Introduction à la programmation parallèle

Oguz Kaya

Polytech Paris-Sud

Orsay, France

Apprentissage 4ème année

Novembre Automne 2018

Objectifs

- Introduire les concepts de base de la programmation parallèle
- Présenter les principaux modèles de programmation
- Acquérir une compréhension global des architectures parallèles
- Fournir des conseils pour développer des programmes parallèles efficaces

UNIVERSITÉ PARIS SUD

Plan

Plan

Plan

Plan

UNIVERSITÉ PARIS SUD

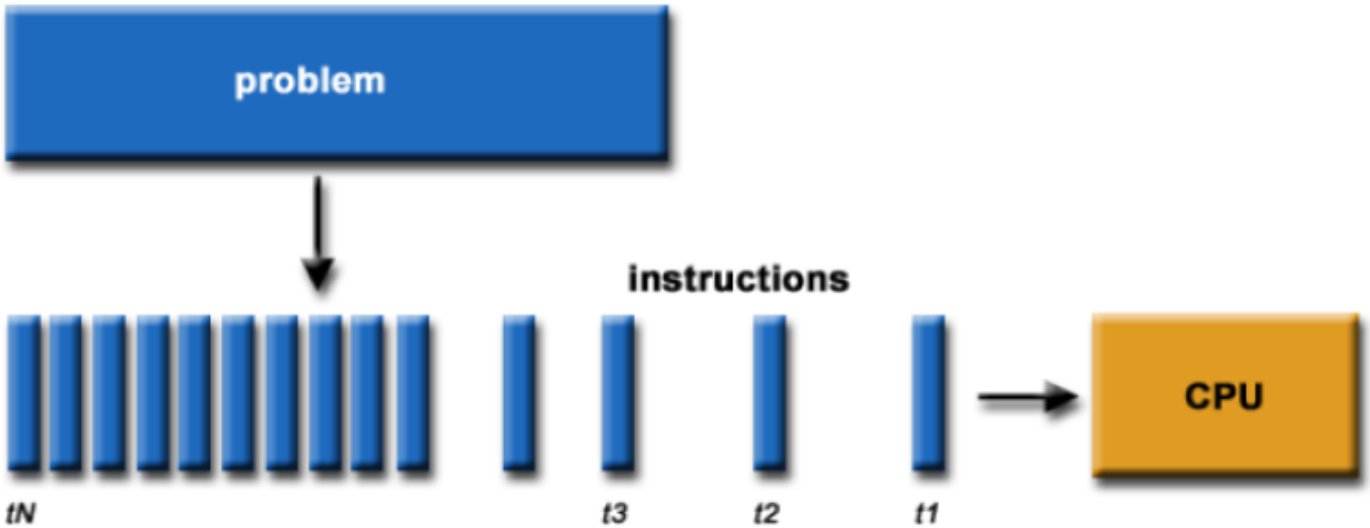
Plan

UNIVERSITÉ PARIS SUD

Programmation séquentielle

Traditionnellement, les logiciels sont basés sur des calculs séquentiels :

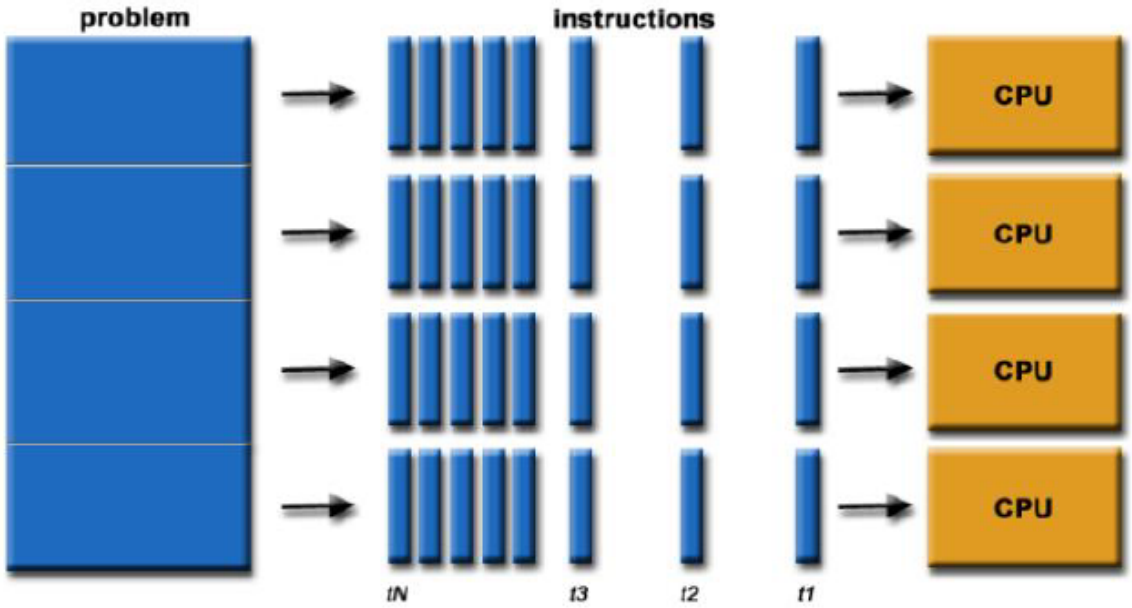
- Un problème est découpé en instructions
- Ces instructions sont exécutées séquentiellement l'une après l'autre
- Elles sont exécutées par un seul processeur
- À un instant donné, une seule instruction est exécutée



Programmation Parallèle

Dans le cas le plus simple, la programmation parallèle est l'utilisation de plusieurs ressources de calcul pour résoudre un problème donné :

- Un problème est découpé en parties qui peuvent être lancées simultanément
- Chaque partie est encore découpée en instructions
- Les instructions de chaque partie sont exécutées en parallèle en utilisant plusieurs processeurs



Acceleration

Le taux auquel le programme s'exécute plus rapidement.

$$\textit{Acceleration} : A(n) = \frac{T(1)}{T(n)}, \text{ où}$$

- T(1) est le temps d'exécution en utilisant 1 processeur et
- T(n) est le temps d'exécution en utilisant n processeurs.

Exemple: Un programme dure 16 seconds avec 1 processeur, 8 seconds avec 2 processeurs, et 4 seconds avec 8 processeurs.

- $A(2) = T(1)/T(2) = 16/8 = 2$
- $A(8) = T(1)/T(8) = 16/4 = 4$

Efficacité

Le ratio d'utilisation des ressources d'une exécution parallèle.

$$\text{Efficacité : } E(n) = \frac{A(n)}{n}$$

Elle indique à quel point la parallélisation d'un code donné est efficace.

Exemple: Un programme dure 16 seconds avec 1 processeur, 8 seconds avec 2 processeurs, et 4 seconds avec 8 processeurs.

- $E(2) = A(2)/2 = 2/2 = 1$, c'est à dire l'efficacité parfaite.
- $E(8) = A(8)/8 = 4/8 = 0.5$, donc on gaspille 50% des ressources.

UNIVERSITÉ PARIS SUD

La loi d'Amdahl

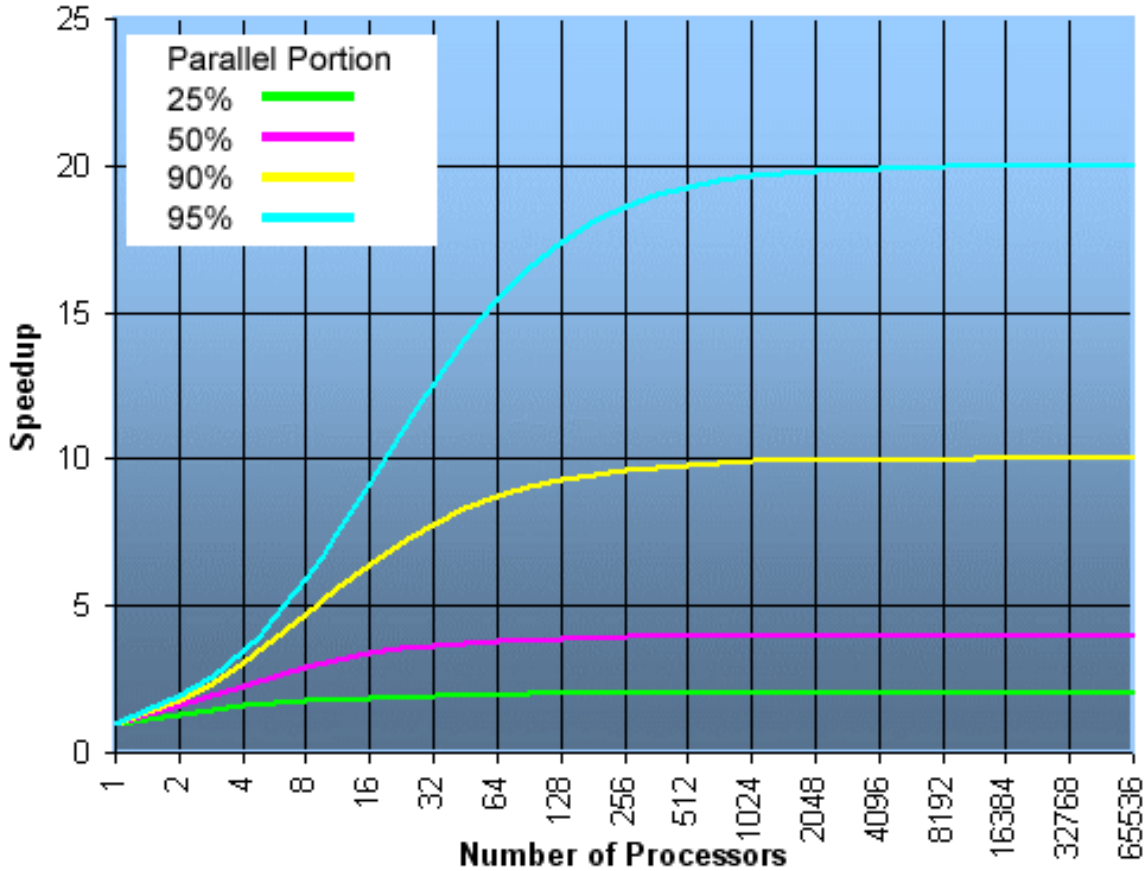
Elle fournit une borne supérieure sur l'accéleration selon la partie séquentielle du program.
Si on considère un programme avec une fraction séquentielle S et une fraction parallèle P (tel que $S + P = 1$), alors

$$A(n) \leqslant \frac{1}{\frac{P}{n} + S} = \frac{1}{\frac{1-S}{n} + S}$$

- Pour $S = 0$, $A(n) \leqslant \frac{1}{1/n+0} = n$
- Pour $S = 1$, $A(n) \leqslant \frac{1}{0/n+1} = 1$

La Loi d'Amdahl

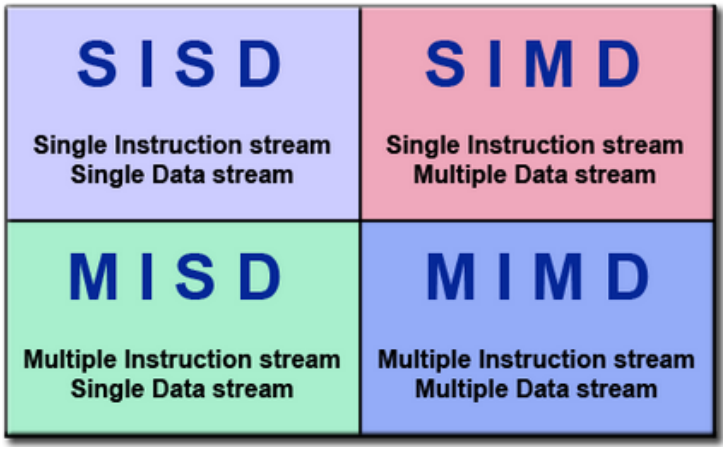
C'est une loi très simpliste mais qui a le mérite de nous indiquer les limites de scalabilité de la parallélisation



Classification de Flynn

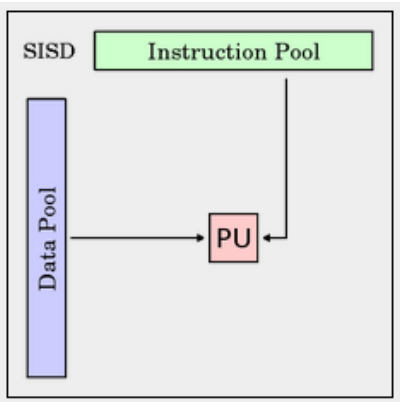
- Il y a différentes manières de classifier les machines parallèles
- Une classification assez répandue, utilisée depuis 1966 est la classification de Flynn
- Cette classification distingue les architectures des machines parallèles suivant les flux d'instructions et de données. Chacun ne peut avoir qu'un seul état : Single (Simple) or Multiple

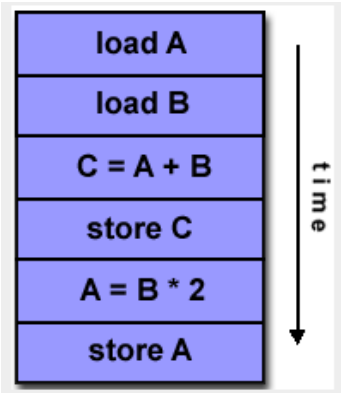
Quatre classifications possibles d'après Flynn:



Single Instruction, Single Data (SISD)

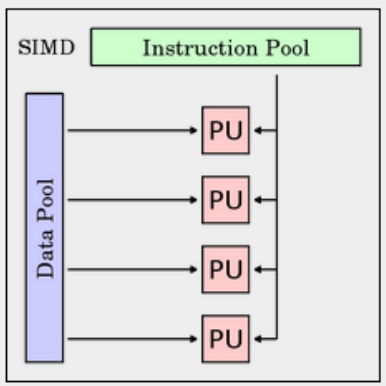
- Une machine séquentielle
- Instruction “Single”: une seule instruction est exécutée par le CPU à chaque cycle d’horloge
- Donnée “Single” : un seul flux de donnée est utilisé comme entrée à chaque cycle d’horloge
- Exécution déterministe
- Les plus anciens des ordinateurs

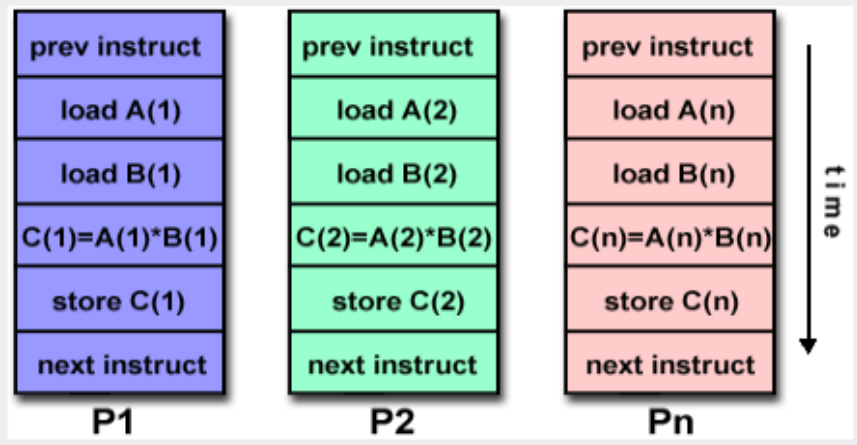




Single Instruction, Multiple Data (SIMD)

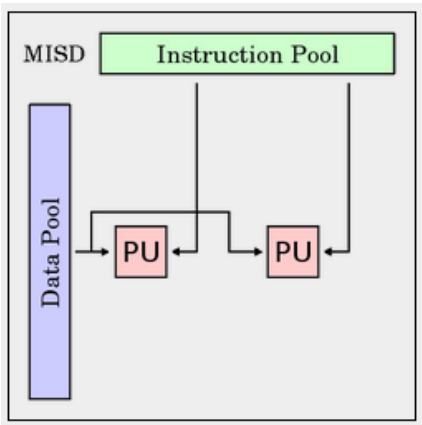
- Un exemple de machine parallèle
- Instruction “Single” : Toutes les unités de calcul exécutent la même instruction à chaque cycle d’horloge
- Données Multiple : Chaque unité de calcul peut opérer sur une donnée différente
- Compatible pour des problème assez réguliers comme le traitement d’images ou l’algebre linéaire

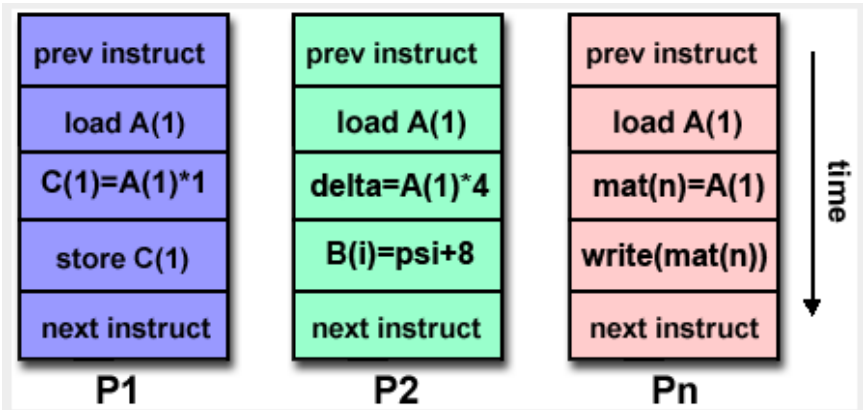




Multiple Instruction, Single Data (MISD)

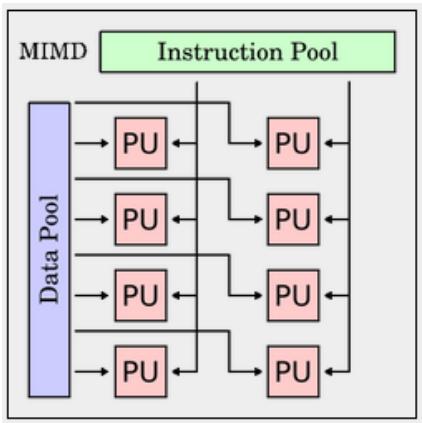
- Un exemple de machine parallèle
- Instruction Multiple : Chaque unité de calcul opère sur des données indépendantes via différents flux d'instructions
- Donnée "Single" : Un seul flux de données alimente plusieurs unités de calcul
- Quelques exemples d'utilisation sont :
  - plusieurs filtres de fréquence opérant sur un seul signal
  - plusieurs algorithme de cryptographies essayant de déchiffrer un seul message codé
  - plusieurs exécutions pour la tolérance aux fautes

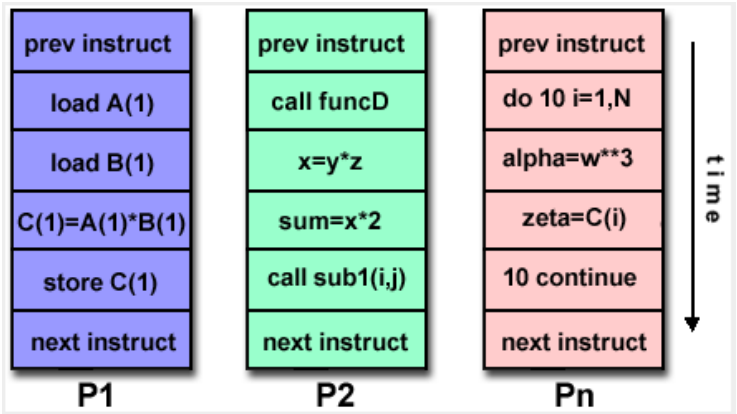




Multiple Instruction, Multiple Data (MIMD)

- Un exemple de machine parallèle
- Instruction Multiple : chaque unité de calcul peut exécuter un flux d'instructions différent
- Données Multiple : chaque processeur peut opérer sur un flux de données différent
- L'exécution peut être synchrone or asynchrone, déterministe or non-déterministe
- La majorité des machines parallèles modernes font partie de cette catégorie
- Plusieurs architectures MIMD incluent aussi des composants SIMD





Plan

UNIVERSITÉ PARIS SUD

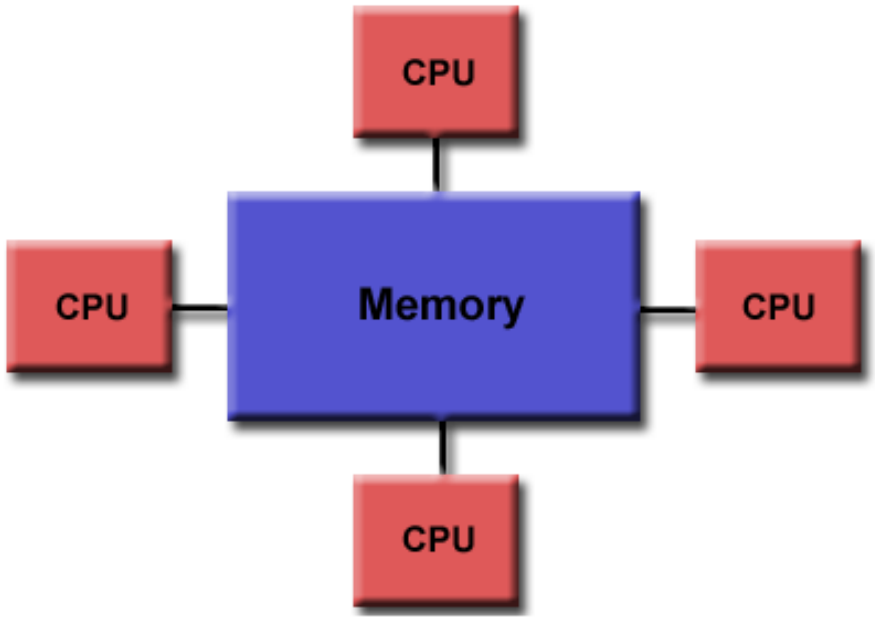
Mémoire partagée : caractéristiques générales

- Tous les processeurs peuvent accèder à la mémoire comme un espace d'adressage global
- Plusieurs processeurs peuvent opérer de façon indépendante mais partagent les mêmes ressources mémoire
- Les modifications par un processeur dans une zone mémoire est visible par tous les autres processeurs

UNIVERSITÉ PARIS SUD

Mémoire partagée : Uniform Memory Access (UMA)

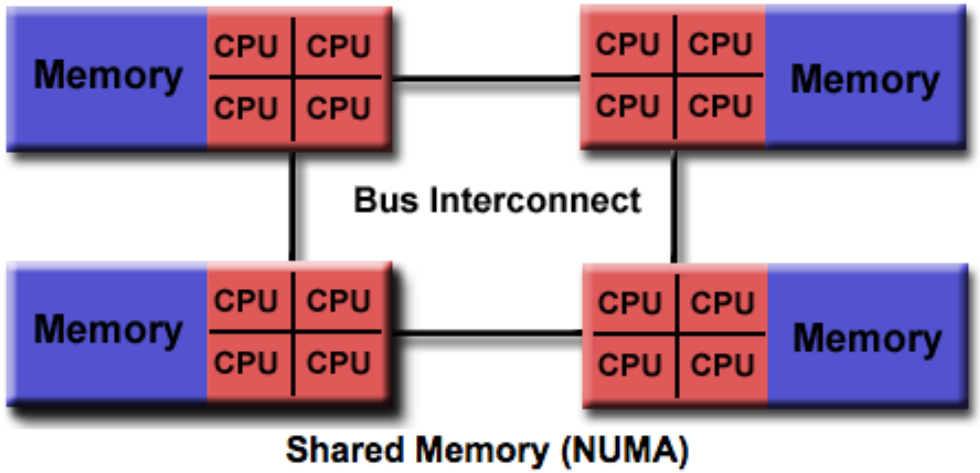
- Présenté par les machines SMP (Symmetric Multiprocessor)
- Processeurs identiques
- Temps d'accès à la mémoire égal pour tous les processeurs



Shared Memory (UMA)

Mémoire partagée : Non-Uniform Memory Access (NUMA)

- Dans la plus part des cas, physiquement construits en liant deux ou plus SMPs
- Un SMP peut accèder directement à la mémoire d'un autre SMP
- Tous les processeurs n'ont pas un temps d'accès égal pour toutes les mémoires
- Les accès mémoire à la mémoire d'un autre SMP sont plus lents
- Si la cohérence du cache est assurée, on parle de cc-NUMA



Mémoire partagée :

1. Avantages :
   - L'espace d'adressage global permet une programmation plus simple de point de vue gestion de la mémoire
   - Le partage des données entre les threads est rapide et uniforme grâce à la proximité de la mémoire du CPU

2. Inconvénients :
   - Le manque de scalabilité entre la mémoire et les CPUs : ajouter plus de CPUs augmentera l'utilisation du bus partagé et pour les systèmes à cache cohérent, celà augmentera l'éffort de gestion de la cohérence entre le cache et la mémoire
   - C'est la responsabilité du programmeur de faire les synchronisations nécessaires pour assurer des accès "corrects" à la mémoire globale

UNIVERSITÉ PARIS SUD

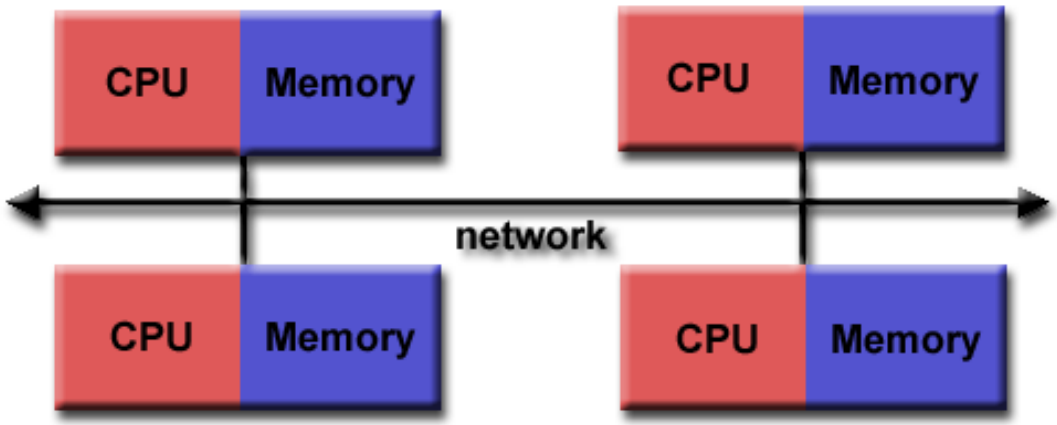
Mémoire distribuée : caractéristiques générales

- Les systèmes à mémoire distribuée nécessitent un réseau pour assurer la connexion entre les processeurs
- Chaque processeur a sa propre mémoire et son propre espace d'adressage
- C'est au programmeur d'indiquer explicitement comment et quand les données doivent être communiquées et quand les synchronisations entre les precesseurs doivent être effectuées

UNIVERSITÉ PARIS SUD

Mémoire distribuée : caractéristiques générales

- Le réseau utilisé pour le transfert des données entre les processeurs est très varié, il peut être aussi simple qu'Ethernet



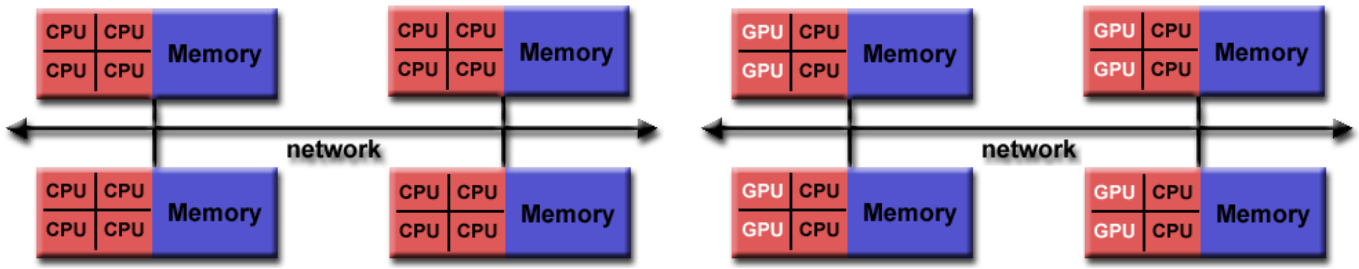
Mémoire distribuée

1. Avantages:
   - La mémoire est scalable avec le nombre de processeurs
   - Chaque processeur accède à sa mémoire rapidement sans ni interference avec les autres processeurs ni de coût additionnel pour maintenir une cohérence globale du cache

2. Inconvénients :
   - Le programmeur est responsable de plusieurs détails associés à la communication des données entre les processeurs
   - Il peut être difficile de distribuer des structures de de données conçues sur une base de mémoire globale sur cette nouvelle organisation mémoire

Mémoire Hybride : caractéristiques générales

- Les machines les plus performantes au monde sont des machines qui utilisent les mémoires partagées et distribuées
- Les composantes à mémoire partagée peuvent être des machines à mémoire partagée ou des GPUs (graphics processing units)
- Les processeurs d'un même neoud de calcul partage le même espace mémoire
- nécessitent des communications pour échanger les données entre les noeuds



Plan

UNIVERSITÉ PARIS SUD

Modèles de programmation parallèle

Les modèles de programmation parallèle existent comme une abstraction au dessus des architectures parallèles

1. Mémoire partagée :
   - **Intrinsics** instructions SIMD (Intel SSE2, ARM NEON), bas niveau (**Plus de détails dans les cours à venir**)
   - Posix Threads bibliothèque
   - **OpenMP** basé sur des directives compilateur à jouter dans un code séquentiel (**Plus de détails dans les cours à venir**)
   - CUDA (**Plus de détails dans les cours à venir ??**)
   - OpenCL
2. Mémoire distribuée :
   - Sockets bibliothèque, bas niveau
   - **MPI** Message Passing Interface le standard pour les architectures à mémoire distribuée, le code parallèle est en général très différent du code séquentiel (**Plus de détails dans les cours à venir**)

UNIVERSITÉ PARIS SUD

Modèles de programmation parallèle

Les modèles de programmation parallèle existent comme une abstraction au dessus des architectures parallèles

1. Mémoire partagée :
   - **Intrinsics** instructions SIMD (Intel SSE2, ARM NEON), bas niveau (**Plus de détails dans les cours à venir**)
   - Posix Threads bibliothèque
   - **OpenMP** basé sur des directives compilateur à jouter dans un code séquentiel (**Plus de détails dans les cours à venir**)
   - CUDA (**Plus de détails dans les cours à venir ??**)
   - OpenCL
2. Mémoire distribuée :
   - Sockets bibliothèque, bas niveau
   - **MPI** Message Passing Interface le standard pour les architectures à mémoire distribuée, le code parallèle est en général très différent du code séquentiel (**Plus de détails dans les cours à venir**)

Quel modèle de programmation utilisé ?

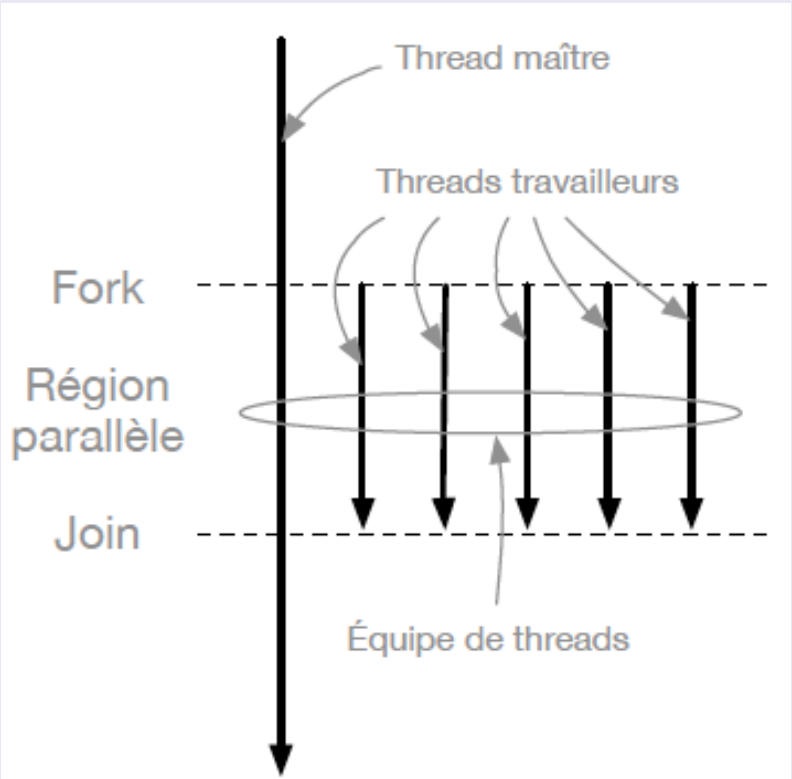
UNIVERSITÉ PARIS SUD

Modèle Thread

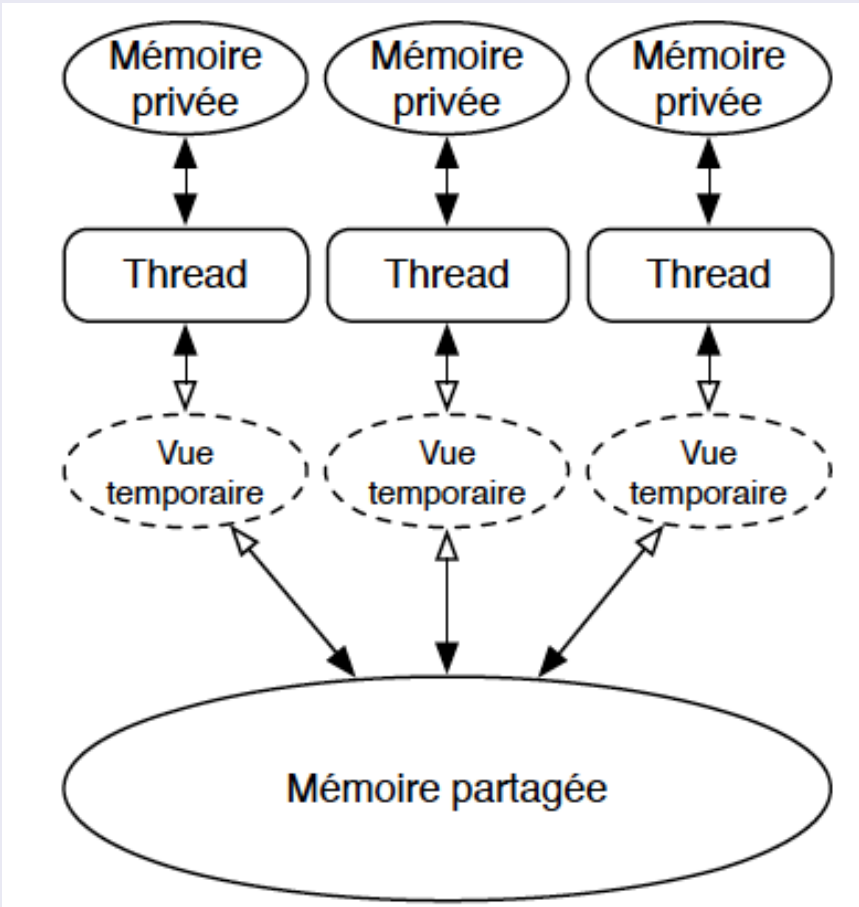
- IEEE POSIX Threads (PThreads)
  - Une API Standard UNIX, existe aussi sous Windows
  - Plus de 60 fonctions: *pthread_create*, *pthread_join*, *pthread_exit*, ...
- OpenMP
  - Une interface plus haut niveau, basée sur
    - des directives compilateur
    - des fonctions bibliothèques
    - un runtime
  - Une orientation vers les application calcul haute performance (HPC)

UNIVERSITÉ PARIS SUD

Modèle d'exécution fork-join



Modèle mémoire



Message Passing Interface

- Spécification et gestion par le forum MPI
  - La bibliothèque fournit un ensemble de primitives de communication : point à point ou collective
  - C/C++ et Fortran
- Un modèle de programmation bas niveau
  - la distribution des données et les communications doivent être faites manuellement
  - Les primitives sont faciles à utiliser mais le développement des programmes parallèles peut être assez difficile
- Les communications
  - Point à point (messages entre deux processeurs)
  - Collective (messages dans des groupes de processeurs)

UNIVERSITÉ PARIS SUD

Produit scalaire : Séquentiel

```c
#include <stdio.h>
#define SIZE 256

int main() {
  double sum, a[SIZE], b[SIZE];

  // Initialization
  sum = 0.;
  for (size_t i = 0; i < SIZE; i++) {
    a[i] = i * 0.5;
    b[i] = i * 2.0;
  }

  // Computation
  for (size_t i = 0; i < SIZE; i++)
    sum = sum + a[i]*b[i];

  printf("sum = %g\n", sum);
  return 0;
}
```

Produit scalaire : instructions SSE

```
#include <immintrin.h>
#include <iostream>
#include <algorithm>
#include <numeric>

int main()
{
  std::size_t const size = 4 * 5;
  std::srand( time( nullptr ) );
  float * array0 = static_cast< float * >( _mm_malloc( size * sizeof( float ), 16 ) );
  float * array1 = static_cast< float * >( _mm_malloc( size * sizeof( float ), 16 ) );
  std::generate_n( array0, size, []() { return std::rand()%10;} );
  std::generate_n( array1, size, []() { return std::rand()%10;} );

  auto r0 = _mm_mul_ps( _mm_load_ps( &array0[ 0 ] ), _mm_load_ps( &array1[ 0 ] ) );

  for( std::size_t i = 4 ; i < size ; i+=4 )
  {
    r0 = _mm_add_ps( r0, _mm_mul_ps( _mm_load_ps( &array0[ i ] ), _mm_load_ps( &array1[ i ] ) ) );

  }

  float tmp[ 4 ] __attribute__((aligned(16)));
  _mm_store_ps( tmp, r0 );
  auto res = std::accumulate( tmp, tmp + 4, 0.0f );

  _mm_free( array0 );
  _mm_free( array1 );

  return 0;
}
```

Produit scalaire : Pthreads

```c
#include <stdio.h>
#include <pthread.h>
#define SIZE 256
#define NUM_THREADS 4
#define CHUNK SIZE/NUM_THREADS

int id[NUM_THREADS];
double sum, a[SIZE], b[SIZE];
pthread_t tid[NUM_THREADS];
pthread_mutex_t mutex_sum;

void* dot(void* id) {
  size_t i;
  int my_first = *(int*)id * CHUNK;
  int my_last = (*(int*)id + 1) * CHUNK;
  double sum_local = 0.;

  // Computation
  for (i = my_first; i < my_last; i++)
    sum_local = sum_local + a[i]*b[i];

  pthread_mutex_lock(&mutex_sum);
  sum = sum + sum_local;
  pthread_mutex_unlock(&mutex_sum);
  return NULL;
}
```

```c
int main() {
  size_t i;

  // Initialization
  sum = 0.;
  for (i = 0; i < SIZE; i++) {
    a[i] = i * 0.5;
    b[i] = i * 2.0;
  }

  pthread_mutex_init(&mutex_sum, NULL);

  for (i = 0; i < NUM_THREADS; i++) {
    id[i] = i;
    pthread_create(&tid[i], NULL, dot,
                   (void*)&id[i]);
  }

  for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);

  pthread_mutex_destroy(&mutex_sum);

  printf("sum = %g\n", sum);
  return 0;
}
```

Produit scalaire : OpenMP

```c
#include <stdio.h>
#define SIZE 256

int main() {
  double sum, a[SIZE], b[SIZE];

  // Initialization
  sum = 0.;
  for (size_t i = 0; i < SIZE; i++) {
    a[i] = i * 0.5;
    b[i] = i * 2.0;
  }

  // Computation
  #pragma omp parallel for reduction(+:sum)
  for (size_t i = 0; i < SIZE; i++) {
    sum = sum + a[i]*b[i];
  }

  printf("sum = %g\n", sum);
  return 0;
}
```

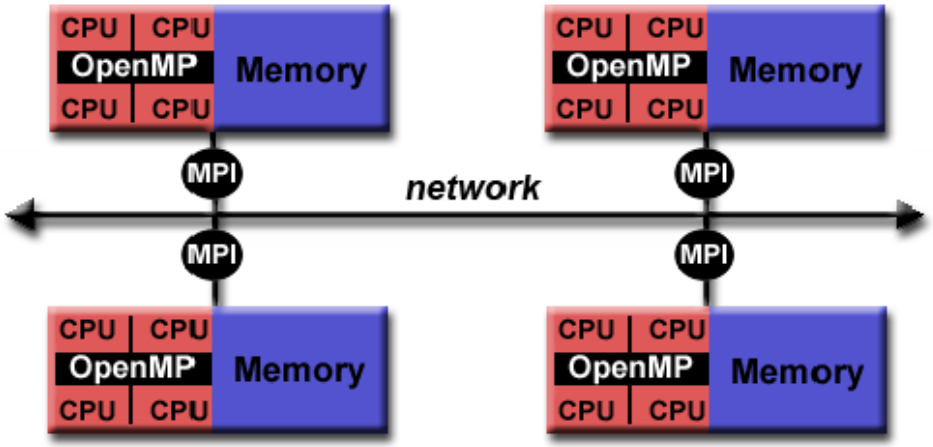
Produit scalaire : MPI

```c
#include <stdio.h>
#include "mpi.h"
#define SIZE 256

int main(int argc, char* argv[]) {
  int numprocs, my_rank, my_first, my_last;
  double sum, sum_local, a[SIZE], b[SIZE];
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  my_first = my_rank * SIZE/numprocs;
  my_last = (my_rank + 1) * SIZE/numprocs;

  // Initialization
  sum_local = 0.;
  for (size_t i = 0; i < SIZE; i++) {
    a[i] = i * 0.5;
    b[i] = i * 2.0;
  }

  // Computation
  for (size_t i = my_first; i < my_last; i++)
    sum_local = sum_local + a[i]*b[i];
  MPI_Allreduce(&sum_local, &sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

  if (my_rank == 0)
    printf("sum = %g\n", sum);
  MPI_Finalize();
  return 0;
}
```

Modèle Hybride

- Plusieurs processus MPI, chacun gèrant un nombre de threads
  - Communication inter-process via envoi de messages (MPI)
  - Communication Intra-process (thread) via la mémoire partagée

- Bien adapté aux architectures hybrides
  - un processus par noeud
  - un thread par coeur



Plan

UNIVERSITÉ PARIS SUD

Un modèle de performance ; le roofline modèle

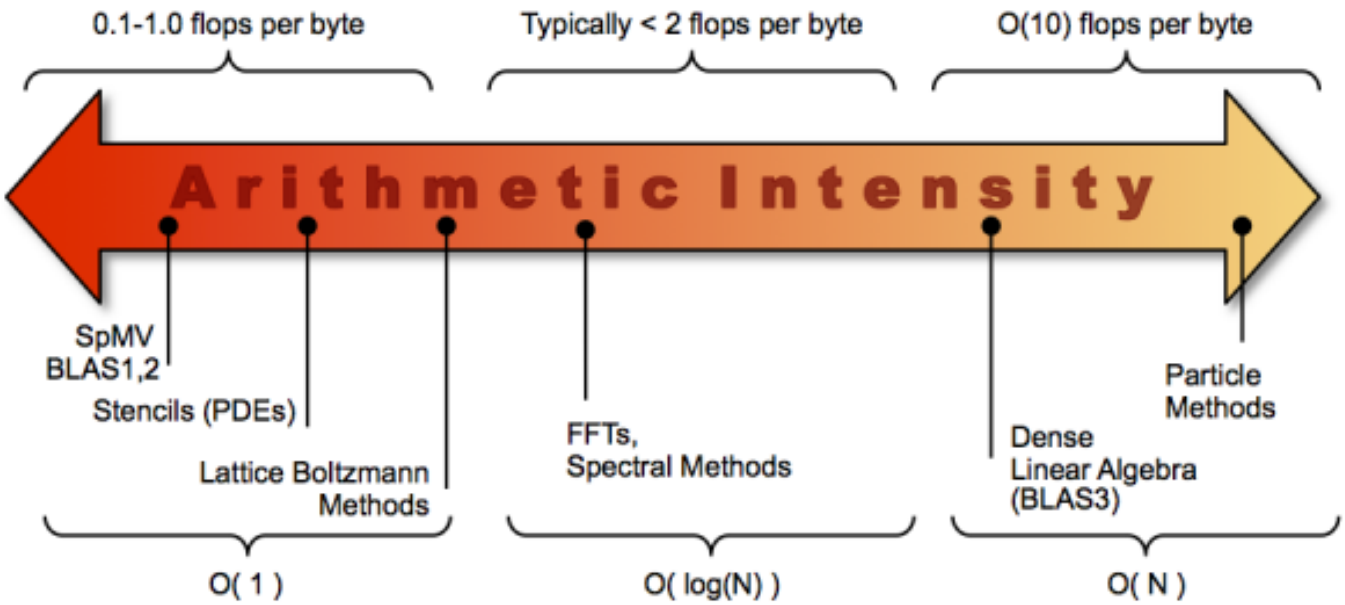
La performance qu'on peut atteindre (flop/s) est bornée par

$$Min \left\{ \begin{array}{l} \text{La performance crête de la machine,} \\ \text{La bande passante maximale} \times \text{l'intensité opérationnelle} \end{array} \right\},$$

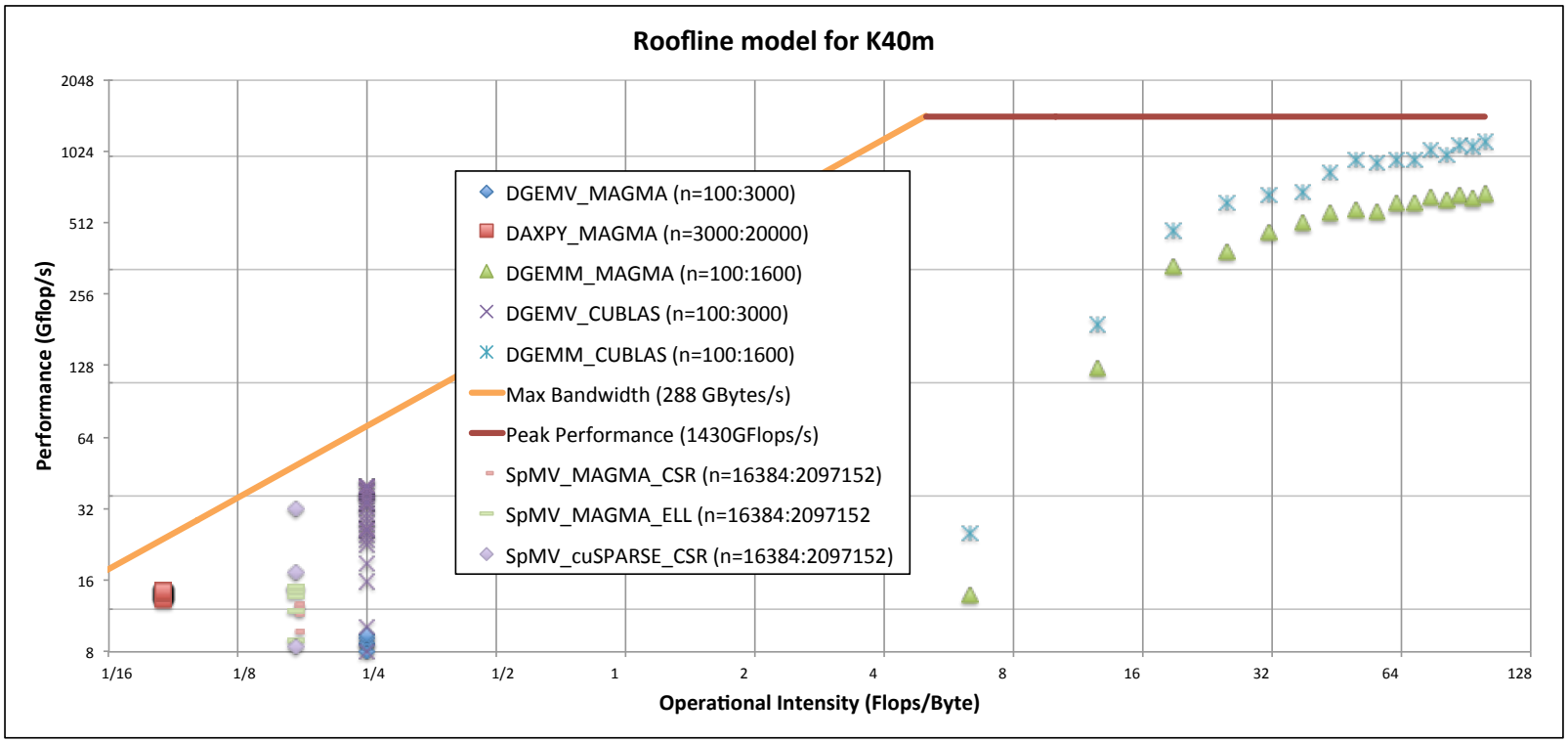
où l'intensité opérationnelle est le nombre d'opération floattantes (flops) effectuée par byte de DRAM transféré (mops)

- $\#flops/\#mops$
- dépend de l'algorithme
- Exemple: $produitscalaire(a, b, n)$
  - On effectue $2n$ mops (charger $a[n]$ et $b[n]$)
  - On effectue $2n-1$ flops (n multiplications et $n-1$ additions)
  - Intensité: $I = (2n-1)/2n \approx 1 = O(1)$.
  - Performance est limité par la bande passante (memory-bound)
- Tant que l'intensité arithmetic augmente, l'algorithme devient de plus en plus limité par la capacité de calcul (compute-bound)

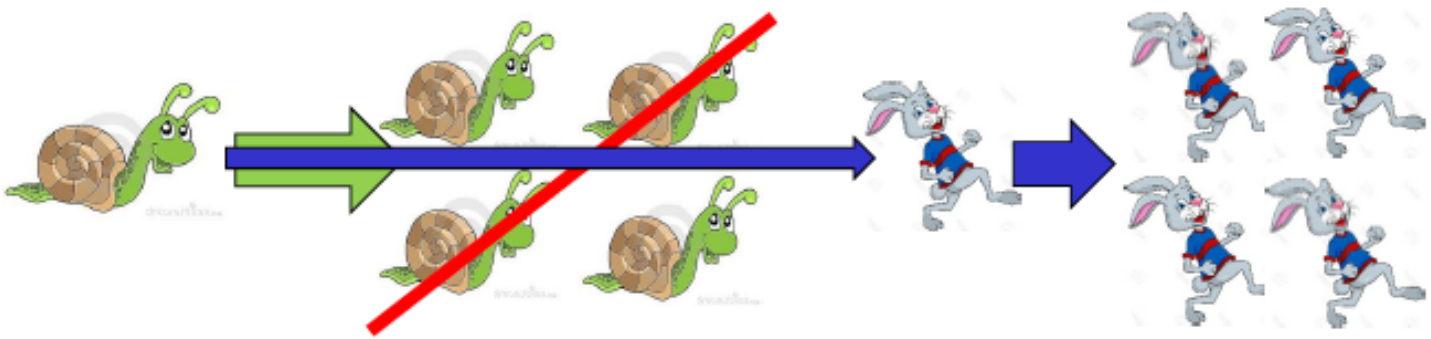
le roofline modèle : l'intensité opérationnelle



Modèle de performance pour NVIDIA Tesla K40



Où commencer ? : Optimisation des noyaux



Performance au niveau du coeur

- Réduire les défauts de cache : blocking, tilling, loop ordering, . . .
- Vectorisation (unités SSE/AVX)

Quoi faire après ?

- Identifier les goulot d'étranglements du programme :
  - savoir les parties qui consomment le plus de temps d'exécution
  - les outils d'analyse de performance peuvent aider ici (profilers . . .)
  - Se concentrer sur la parallélisation des goulot d'étranglements
- Identifier les goulot d'étranglements du programme :
  - Est ce qu'il y a des parties très lentes
  - Pourquoi ?
- Re-structurer le programme ou utiliser un autre algorithme pour réduire les parties qui sont très lentes
- Utiliser l'existant : logiciels et bibliothèques parallèles optimisés (IBM's ESSL, Intel's MKL, AMD's AMCL, LAPACK, . . .)
- Développer de nouveaux algorithmes

UNIVERSITÉ PARIS SUD

Qu'est ce qui doit être considéré ? : quelques éléments

- La distribution des données : 1D, 2D, block, block cyclic, tiles . . .
- La granularité
- Les communications
- Les synchronisations
- Le recouvrement des calculs et des communications
- L'équilibrage de charge entre les threads et/ou les processeurs
- . . .