonymes pour établir plus facilement une communication entre processus sur un ordinateur local. Vous ne pouvez pas utiliser les canaux anonymes pour la communication sur un réseau. Pour implémenter des canaux anonymes, utilisez les classes: AnonymousPipeServerStream et AnonymousPipeClientStream.

Communication Interprocessus en C#

Les Canaux anonymes(2/2)

L'exemple suivant montre une façon d'envoyer une chaîne à partir d'un processus parent à un processus enfant à l'aide de canaux anonymes. Cet exemple crée un objet AnonymousPipeServerStream dans un processus parent avec une valeur PipeDirection de Out. Le processus parent crée ensuite un processus enfant à l'aide d'un handle client pour créer un objet AnonymousPipeClientStream. Le processus enfant a une valeur PipeDirection de In.

Le processus parent envoie ensuite une chaîne fournie par l'utilisateur au processus enfant. La chaîne est affichée sur la console dans le processus enfant.

Exemple:

Lesson30_APipeServer: présente le processus serveur

Lesson30_APipeClient : présente le processus client

Communication Interprocessus en C#

Les Canaux nommés(1/3)

Les canaux nommés fournissent la communication entre un serveur de canaux et un ou plusieurs clients de canaux. Ils offrent plus de fonctionnalités que les canaux anonymes, qui fournissent la communication entre processus sur un ordinateur local. Les canaux nommés prennent en charge la communication en duplex intégrale sur un réseau et plusieurs instances de serveur, la communication basée sur message et l'emprunt d'identité du client, ce qui permet aux processus de connexion d'utiliser leur propre jeu d'autorisations sur des serveurs distants. Pour implémenter des canaux nommés, utilisez les classes:

NamedPipeServerStream **et** NamedPipeClientStream.

Communication Interprocessus en C#

Les Canaux nommés(2/3)

Exemple: **Lesson31_NPipeServer**: présente le processus serveur
L'exemple illustre la création d'un canal nommé à l'aide de la classe NamedPipeServerStream. Dans cet exemple, le processus serveur crée quatre threads. Chaque thread peut accepter une connexion cliente. Le processus client connecté fournit ensuite au serveur un nom de fichier. Si le client dispose des autorisations suffisantes, le processus serveur ouvre le fichier et renvoie son contenu au client.

Exemple: **Lesson31_NPipeClient** : présente le processus client
L'exemple suivant présente le processus client, qui utilise la classe NamedPipeClientStream. Le client se connecte au processus serveur et envoie un nom de fichier au serveur. L'exemple utilise l'emprunt d'identité, donc l'identité qui exécute l'application cliente doit avoir l'autorisation d'accéder au fichier. Le serveur renvoie ensuite le contenu du fichier au client. Le contenu du fichier est ensuite affiché sur la console.

Communication Interprocessus en C#

Les Canaux nommés(3/3)

Les processus client et serveur dans cet exemple sont censés être exécutés sur le même ordinateur, donc le nom du serveur fourni à l'objet NamedPipeClientStream est «.».
Si les processus client et serveur se trouvent sur des ordinateurs distincts, « . » est remplacé par le nom réseau de l'ordinateur qui exécute le processus serveur.

Communication Interprocessus en C#

Les files de messages

MERCI