SPD : Systèmes et Programmation Distribués

Nicolas Szczepanski

16 janvier 2018

szczepanski.nicolas@gmail.com

Partie 1 : Introduction au Parallélisme

Présentation Du Cours

- Objectifs :
  - Suite des cours de réseaux et de systèmes
  - Exploitation d'un système distribué/réparti
  - Avoir des notions de programmation parallèle
    - `pthread` : C/C++
    - Une librairie MPI : C/C++
    - En fonction du temps : Go, OpenMP, ...
- Organisation :
  - 2h CM/TD par semaine
  - 2h de TP (5 séances)
- Évaluation :
  - 2/3 EXAMEN + 1/3 PROJET
  - Projet évalué sur l'ensemble des machines des 2 salles TP

Introduction

Le parallélisme

Définition :

Le **parallélisme** consiste à mettre en œuvre des architectures électroniques permettant de traiter des informations de manière simultanée.

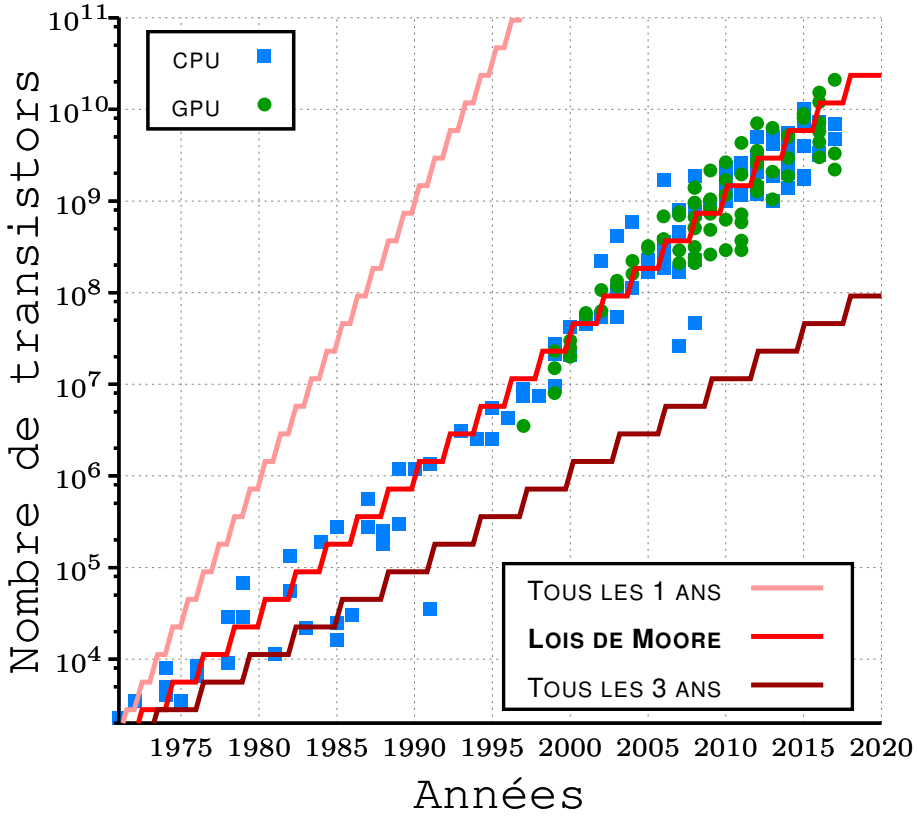
Définition :

Une **instruction** informatique désigne une étape dans un programme informatique. Une instruction dicte à l'ordinateur l'action nécessaire qu'il doit effectuer avant de passer à l'instruction suivante. Un programme informatique est constitué d'une suite d'instructions.

Définitions :

Le **calcul séquentiel** est une exécution étape par étape, où chaque instruction est déclenchée lorsque la précédente est terminée, même si deux instructions sont indépendantes. Ce calcul s'oppose au **calcul parallèle**, où plusieurs instructions peuvent être exécutées simultanément.

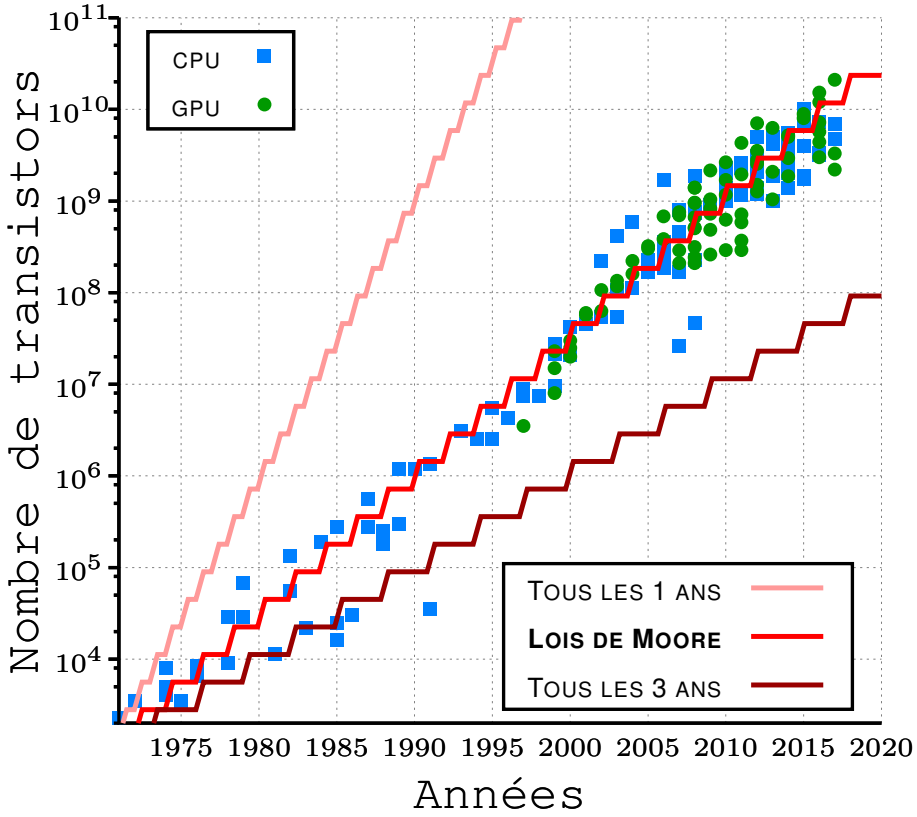
Loi de Moore



Définition : Loi de Moore

Le nombre de transistors des processeurs double tous les deux ans

Loi de Moore

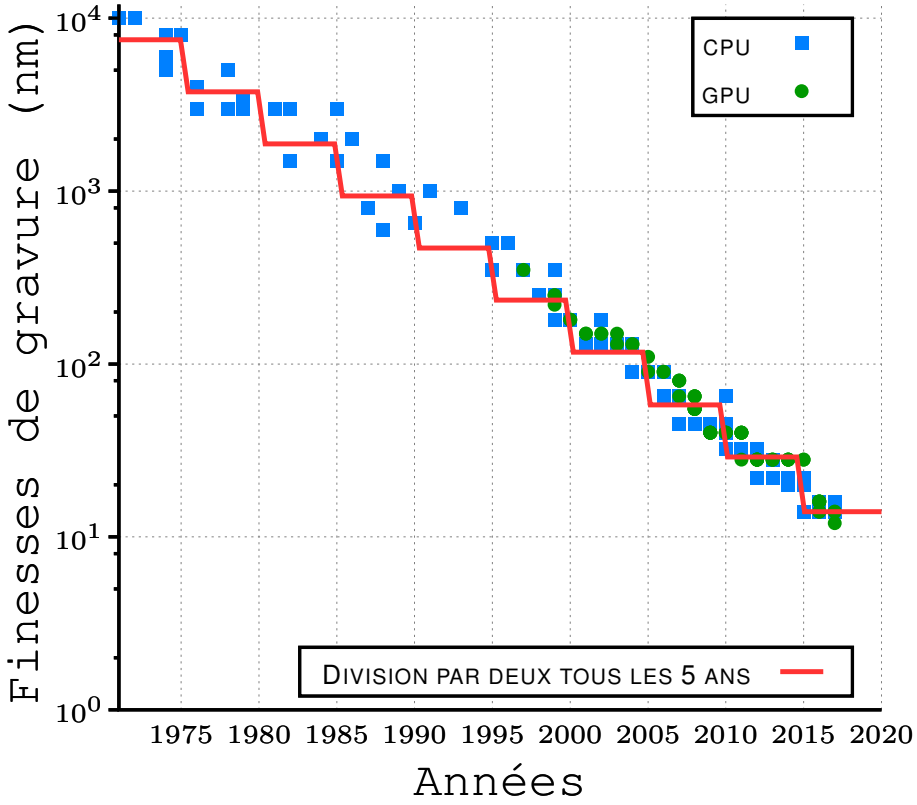


Définition : Loi de Moore

Le nombre de transistors des processeurs double tous les deux ans

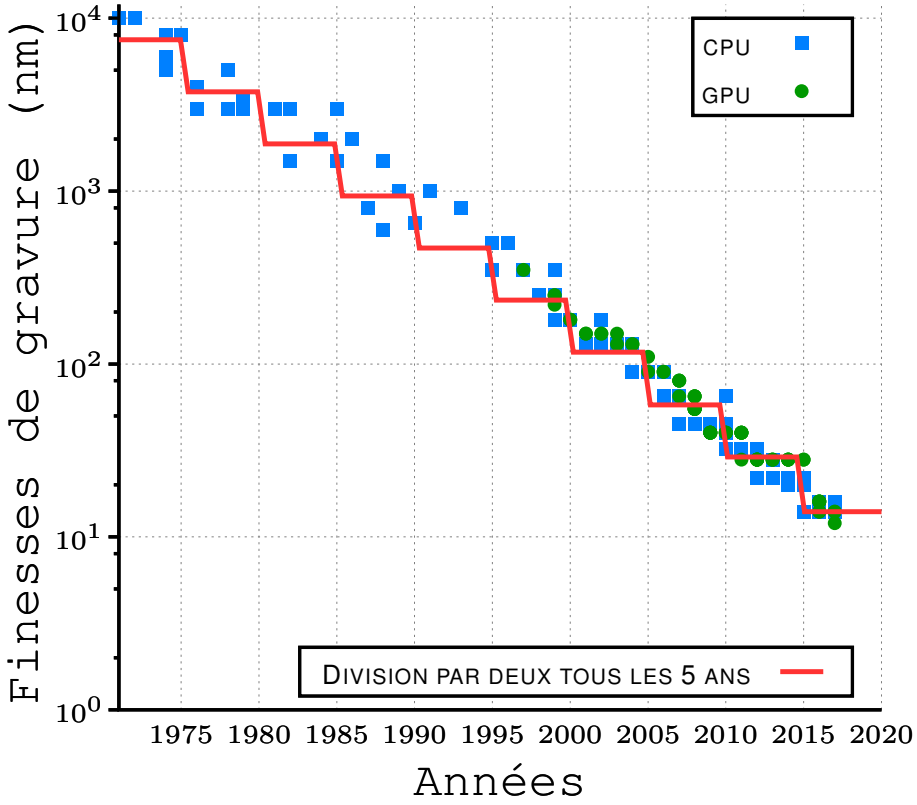
Notamment grâce à la finesse de gravure

Finesse de Gravure



- Épaisseur d'un cheveu humain : 100000 nm
- Processeur Cannonlake d'Intel (début 2018) : 10 nm
- Atome de silicium : 0,1 nm

Finesse de Gravure



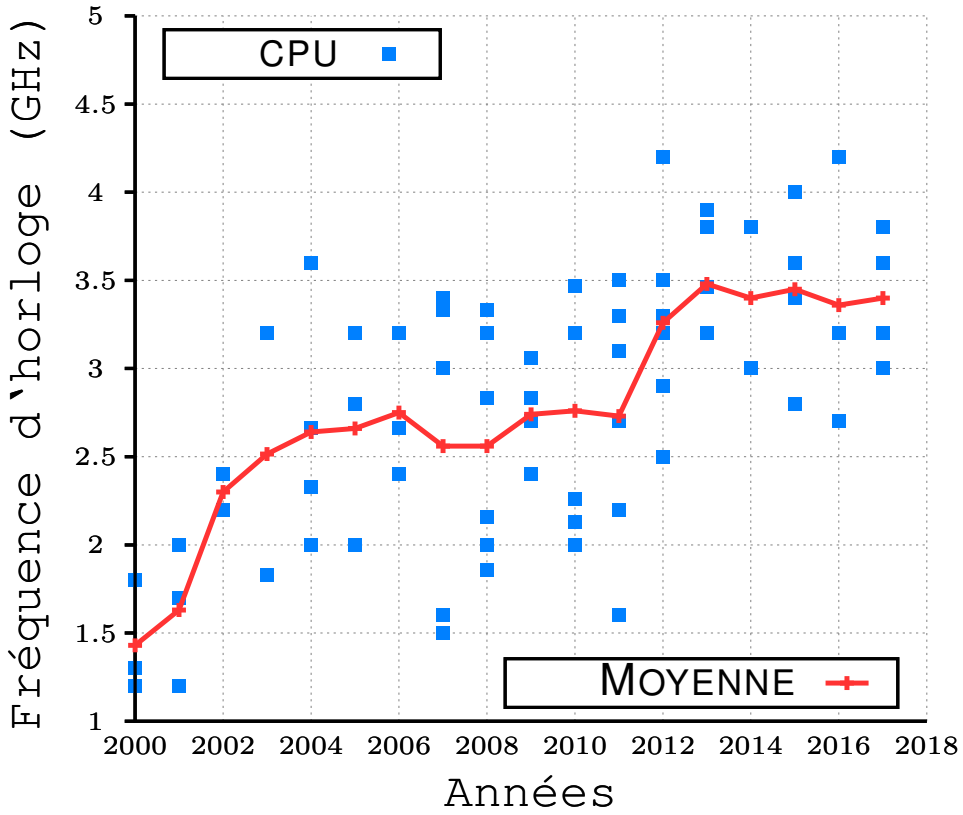
- Épaisseur d'un cheveu humain : 100000 nm
- Processeur Cannonlake d'Intel (début 2018) : 10 nm
- Atome de silicium : 0,1 nm

Problème : la quantité de chaleur dissipée par l'effet Joule

Fréquence d'horloge

L'augmentation de la fréquence :

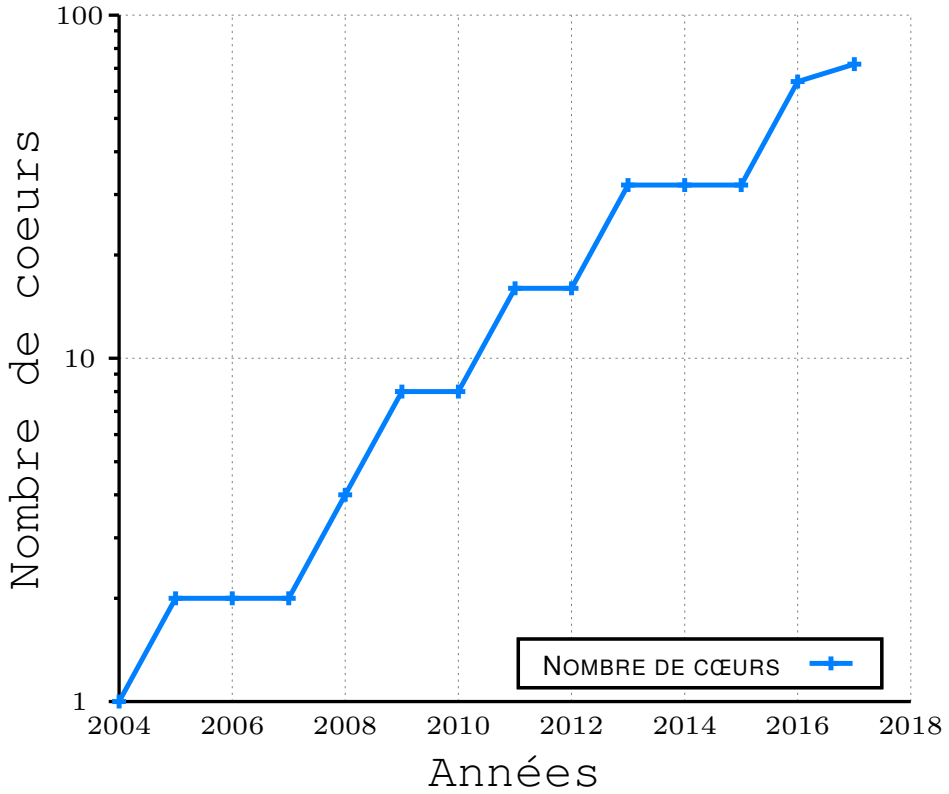
- Faisait face à des problèmes de refroidissement des circuits
- Devenait trop onéreuse



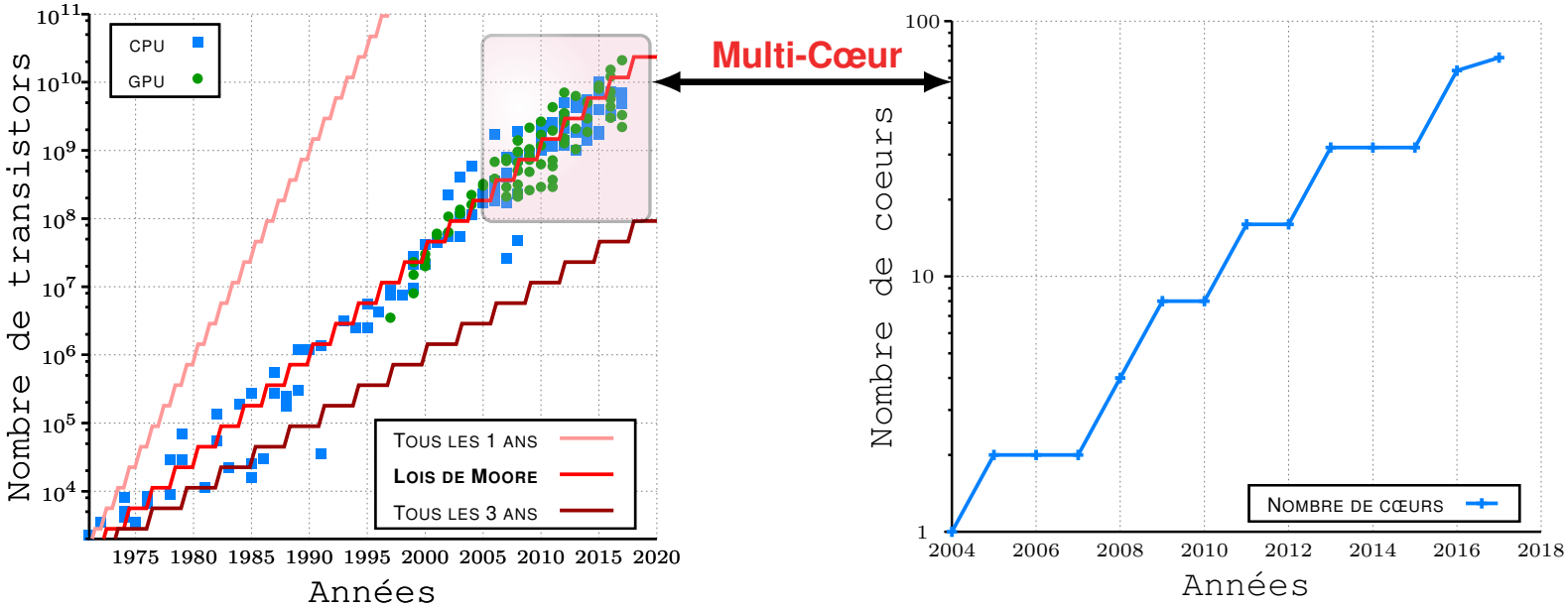
Les Processeurs Multi-Cœurs

La solution : Le multi-cœurs :

- Ne plus augmenter la fréquence d'horloge
- Augmenter le nombre d'opérations exécutées simultanément
- En multipliant le nombre d'unités de calcul d'une machine



Les Processeurs Multi-Cœurs



Exemple :

Les processeurs Intel Xeon Phi sont composés de 57 à 72 cœurs possédant seulement chacun une fréquence d'horloge de 1,1 GHz à 1,5 Ghz.

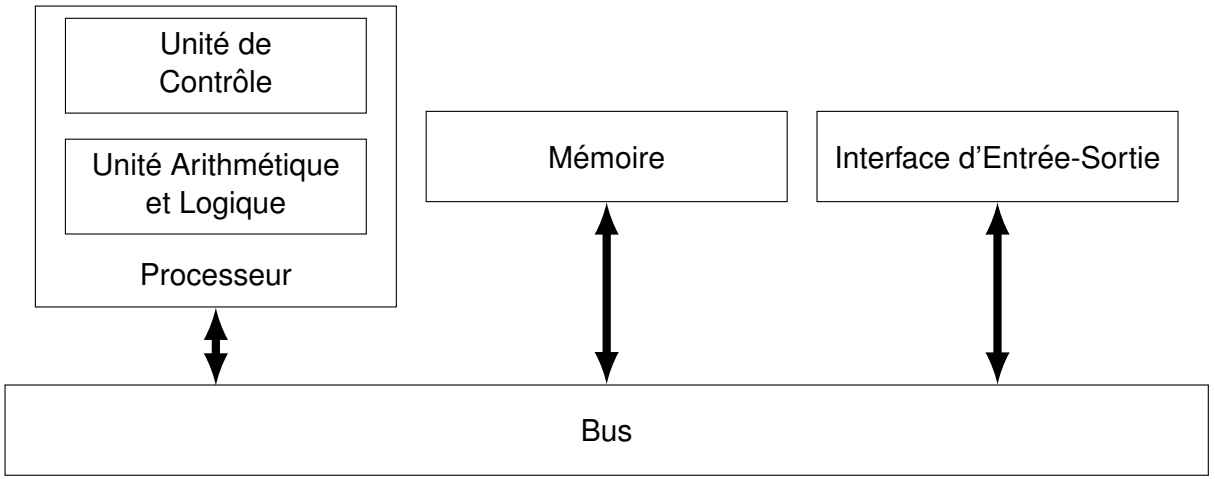
Les Architectures

Architecture de Von Neumann

Définition :

Décompose l'ordinateur en quatre parties distinctes :

- L'**Unité Arithmétique et Logique** (UAL)
- L'**Unité de Contrôle** (UC)
- La **Mémoire** (MEM)
- Les **dispositifs d'Entrée-Sortie**

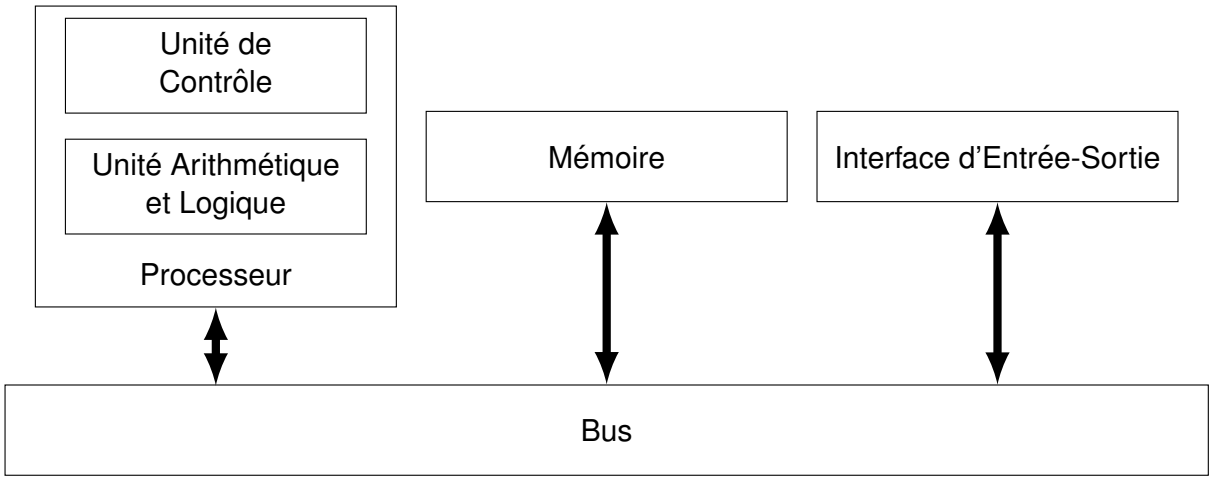


Architecture de Von Neumann

Définition :

Décompose l'ordinateur en quatre parties distinctes :

- L'**Unité Arithmétique et Logique** (UAL)
- L'**Unité de Contrôle** (UC)
- La **Mémoire** (MEM)
- Les **dispositifs d'Entrée-Sortie**



Attente des données : l'unité de calcul est considérablement ralentie

Registre et Mémoire Cache

Définition :

Les **registres** sont le niveau de mémoire le plus proche du processeur. Il s'agit de la mémoire la plus rapide d'un ordinateur, mais leur capacité dépasse rarement quelques dizaines d'octets car la place dans un microprocesseur est limitée. Les programmes écrits en langage de bas niveau peuvent directement utiliser ces registres. Certains registres ont certaines fonctionnalités bien précises comme le compteur ordinal qui contient l'adresse de la prochaine instruction à traiter.

Définition :

Une **mémoire cache** ou **antémémoire** est une mémoire qui sert d'intermédiaire entre une autre source de mémoire et le processeur afin de diminuer le temps d'un accès ultérieur d'un processeur à ces données. Il existe différents niveaux de mémoire cache, qui ont une latence plus faible et une bande passante plus élevée que la mémoire principale.

Mémoires

Mémoire	**Taille** (Bytes)	**Latence** (Cycles d'horloges)	**Temps d'accès** (Nanosecondes)
Registres	4 à 32 Bytes	≤ 1	≤ 1
Cache L1	64 KB par cœurs	~ 4	$\sim 1{,}2$
Cache L2	256 KB par cœurs	~ 10	~ 3
Cache L3	4 MB à 24 MB partagé	$\sim$ 40-75	$\sim$ 12-22
Mémoire principale	1 GB à 16 GB	~ 240	~ 60
Disque dur	1 TB à 8 TB	Non Documenté	~ 4000000

TABLE : Hiérarchie de la mémoire d'un Intel Xeon 5500 caractérisé par la vitesse des accès mémoires et leurs tailles.

Parallélisme au Niveau des Instructions et Pipeline

Définition :

Le **parallélisme au niveau des instructions** (ILP pour *instruction-level parallelism*) est une mesure du nombre d'instructions dans un programme informatique pouvant être exécuté simultanément par une seule unité de calcul (par un seul cœur pour un processeur multi-cœurs). Ce parallélisme peut être matériel ou logiciel.

Définition :

Un **pipeline** ou **chaîne de traitement**, est l'élément d'un processeur dans lequel l'exécution des instructions est découpée en plusieurs étapes. Un pipeline permet le parallélisme au niveau des instructions. La profondeur d'un pipeline représente le nombre de composants nécessaires à l'exécution d'une instruction.

Exemple : pipeline de type RISC classique

Code	Nom	Description
IF	*Instruction Fetch*	Charge l'instruction à exécuter dans le pipeline
ID	*Instruction Decode*	Décode l'instruction et adresse les registres
EX	*Execute*	Exécute l'instruction (par la ou les unités arithmétiques et logiques)
MEM	*Memory*	Dénote un transfert depuis un registre vers la mémoire ou vice-versa
WB	*Write Back*	Stocke le résultat dans un registre

TABLE : Étapes nécessaires afin d'exécuter une instruction par un processeur disposant d'un pipeline de type RISC classique.

Exemple : pipeline de type RISC classique

Considérons l'instruction triviale ADD(R, S_1, S_2) qui affecte le résultat de l'addition des valeurs des registres S_1 et S_2 dans le registre R. Ici, nous voulons calculer l'opération 1+2=3. Ainsi, cette instruction sera dans la mémoire sous le format *Simple MIPS Instruction* possédant 32 bits que nous pouvons visualiser dans ce tableau :

Opération	S_1	S_2	R	Format	Fonction
000000	00001	00010	00011	00000	100000

Cette instruction est composée de cinq étapes dans un pipeline de type RISC classique :

- IF : Récupère l'instruction à partir de la mémoire.
- ID : Détermine quelle est l'instruction et lit les registres
  - 000000 avec 100000 est l'instruction ADD
  - Les contenus de S_1 et S_2 sont 1 et 2
- EX : Ajoute 1 et 2 = 3
- MEM : Ne fait rien pour cette instruction
- WB : Stocke 3 dans le registre R

Exemple : pipeline de type RISC classique

Cycle	1	2	3	4	5	6	7	8	9	10
Inst. 1	IF	ID	EX	MEM	WB					
Inst. 2						IF	ID	EX	MEM	WB

TABLE : Deux instructions exécutées sur un processeur sans pipeline.

Cycle	1	2	3	4	5	6	7	8	9	10
Inst. 1	IF	ID	EX	MEM	WB					
Inst. 2		IF	ID	EX	MEM	WB				

TABLE : Deux instructions exécutées sur un processeur possédant un pipeline de type RISC classique.

Autres exemples :

- Pentium 4 : pipeline d'une profondeur de 20 composants
- Intel Xeon Phi : pipeline particulier et adapté aux multi-cœurs

Problème de Dépendances d'un pipeline

- Les dépendances sont structurelles si deux instructions veulent utiliser la même ressource en même temps (unité de calcul, mémoire, registre, composant, . . .)
- Les dépendances sont de données si deux instructions veulent lire ou écrire dans le même registre ou la même adresse mémoire
- Les dépendances de contrôles sont les exceptions matérielles, interruptions et branchements (saut d'une instruction à une autre). L'adresse de branchement n'est connue que quelques cycles plus tard vers le décodage de l'instruction dans le meilleur des cas. Et le processeur chargera des instructions inutiles dans son pipeline par erreur (exécution spéculative : attaque SPECTRE)

  Lien ver l'explication de l'attaque SPECTRE

Architecture Parallèle : Taxonomie de Flynn

Définition :

La **taxonomie de Flynn** classe les architectures en fonction du nombre de flux d'instructions et de données traité simultanément. Soit il y a un seul flux d'instructions ou de données (*Single*), soit plusieurs (*Multiple*).

		Flux de données	
		Single	*Multiple*
Flux d'instruction	*Single*	SISD	SIMD
	Multiple	MISD	MIMD

TABLE : La taxonomie de Flynn

Architecture Parallèle : Taxonomie de Flynn

- SISD pour ***Single Instruction, Single Data*** :
  Exemple :
- SIMD pour ***Single Instruction, Multiple Data*** :
  Exemple :
- MISD pour ***Multiple Instructions, Single Data*** :
  Peu répandu car pas naturel, voir inexistant.
- MIMD pour ***Multiple Instructions, Multiple Data*** :
  Exemple :

Architecture Parallèle : Taxonomie de Flynn

- SISD pour ***Single Instruction, Single Data*** :
  Exemple : Machine séquentielle à la Von Neumann monoprocesseur et monocœur (mais multitâche simulé !)
- SIMD pour ***Single Instruction, Multiple Data*** :
  Exemple :
- MISD pour ***Multiple Instructions, Single Data*** :
  Peu répandu car pas naturel, voir inexistant.
- MIMD pour ***Multiple Instructions, Multiple Data*** :
  Exemple :

Architecture Parallèle : Taxonomie de Flynn

- SISD pour ***Single Instruction, Single Data*** :
  Exemple : Machine séquentielle à la Von Neumann monoprocesseur et monocœur (mais multitâche simulé !)
- SIMD pour ***Single Instruction, Multiple Data*** :
  Exemple : cartes graphiques (OpenCL, CUDA), instructions vectorielles (MMX, SSE)
- MISD pour ***Multiple Instructions, Single Data*** :
  Peu répandu car pas naturel, voir inexistant.
- MIMD pour ***Multiple Instructions, Multiple Data*** :
  Exemple :

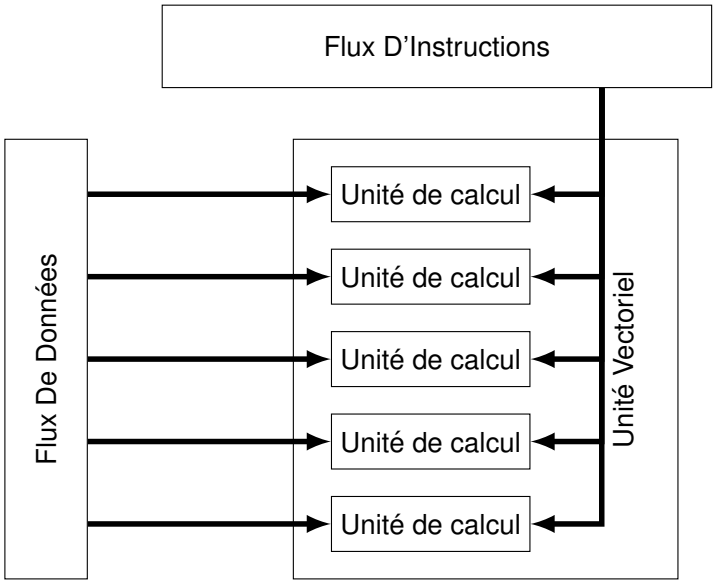
Architecture Parallèle : Taxonomie de Flynn

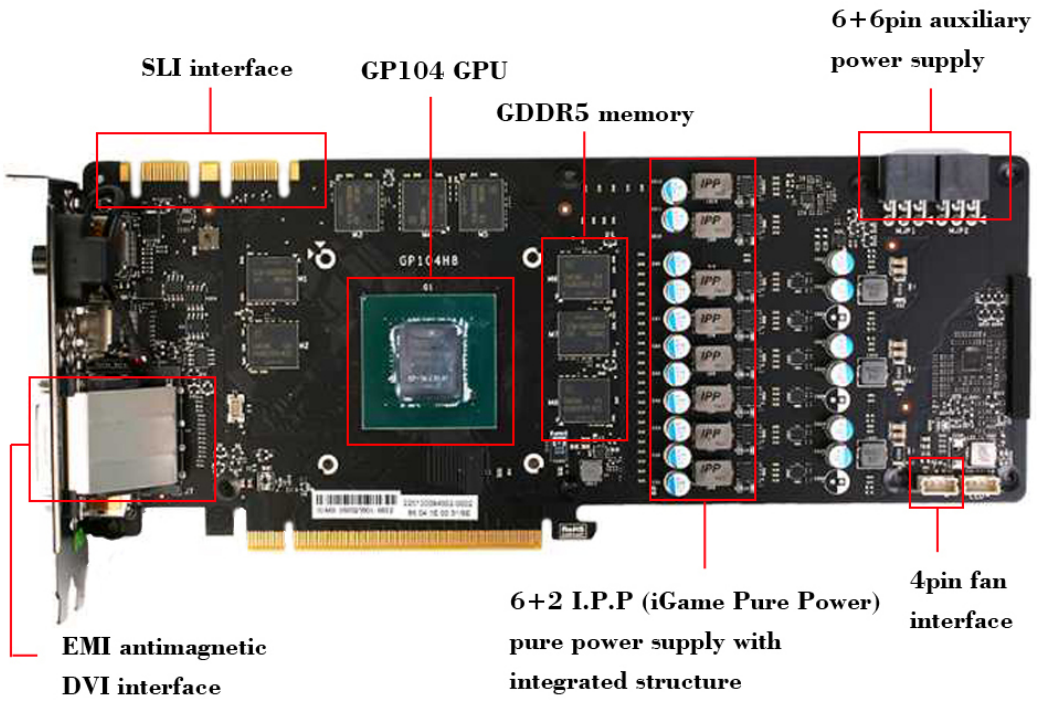
- SISD pour ***Single Instruction, Single Data*** :
  Exemple : Machine séquentielle à la Von Neumann monoprocesseur et monocœur (mais multitâche simulé !)
- SIMD pour ***Single Instruction, Multiple Data*** :
  Exemple : cartes graphiques (OpenCL, CUDA), instructions vectorielles (MMX, SSE)
- MISD pour ***Multiple Instructions, Single Data*** :
  Peu répandu car pas naturel, voir inexistant.
- MIMD pour ***Multiple Instructions, Multiple Data*** :
  Exemple :
  Chaque processeur peut exécuter un programme différent
  Cluster de machines, cloud, machines multiprocesseur/multicœur

Architecture Parallèle : Architecture SIMD

Une application tirant profit de l'architecture SIMD est celle où la même opération doit être effectuée sur un grand nombre de données.

- Simulations physiques
- Calcul de matrice
- Traitement d'images
- Applications graphiques, . . .





Architecture Parallèle : Architecture SIMD

Exemple :

Pour changer la luminosité d'une image, cette architecture est indispensable. Chaque *pixel* d'une image est la composition de trois valeurs RGB (Rouge, vert et bleu) représentant sa couleur. Pour changer la luminosité, ces trois valeurs doivent d'abord être lues à partir de la mémoire. Ensuite, des additions ou des soustractions sont effectuées sur ces trois valeurs et les résultats sont retournés dans la mémoire. Avec l'architecture SIMD, il y a deux avantages :

Architecture Parallèle : Architecture SIMD

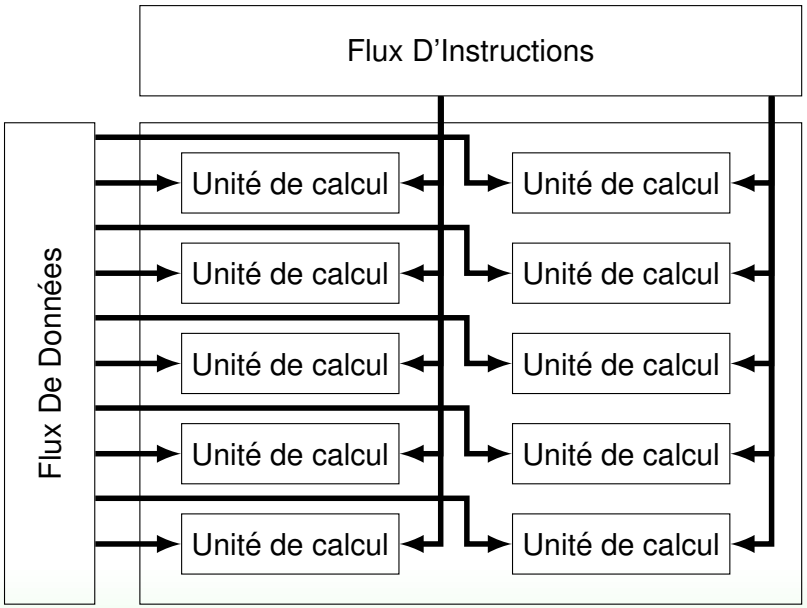
Exemple :

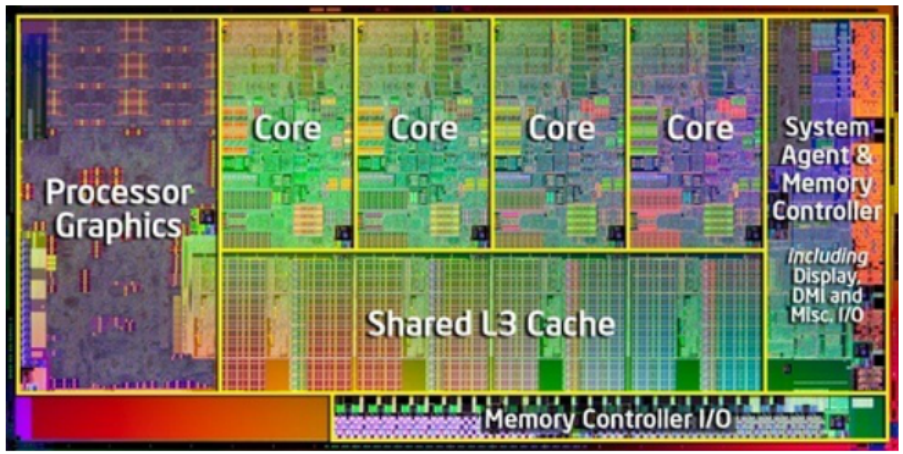
Pour changer la luminosité d'une image, cette architecture est indispensable. Chaque *pixel* d'une image est la composition de trois valeurs RGB (Rouge, vert et bleu) représentant sa couleur. Pour changer la luminosité, ces trois valeurs doivent d'abord être lues à partir de la mémoire. Ensuite, des additions ou des soustractions sont effectuées sur ces trois valeurs et les résultats sont retournés dans la mémoire. Avec l'architecture SIMD, il y a deux avantages :

- Les données (les valeurs RGB des *pixels*) étant les unes à la suite des autres, sont chargées en une seule fois. Autrement dit, plutôt que d'avoir une longue série d'instructions $\ll$ retrouve ce *pixel*, ensuite, retrouve le prochain *pixel*, ... $\gg$, un processeur SIMD fait la même chose en une seule instruction $\ll$ retrouve n *pixels* $\gg$. Avec un processeur traditionnel, cette tâche prendrait beaucoup plus de temps.
- L'autre avantage est que l'instruction de calcul (Addition ou soustraction) est appliquée sur toutes les données chargées en une seule opération.

Architecture Parallèle : Architecture MIMD

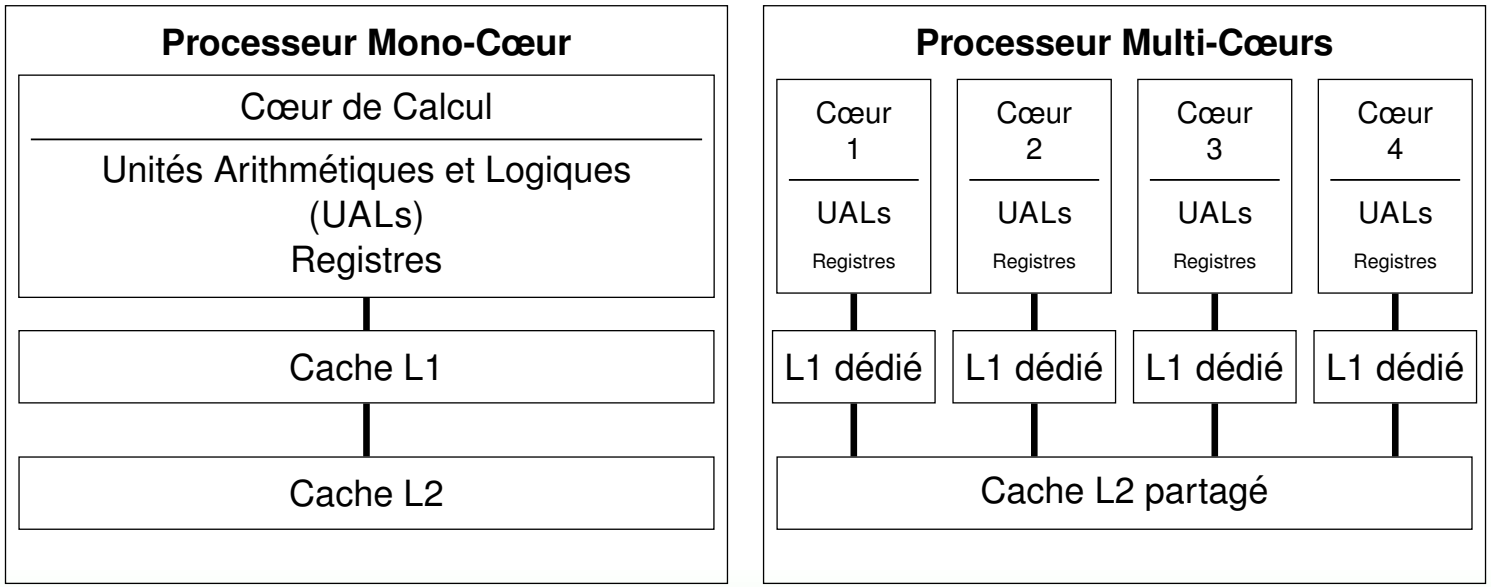
- Chaque processeur peut exécuter un programme différent
- L'exécution peut être :
  - asynchrone ou synchrone
  - déterministe ou indéterministe
- Exemple : Cluster de machines, cloud, machines multiprocesseur/multicœur, . . .
- De nombreuses architectures MIMD incluent aussi des sous-composants SIMD





Architecture Parallèle : Architecture MIMD

- Chaque processeur peut exécuter un programme différent
- L'exécution peut être :
  - asynchrone ou synchrone
  - déterministe ou indéterministe
- Exemple : Cluster de machines, cloud, machines multiprocesseur/multicœur, . . .
- De nombreuses architectures MIMD incluent aussi des sous-composants SIMD



Architectures de la mémoire : mémoire partagée

Définitions :

La **mémoire partagée** est une mémoire accessible par plusieurs programmes afin qu'ils puissent communiquer entre eux et ainsi éviter les copies redondantes. C'est une manière efficace d'échanger des données entre les programmes.

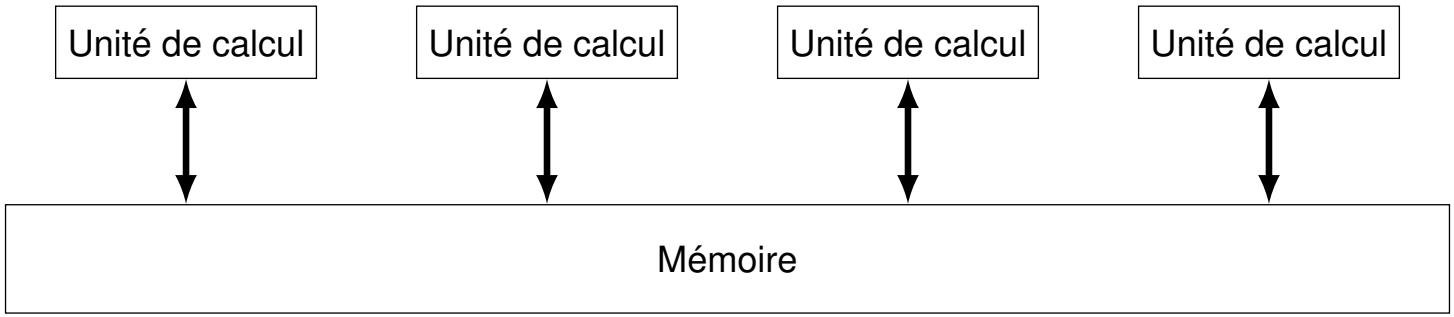
Définitions :

Un **multiprocesseur symétrique** (SMP pour *Symmetric shared memory MultiProcessor*) est une architecture parallèle qui consiste à multiplier les processeurs identiques au sein d'un ordinateur, de manière à augmenter la puissance de calcul, tout en conservant une unique mémoire partagée.

Architectures de la mémoire : UMA

Définitions :

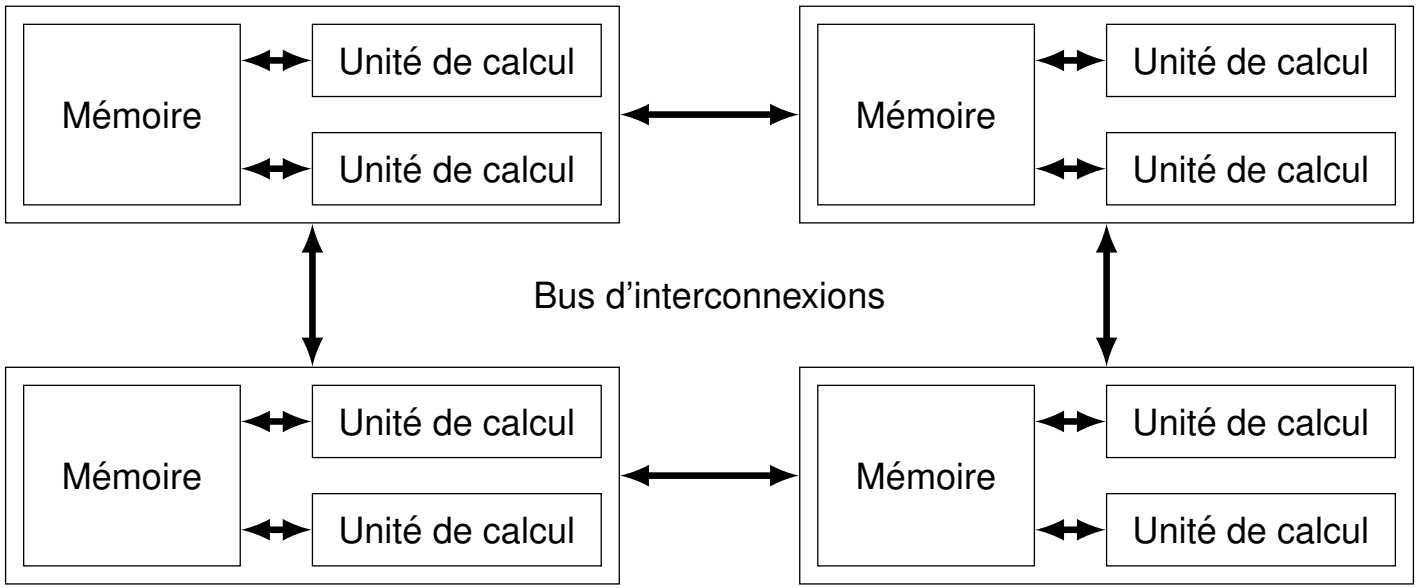
Dans une **architecture de type UMA** (pour *Uniform Memory Access*), toutes les unités de calcul partagent une mémoire physique uniforme. Le temps d'accès à la mémoire est indépendant de l'unité de calcul faisant la requête ou de la puce contenant les données.



Architectures de la mémoire : NUMA

Définitions :

Une **architecture de type NUMA** (pour *Non-Uniform Memory Access*) est un système multiprocesseur dans lequel les zones mémoires sont séparées et placées dans des endroits distincts liées à différents bus. Vis-à-vis de chaque processeur (ou unité de calcul) et suivant la zone mémoire à accéder, les temps d'accès différent.



Architectures de la mémoire : Cohérence du cache

Définitions :

Garder des copies de zones mémoires synchronisées est connu comme le problème de la **Cohérence du cache** (en anglais, *cache coherence*) et les multiprocesseurs utilisant ce système sont appelés **ccNUMA** pour *cache-coherent Non-Uniform Memory Access* et **ccUMA** pour *cache-coherent Uniform Memory Access*.

Architectures de la mémoire : mémoire distribuée

Définitions :

Les systèmes à **mémoire distribuée** possèdent une caractéristique commune : ils nécessitent un réseau de communications pour connecter toutes les unités de calcul entre elles. Dans cette configuration, les processeurs ont leur propre mémoire locale. Les adresses mémoires d'un processeur ne sont pas connues par les autres processeurs. Par conséquent, il n'y a aucun concept d'espace d'adresse global entre toutes les unités de calcul. Lorsqu'un processeur a besoin d'accéder aux données d'un autre processeur, il appartient habituellement au programmeur de définir explicitement quand et comment les données sont communiquées. La synchronisation entre les programmes est également à la responsabilité du programmeur. Le réseau de base entre les machines est le réseau Ethernet, bien que cela n'est pas toujours le cas (InfiniBand, . . .).

Hybridation mémoire partagée/distribuée

Aujourd'hui, la plupart des ordinateurs sont une hybridation des deux architectures de la mémoire : partagée et distribuée. C'est notamment le cas des super ordinateurs ou *cluster* de machines. Le composant utilisant la mémoire partagée peut être un CPU ou un GPU. Les tendances actuelles semblent indiquer que ce type d'architecture de mémoire continuera à prévaloir sur les autres. Néanmoins, il est très compliqué pour un programmeur de coder un programme utilisant correctement cette architecture. Les modèles de programmation parallèle pour ce type d'architecture sont nombreux.

Processeur many-cœurs

Définitions :

Les **Processeurs many-cœurs** sont conçus pour un degré élevé de traitement en parallèle, contenant un grand nombre de cœurs de calcul simples et indépendants. Ils se différencient des multi-cœurs par une architecture mieux adaptée à un très grand nombre de cœurs. Ils ont la particularité d'avoir un pipeline pouvant traiter des opérations vectorielles. C'est donc une hybridation MIMD et SIMD.

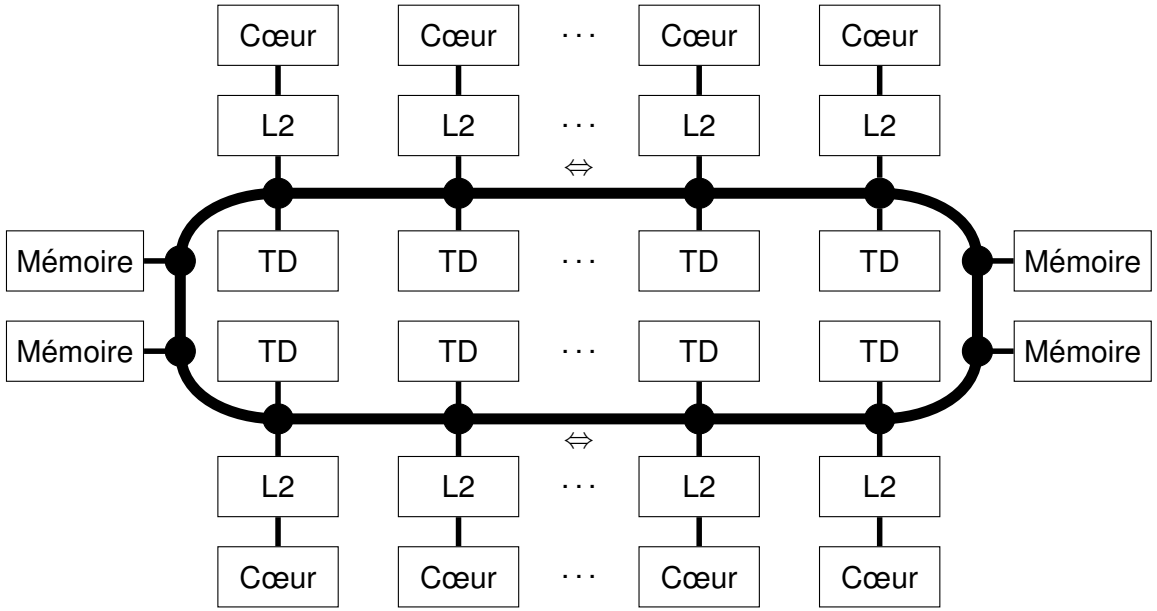


FIGURE : Architecture d'un Xeon Phi d'Intel.

Super-ordinateur

Définition :

Un **superordinateur**, ou **super calculateur**, est un ordinateur conçu pour atteindre les plus hautes performances possibles en ce qui concerne la vitesse de calcul. Ils sont souvent basés sur une architecture de type ccNUMA. Le site `www.top500.org` énumère une liste des 500 super ordinateurs les plus performants. Le « benchmarck Linpack » mesure le taux d'exécution en comptant le nombre d'opérations en virgule flottante par seconde (flops) d'un super ordinateur. Cela est déterminé par l'exécution d'un programme qui résout un système dense d'équations linéaires. La barre des 100 petaflops (1 petaflops $= 10^{15}$ flops) a été franchi pour la première fois en 2016 par un superordinateur de nationalité chinoise nommé le « Sunway TaihuLight ».

Principes de la Parallélisation

Algorithme et Exécution

Définitions :

Un **algorithme parallèle**, par opposition à un traditionnel **algorithme séquentiel**, est un algorithme qui peut être exécuté en différentes parties sur de nombreux appareils de traitement, puis combiné à la fin pour obtenir le résultat correct.
Une seule unité de calcul exécute une seule tâche à la fois, c'est une **exécution séquentielle**. En revanche, n unités de calcul exécute n tâches distinctes simultanément : c'est une **exécution parallèle**.

SpeedUp

Définition :

L'**accélération relative** (resp. **absolue**), en anglais *speedup*, notée $S(n)$, représente le nombre de fois que le programme a été accéléré par son exécution parallèle sur n processeurs par rapport à son exécution séquentielle (resp. au meilleur algorithme séquentiel). Elle est souvent calculée suivant les temps d'exécution (séquentiel T_s et parallèle $T_p(n)$) via la formule suivante :

$$S(n) = \frac{T_s}{T_p(n)}$$

De cette manière, quand le temps séquentiel n'est pas disponible ou est indéterminé, nous ne pouvons pas calculer l'accélération.

Il est aussi possible de calculer l'accélération entre deux architectures afin de mesurer, par exemple, l'amélioration apportée par un pipeline.

SpeedUp

Un processeur sans pipeline exécute deux instructions ADD en 10 cycles d'horloges contre 6 avec un pipeline de type RISC.

- Combien chaque configuration fait elle de cycle par instructions ?

- Quelle est alors le *speedup* apporté par le pipeline ?

SpeedUp

Un processeur sans pipeline exécute deux instructions ADD en 10 cycles d'horloges contre 6 avec un pipeline de type RISC.

- Combien chaque configuration fait elle de cycle par instructions ? Celui sans pipeline fait donc 5 CPI (cycle par instructions) tandis que celui avec un pipeline fait 3 CPI.
- Quelle est alors le *speedup* apporté par le pipeline ?

SpeedUp

Un processeur sans pipeline exécute deux instructions ADD en 10 cycles d'horloges contre 6 avec un pipeline de type RISC.

- Combien chaque configuration fait elle de cycle par instructions ? Celui sans pipeline fait donc 5 CPI (cycle par instructions) tandis que celui avec un pipeline fait 3 CPI.
- Quelle est alors le *speedup* apporté par le pipeline ?

$$S = \frac{5}{3} = 1,666$$

Efficacité

Définition :

L'**efficacité** d'un programme parallèle, notée $E(n)$, est l'accélération $S(n)$ divisée par le nombre n d'unités de calcul. Il représente la qualité de la parallélisation par rapport au nombre de processeurs :

$$E(n) = \frac{S(n)}{n}$$

Accélération (*SpeedUp*)

Définition :

Une **accélération linéaire** est obtenue quand l'algorithme parallèle à une accélération $S(n)$ approximativement égale au nombre n d'unités de calcul. Dans ce cas, nous avons $T_s \approx T_p(n) * n$ et les formules suivantes :

Accélération linéaire :

De la même manière, une **accélération sous-linéaire** (resp. **super-linéaire**) est obtenue quand l'algorithme parallèle à une accélération $S(n)$ strictement inférieure (resp. strictement supérieure) au nombre n d'unités de calcul. Nous avons donc les deux formules suivantes :

Accélération sous-linéaire :

Accélération super-linéaire :

Accélération (*SpeedUp*)

Définition :

Une **accélération linéaire** est obtenue quand l'algorithme parallèle à une accélération $S(n)$ approximativement égale au nombre n d'unités de calcul. Dans ce cas, nous avons $T_s \approx T_p(n) * n$ et les formules suivantes :

Accélération linéaire : $$S(n) = \frac{T_s}{T_p(n)} \approx \frac{T_p(n) * n}{T_p(n)} \approx n$$

$$\text{et } E(n) = \frac{S(n)}{n} \approx \frac{n}{n} \approx 1$$

De la même manière, une **accélération sous-linéaire** (resp. **super-linéaire**) est obtenue quand l'algorithme parallèle à une accélération $S(n)$ strictement inférieure (resp. strictement supérieure) au nombre n d'unités de calcul. Nous avons donc les deux formules suivantes :

Accélération sous-linéaire :

Accélération super-linéaire :

Accélération (*SpeedUp*)

Définition :

Une **accélération linéaire** est obtenue quand l'algorithme parallèle à une accélération $S(n)$ approximativement égale au nombre n d'unités de calcul. Dans ce cas, nous avons $T_s \approx T_p(n) * n$ et les formules suivantes :

Accélération linéaire : $$S(n) = \frac{T_s}{T_p(n)} \approx \frac{T_p(n) * n}{T_p(n)} \approx n$$

$$\text{et } E(n) = \frac{S(n)}{n} \approx \frac{n}{n} \approx 1$$

De la même manière, une **accélération sous-linéaire** (resp. **super-linéaire**) est obtenue quand l'algorithme parallèle à une accélération $S(n)$ strictement inférieure (resp. strictement supérieure) au nombre n d'unités de calcul. Nous avons donc les deux formules suivantes :

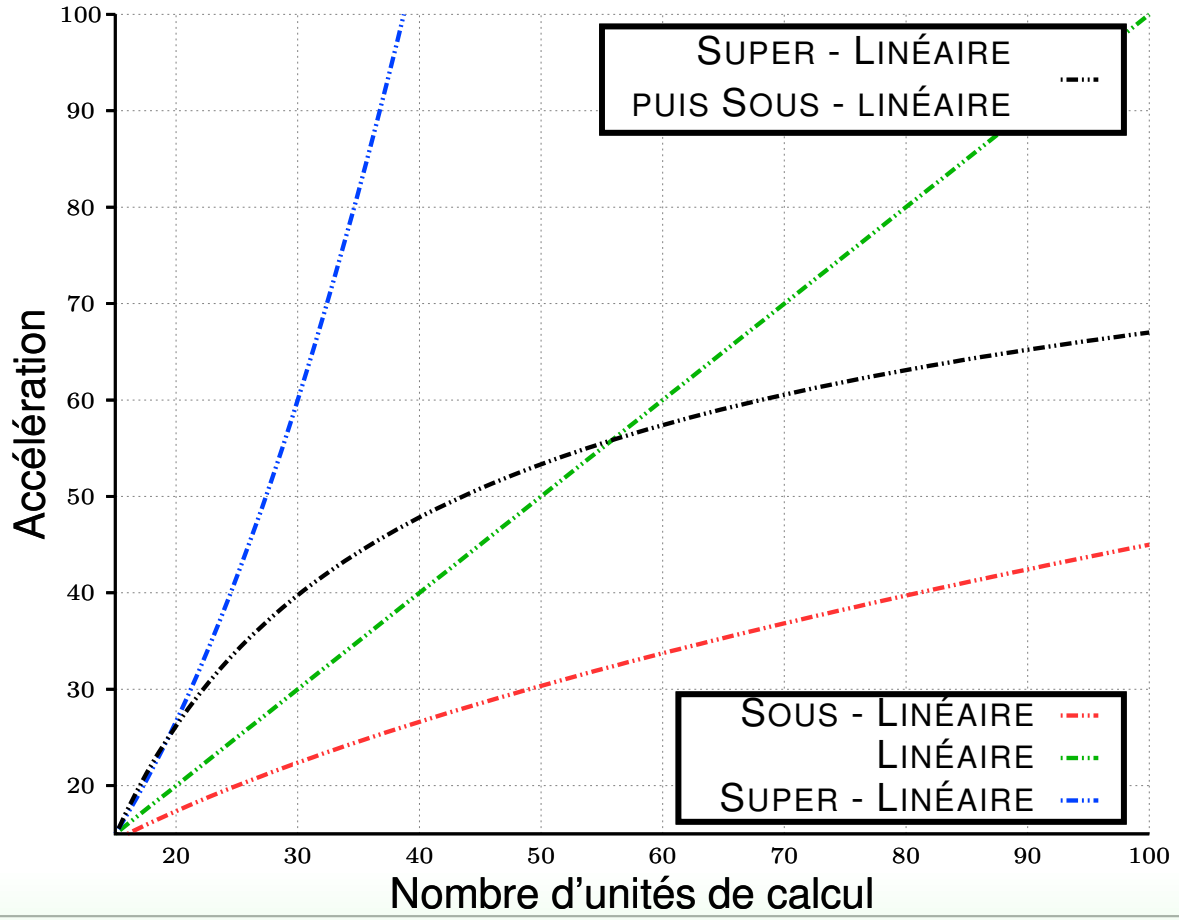
Accélération sous-linéaire : $S(n) < n$ et $E(n) < 1$

Accélération super-linéaire : $S(n) > n$ et $E(n) > 1$

Extensibilité (*Scalability*)

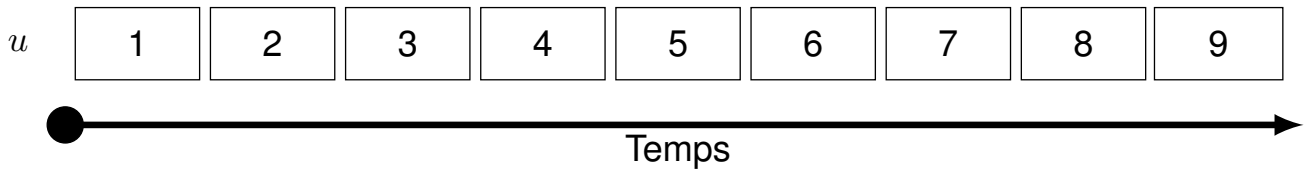
Définition :

L'**extensibilité** (en anglais, *scalability*) d'un algorithme parallèle est sa capacité à maintenir son efficacité quelque soit le nombre d'unités de calcul utilisé.

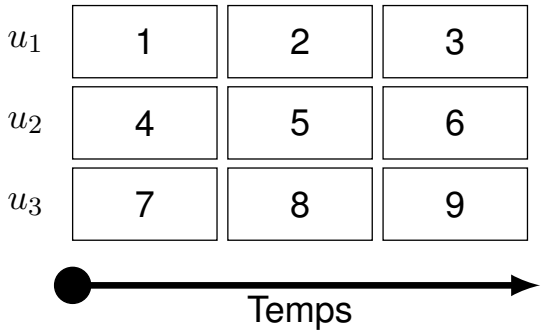


Difficultés du Parallélisme

Algorithme séquentiel



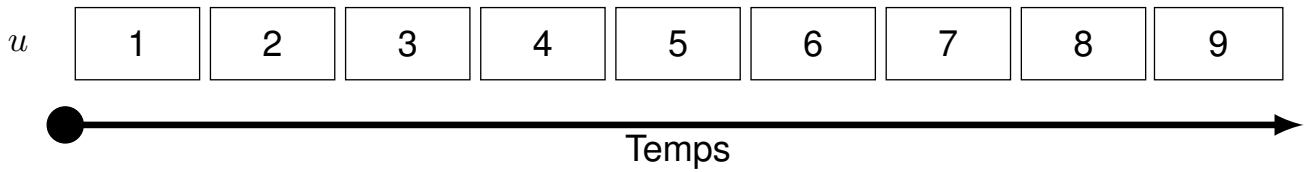
Algorithme parallèle



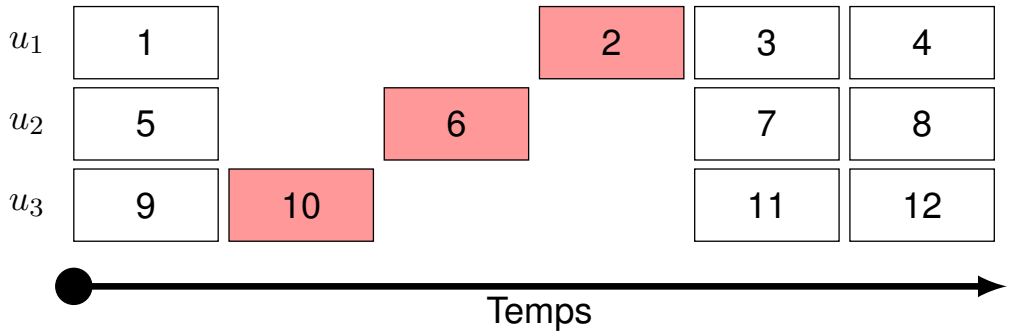
Division du problème en sous-problèmes

Difficultés du Parallélisme

Algorithme séquentiel



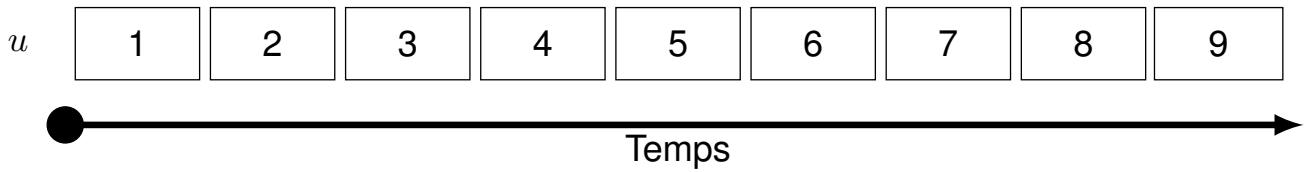
Algorithme parallèle



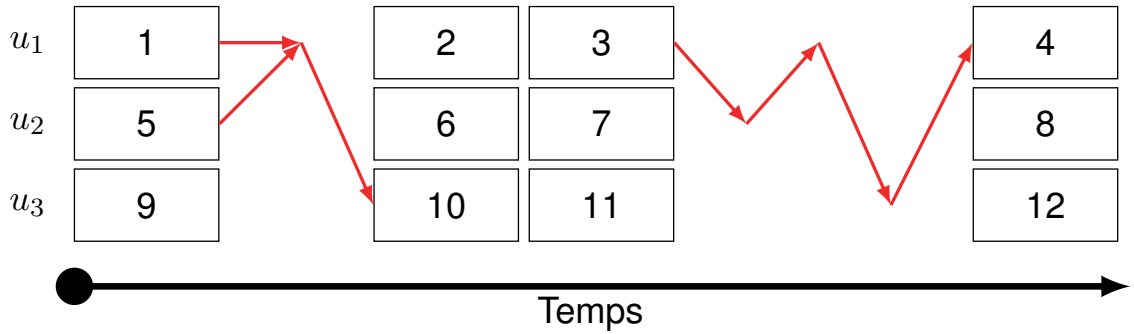
Problème de dépendance

Difficultés du Parallélisme

Algorithme séquentiel



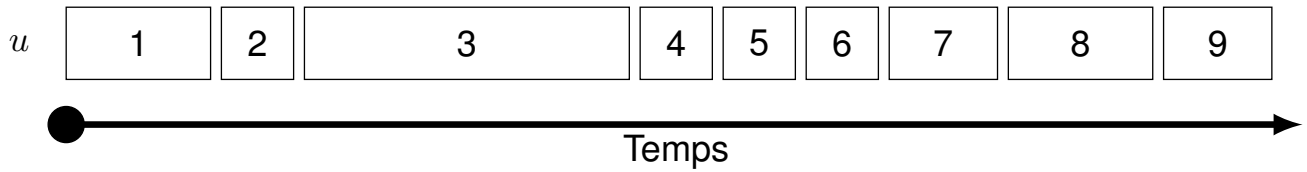
Algorithme parallèle



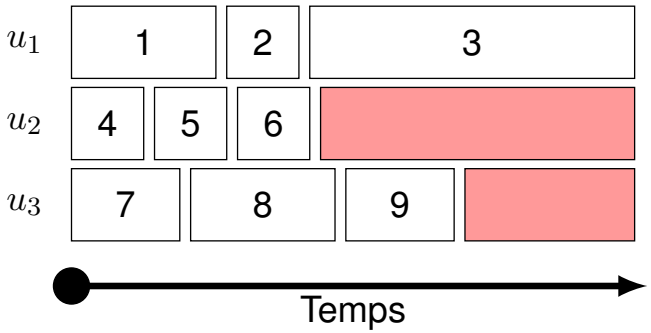
Ralentissement à cause des communications

Difficultés du Parallélisme

Algorithme séquentiel



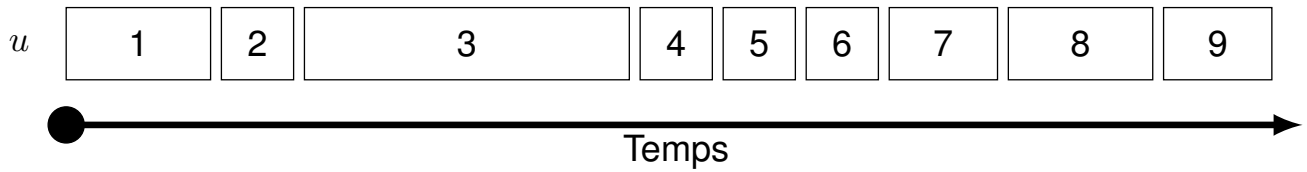
Algorithme parallèle



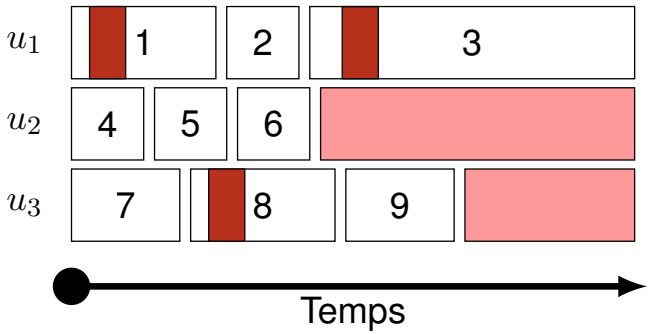
Problème d'équilibrage des charges

Difficultés du Parallélisme

Algorithme séquentiel



Algorithme parallèle



Problème de travaux redondants

Problème de dépendance

Exemple :

Donner un algorithme parallèle permettant de calculer la suite de Fibonacci (0,1,1,2,3,5,8,13,21,...) en utilisant la formule :

$$F(n) = F(n-1) + F(n-2)$$

Problème de dépendance

Exemple :

Donner un algorithme parallèle permettant de calculer la suite de Fibonacci (0,1,1,2,3,5,8,13,21,...) en utilisant la formule :

$$F(n) = F(n-1) + F(n-2)$$

Le calcul des valeurs $F(n)$ utilise à la fois $F(n-1)$ et $F(n-2)$, ils doivent être calculer avant : problème de dépendance.

Problème d'équilibrage des charges

Définition :

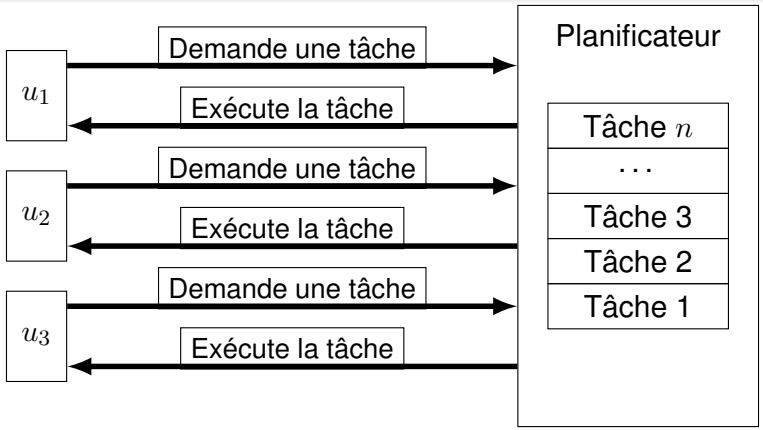
À quelques rares exceptions prés, le problème que l'on doit résoudre, ne peut pas être divisé en morceaux de même complexité. De ce fait, toutes les unités de calcul ne peuvent pas exécuter leurs tâches dans le même délai. Par conséquent, il y a des instants où quelques unités de calcul doivent attendre que d'autres finissent leurs tâches. Cela induit que par moments, des ressources ne sont pas pleinement exploitées, à tel point que le temps nécessaire à l'exécution de l'algorithme augmente. Ce problème, appelé l'**équilibrage de charge** est très fréquent en parallèle.

Quelle sont les solutions a ce problème ?

Problème d'équilibrage des charges

La première idée logique est d'essayer de diviser le problème en plusieurs tâches similaires ayant la même complexité. Néanmoins, il n'est pas souvent possible d'avoir un tel contrôle sur la division d'un problème. Afin d'avoir des tâches les plus équivalentes possibles, une astuce est alors d'entreprendre une division formant de nombreuses petites tâches.

La deuxième idée est de gérer dynamiquement l'attribution des tâches aux unités de calcul. Cela a pour effet de diminuer le déséquilibre des charges, pas parfaitement, mais raisonnablement.



Le langage Go possède une telle attribution dynamique des tâches

Problème parfaitement parallèle

Définition :

Dans le calcul parallèle, un **problème parfaitement parallèle** (en anglais, ***embarrassingly parallel problem***) est celui où il faut peu ou aucun effort pour séparer le problème en un certain nombre de tâches parallèles. Il a donc pour avantage, de ne pas être, ou être très peu concerné par l'équilibrage de charge.

- Calculer l'énergie de chaque conformation (disposition de ses atomes dans l'espace indépendamment des rotations autour des liaisons simples) indépendante d'une molécule. Lorsque vous avez terminé, trouvez la conformation énergétique minimale
- Applications graphiques où chaque *pixel* est indépendant
- Fractale de l'ensemble de Mandelbrot
- Transformation de Fourier discrète, ...

Charge d'exécution et latence

Définitions :

La **charge** C d'exécution d'une tâche est sa quantité de travail en entrée et peut augmenter suivant le nombre de données qui doit être traité par un algorithme.
La **latence** d'une architecture (unité de calcul,...) ou d'un algorithme (séquentiel ou parallèle) est définie via un temps T et une charge C d'exécution d'une tâche par la formule suivante :

$$L = \frac{T}{C} = \frac{1}{V}$$

C'est aussi l'inverse de la vitesse V d'exécution d'une tâche.
L'**accélération en latence** entre deux architectures ou algorithmes est définie par la formule suivante :

$$S_{latence} = \frac{L_1}{L_2} = \frac{T_1 * C_2}{T_2 * C_1}$$

Quand C_1 et C_2 sont égaux, on retrouve la définition de l'accélération relative $S(n) = \frac{T_s}{T_p(n)}$ par rapport à un nombre d'unités de calcul n

Loi d'Amdahl

Soit F_p la fraction du temps d'exécution de la tâche qui doit bénéficier d'une amélioration parallèle à la suite d'une augmentation du nombre d'unités de calcul. La fraction représentant la partie ne bénéficiant pas de cette amélioration, celle séquentielle, est donc $1 - F_p$. Ainsi, le temps total du programme séquentiel T_s (avec une unité de calcul) est :

$$T_s = (1 - F_p) * T_s + F_p * T_s$$

La loi d'Amdahl suppose que le temps de la partie parallèle est amélioré linéairement en fonction du nombre d'unités de calcul n. Le temps total du programme parallèle T_p est donc :

$$T_p = (1 - F_p) * T_s + \frac{F_p}{n} * T_s$$

Loi d'Amdahl

Pour finir, nous appliquons l'accélération en latence $S_{latence}$ avec une charge de travail C fixe :

$$S_{latence} = \frac{T_S * C}{T_p * C} = \frac{T_S}{T_p} = \frac{(1 - F_p) * T_s + F_p * T_s}{(1 - F_p) * T_s + \frac{F_p}{n} * T_s} =$$
$$\frac{(1 - F_p) + F_p}{(1 - F_p) + \frac{F_p}{n}} = \frac{1}{1 - F_p + \frac{F_p}{n}}$$

Définition :

Soit F_p le pourcentage du temps d'exécution de la tâche qui doit bénéficier d'une amélioration parallèle grâce à n unités de calcul. La **loi d'Amdahl** définie l'accélération théorique en latence par :

$$S_{amdahl} = \frac{1}{1 - F_p + \frac{F_p}{n}}$$

Loi de Gustafson

Nous allons procéder de la même manière qu'avec la loi d'Amdahl mais en utilisant la charge plutôt que le temps d'exécution. Nous avons donc :

$$C_s = (1 - F_p) * C_s + F_p * C_s$$

La loi de Gustafson suppose que la charge de la partie parallèle est améliorée linéairement en fonction du nombre d'unités de calcul n. Le charge total du programme parallèle C_p est donc :

$$C_p = (1 - F_p) * C_s + F_p * n * C_s$$

Loi de Gustafson

Pour finir, nous appliquons l'accélération en latence $S_{latence}$ avec un temps d'exécution T fixe :

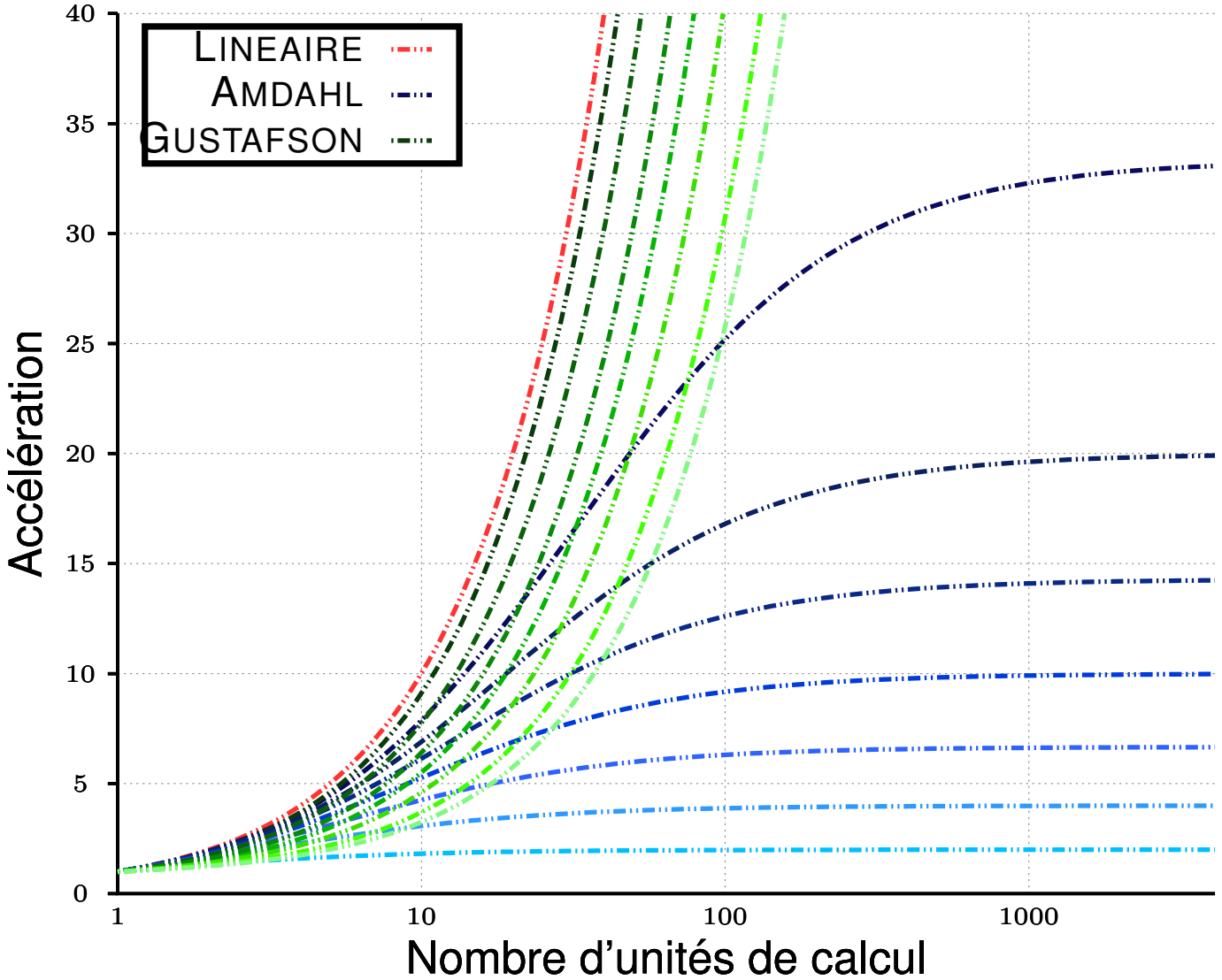
$$S_{latence} = \frac{T * C_p}{T * C_s} = \frac{C_p}{C_s} = \frac{(1 - F_p) * C_s + F_p * n * C_s}{(1 - F_p) * C_s + F_p * C_s} = \frac{(1 - F_p) + F_p * n}{(1 - F_p) + F_p} = (1 - F_p) + F_p * n$$

Définition :

Soit F_p le pourcentage de la charge d'exécution de la tâche qui doit bénéficier d'une amélioration parallèle grâce à n unités de calcul. La **loi de Gustafson** définie l'accélération théorique en latence par :

$$S_{gustafson} = (1 - F_p) + F_p * n$$

Moralité : Amdahl Vs Gustafson



Moralité : Amdahl Vs Gustafson

Comme nous pouvons le constater, la loi d'Amdahl montre que l'accélération théorique est toujours limitée par la partie de la tâche qui ne peut tirer profit de l'amélioration parallèle. Par conséquent, la limite quand le nombre d'unités de calcul n tend vers l'infini est :

$$\lim_{n \to +\infty} S_{amdahl} = \frac{1}{F_p}.$$

Avec une telle limite, cette loi prédit une accélération monotone à partir d'un certain point et n'est donc pas très optimiste. En revanche, la loi de Gustafson est équivalente à l'accélération linéaire quand la partie parallèle est de 100%. Elle montre qu'augmenter la taille du problème peut être bénéfique avec plus d'unités de calcul. Notons qu'aucune des lois existantes ne prédisent une accélération théorique super-linéaire.

Modèles parallèles

Modèle d'exécution Vs de programmation

Définitions :

Le **modèle d'exécution** (ou de fonctionnement) est lié à l'architecture de la machine :

- Caractérise la façon dont sont exécutées les instructions élémentaires
- Diverses classifications différencient ces modes de fonctionnement (Taxonomie de Flynn, . . .)

Le **modèle de programmation** est lié à la traduction de l'algorithme :

- Caractérise la méthode de parallélisation et les algorithmes utilisés

Calcul = Machine + Programme

Modèle de calcul abstrait = Modèle d'exécution + Modèle de programmation

Modèle d'exécution Vs de programmation

La situation dans le monde du calcul parallèle a avancée très rapidement. En faite, au regard des nouvelles architectures sorties ces dernières années, il semble que les machines massivement parallèles ce sont développées si vite que les langages de programmation n'ont pas eu le temps de s'adapter. Une tendance générale a été d'apporter des nouveaux langages pour une architecture donnée. Ces langages contrôlent ainsi certaines fonctionnalités des matériels (architectures) parallèles. Par conséquent, dans la plupart des situations, un modèle de programmation forme un sous-ensemble d'un modèle d'exécution.

Modèle de programmation $\subset$ Modèle d'exécution

Granularité

Définition :

Cette classification se base sur la taille des **grains** : la quantité de calcul impliquée dans un processus en nombre d'instructions. La granularité peut être grossière (entres différents programmes) ou très fine (au niveau des instructions).

Niveau	Granularité	Parallélisme	Division	Communication
Instruction	Fin grain	Le plus haut	Petites tâches	Nombreuses
Boucle	Fin grain	Modéré	Compromis	Compromis
Fonction	Moyen grain	Modéré	Compromis	Compromis
Programme	Gros grain	Bas	Grandes tâches	Peu nombreuses

Parallélisme à Fin Grain Vs à Gros Grain

- Fin Grain (*Fine*) : de nombreuses unités de calcul (des millions de processeurs)
- Gros Grain (*Coarse*), peu d'unités de calcul
- Si la granularité est trop fine : les performances peuvent souffrir d'un surcoûts des communications
- Si la granularité est trop grosse : les performances peuvent être réduites à cause de l'équilibrage des charges

Exécution SPMD Vs MPMD

Définitions :

La technique d'exécution **SPMD** (*Single Program Multiple Data*) consiste à exécuter un seul programme (pouvant néanmoins représenter plusieurs flux d'instructions) sur différentes données tandis que celle **MPMD** (*Multiple Program Multiple Data*) exécute plusieurs programmes distincts.

Ces définitions sont donc associées à la notion de programme où celui-ci possède soit un seul exécutable (SPMD) ou plusieurs (MPMD). Bien que SPMD utilise un seul programme, celui-ci est souvent décomposé via des branchements conditionnels en plusieurs flux d'instructions afin de permettre un traitement parallèle. La technique SPMD peut donc simuler la MPMD. Notons toutefois qu'il y a une différence jouant sur les performances, dans SPMD, il y a un seul grand fichier binaire exécutable, tandis qu'avec MPMD, il y en a plusieurs petits. De ce fait, la technique MPMD est souvent employée avec une architecture distribuée. Par exemple, un logiciel client/serveur possède souvent deux programmes distincts.

Modèles de programmation de la mémoire partagée : IPC

communication inter-processus (IPC) :

Utilise un grand nombre d'appels systèmes. Ainsi, pour faire un programme parallèle avec ce modèle, il faut créer plusieurs processus, où chacun doit faire une cartographie du même espace d'adresses afin de partager des données. Cette dernière manipulation est inefficace car il y a un travail redondant et c'est pourquoi les processus légers (*threads*) ont été créés.

Le standard POSIX apporte une API (bibliothèque de fonctions et procédures (pthread.h) pour utiliser la mémoire partagée (shm_open, sem_open, . . .) et UNIX apporte les fonctions gérant la mémoire et les processus (shmget, shmat, shmctl, . . .).

Modèles de programmation de la mémoire partagée : Processus légers

Processus légers :

Un *thread*, en français, processus léger ou fil d'exécution qui ressemble du point de vue de l'utilisateur à un processus. Toutefois, là où chaque processus possède sa propre mémoire virtuelle, les *threads* d'un même processus se la partagent. Notamment, les *threads* d'un processus partagent leurs codes exécutables et les valeurs de leurs variables tout le temps. Cela a pour effet de rendre plus efficace certaines manipulations qui ne l'étaient pas auparavant comme le démarrage et l'extinction d'un processus.

Il existe de nombreuses librairies utilisant les threads : Intel TBB, Intel Cilk Plus, OpenMP, C++11 Threads, Pthreads, ...

Exclusion mutuelle et sections critiques

Définitions :

Lors de l'utilisation de la mémoire partagée, certaines ressources partagées d'un système ne doivent pas être utilisées en même temps (données, fichiers, imprimantes, ...). Ainsi, pour éviter qu'une données soit modifier par plus d'un *thread* en même temps, un mécanisme (algorithme) d'**exclusion mutuelle** doit être mis en œuvre. Ce mécanisme définit des **sections critiques** qui assurent qu'un seul *thread* à la fois les traverses. Ainsi, les données utilisées par plusieurs *threads* à l'intérieure des sections critiques sont protégées des **situations concurrentes** (*race condition*), situations dans lesquelles le programme serait dans des états imprévus (bogues).

Modèles de programmation de la mémoire distribuée

Définition :

Le modèle de **programmation de la mémoire distribuée** ou du **passage de message** doit prendre en compte une architecture distribuée qui utilise un réseau informatique.

Les sockets :

Il existe deux moyens de communiquer des messages via ce réseau grâce à la notion de socket : les sockets de flux utilisant le protocole TCP (Transmission Control Protocol) et ceux de paquets utilisant le protocole UDP (User Datagram Protocol). Les sockets de flux possèdent deux voies de communications bi-directionnelles et fiables. TCP s'assure que vos données arrivent séquentiellement et sans erreur, en revanche, UDP ne l'assure pas et un message de confirmation de réception doit être envoyé à l'expéditeur.

Modèles de programmation de la mémoire distribuée : MPI

Définition :

Dans le domaine du calcul à haute performance (en anglais, HPC pour *High performance computing*), il existe une spécification officielle nommée MPI pour Message-Passing Interface afin de s'abstraire des notions associées aux sockets pour ne laisser que celles de messages et de données. MPI est une interface de spécification pour les librairies de passage de messages.

MPI :

MPI n'est ni un langage, ni une implémentation, c'est une interface guidant un grand nombre d'implémentations : OpenMPI, MPICH, IntelMPI, ... La plupart des ces implémentations utilisent le protocole TCP pour des raisons de fiabilité.

Modèles de programmation de la mémoire distribuée : MPI

Les librairies qui étendent MPI supportent souvent plusieurs langages. Néanmoins, il n'existe pas d'implémentations MPI qui supportent à la fois le C++ et le Java. Pourtant, des applications Web Client/Serveur pour ces deux langages existent (Architecture REST). Le langage Java ne dispose pas d'une liaison MPI officielle. Toutefois, plusieurs groupes tentent de relier ces deux langages, avec différents degrés de succès et de compatibilités.

Modèles de programmation hybride

Définition :

Un **modèle hybride** combine certains des modèles de programmation décrit précédemment. Ces hybridations permettent de tirer profit de plusieurs bibliothèques afin de gagner en performance. Néanmoins, de telles hybridations apportent des problèmes de compatibilité.

Exemple :

Un exemple commun est la combinaison du modèle de passage de message avec celui des *threads* : OpenMPI/OpenMP, MPICH/pthread, ... Les *threads* réalisent des calculs intensifs en utilisant la mémoire partagée locale des ordinateurs tandis que les communications entre les processus (ou machine) sont faites à travers le réseau en utilisant MPI.

Interblocage

Interblocage

Définition :

Un **interblocage** (en anglais, *deadlock*) est un état dans lequel chaque membre d'un groupe est en train d'attendre indéfiniment qu'un ou plusieurs autres membres fassent une action comme la récupération d'un message ou le blocage/déblocage d'un mutex. Les processus bloqués dans cet état le sont définitivement, il s'agit donc d'une situation catastrophique. Nous pouvons avoir des interblocages provenant d'un modèle de programmation :

- de la mémoire partagée (attente d'un mutex (d'une ressource))
- de la mémoire distribuée (attente d'un message)

Interblocage via la mémoire partagée

Définition :

Ils existent plusieurs manières de déclarer des sections critiques, nous utilisons les notions de mutex (verrou) et de variable condition. Néanmoins, leurs utilisations peuvent entraîner des interblocages (*deadlocks*).

Exemple :

Supposons un système possédant les ressources 1 et 2, ayant les processus A et B en exécution.

- Le processus A réserve la ressource 1
- Le processus B réserve la ressource 2
- Le processus A demande la ressource 2, et tombe en attente
- Le processus B demande la ressource 1, et tombe en attente
- Interblocage !

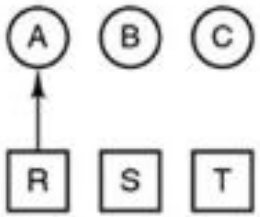
Exemple d'Interblocage

A
Demande R
Demande S
Libération R
Libération S
(a)

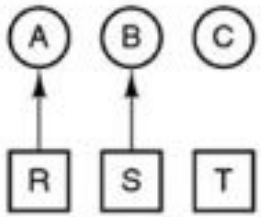
B
Demande S
Demande T
Libération S
Libération T
(b)

C
Demande T
Demande R
Libération T
Libération R
(c)

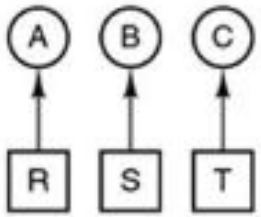
1. A Demande R
2. B Demande S
3. C Demande T
4. A Demande S
5. B Demande T
6. C Demande R
interblocage
(d)



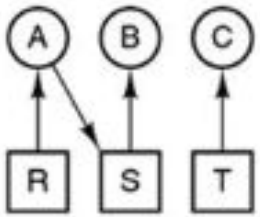
(e)



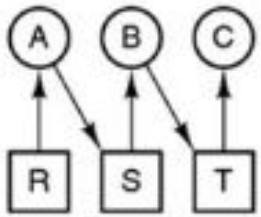
(f)



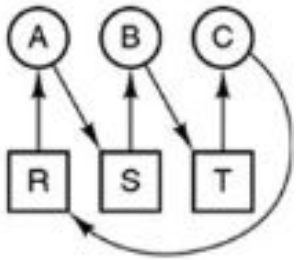
(g)



(h)



(i)

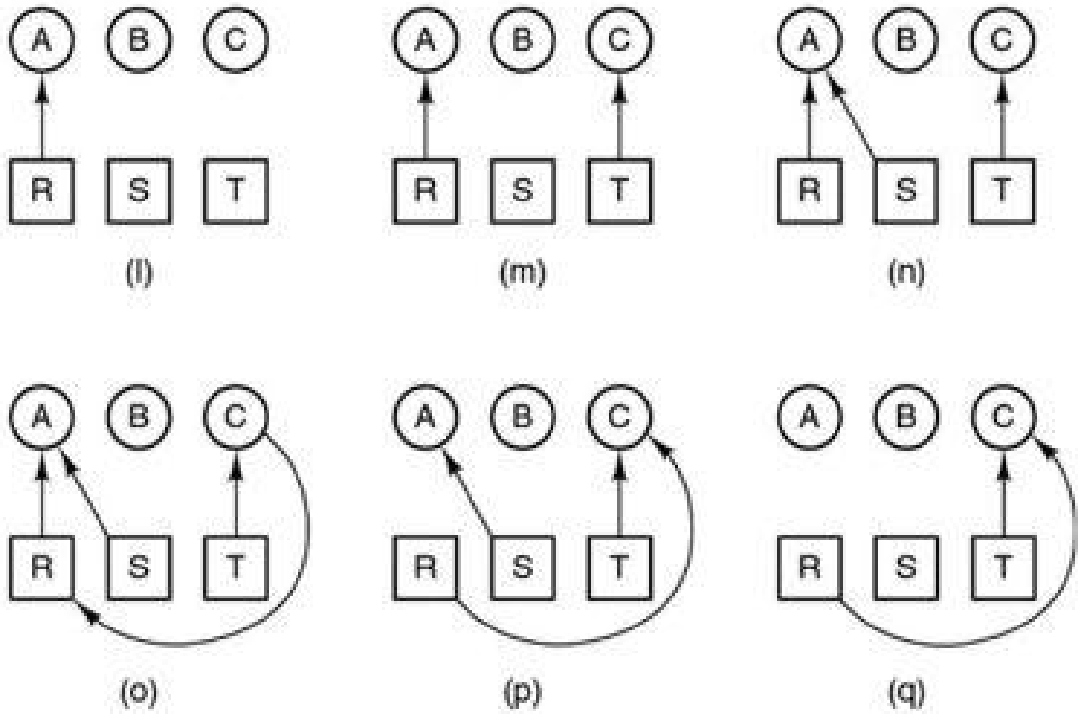


(j)

Exemple d'Interblocage

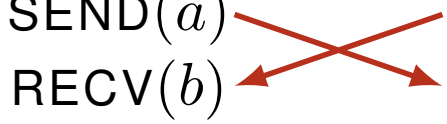
1. A Demande R
2. C Demande T
3. A Demande S
4. C Demande R
5. A Libération R
6. A Libération S

pas d'interblocage



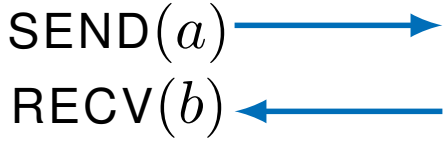
Interblocage via la mémoire distribuée

PROCESSUS 1	PROCESSUS 2
SEND(a)	SEND(b)
RECV(b)	RECV(a)



Interblocage !

PROCESSUS 1	PROCESSUS 2
SEND(a)	RECV(a)
RECV(b)	SEND(b)



Une solution : Sérialiser les communications !

Interblocage via une hybridation

PROCESSUS 1		PROCESSUS 2	
THREAD 1	THREAD 2	THREAD 1	THREAD 2
SEND(a)		SEND(b)	
	RECV(b)		RECV(a)

Thread Safety assure que les threads ne vont jamais avoir d'interblocage quelque soit leur ordre d'exécution

Des travaux récents dans la librairie MPICH permettent un *Thread Safety* efficace !