

Introduction au Calcul Parallèle et Haute Performance

Oguz Kaya

Maître de Conférences

Université Paris-Saclay et l'Équipe ParSys du LRI, Orsay, France

Outline

- 1 Introduction
- 2 Calcul parallèle, pourquoi?
- 3 Concepts du calcul parallèle
- 4 Types de parallélisme
- 5 Architecture Parallèle
- 6 Programmation Parallèle
- 7 Autour du développement de programmes parallèles

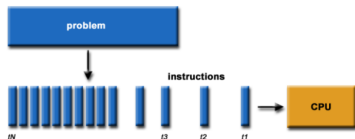
- 1 Introduction
- 2 Calcul parallèle, pourquoi?
- 3 Concepts du calcul parallèle
- 4 Types de parallélisme
- 5 Architecture Parallèle
- 6 Programmation Parallèle
- 7 Autour du développement de programmes parallèles

Objectifs

- Faire connaissance avec le calcul parallèle
- Découvrir les applications qui ont besoin de la puissance de calcul
- Explorer l'architecture moderne d'un ordinateur parallèle
- Présenter les principaux modèles de programmation parallèle
- Introduire les concepts de base et les exemples
- Fournir quelques conseils pour développer des programmes parallèles efficaces

Programmation séquentielle

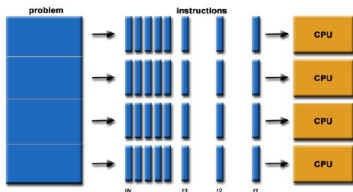
Traditionnellement, les logiciels sont basés sur le calcul **séquentiel**:



- Un problème est découpé en instructions.
- Ces instructions sont exécutées **séquentiellement** les unes après les autres.
- Elles sont exécutées par **un seul processeur**.
- À un instant donné, une seule instruction est exécutée.
- La performance est déterminée principalement par **la fréquence** (Hz) du processeur.

Programmation parallèle

La **programmation parallèle** permet l'utilisation de plusieurs ressources de calcul pour résoudre un problème donné:

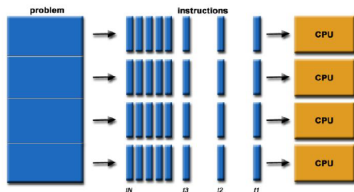


- Un problème est découpé en parties qui peuvent être lancées simultanément.
- Chaque partie est encore découpée en instructions.
- Les instructions de chaque partie sont exécutées **en parallèle** en utilisant plusieurs processeurs.
- La performance est déterminée par:
 - La fréquence du processeur
 - Le nombre de processeurs
 - Le degré de parallélisation du problème

- 1 Introduction
- 2 Calcul parallèle, pourquoi?
- 3 Concepts du calcul parallèle
- 4 Types de parallélisme
- 5 Architecture Parallèle
- 6 Programmation Parallèle
- 7 Autour du développement de programmes parallèles

Applications du calcul parallèle

De nombreuses applications gourmandes en temps de calcul dans de diverses domaines:



- Calcul scientifique: Simulations en physique, chimie, biologie, ...
- Traitement des reseaux neurones
- Graphiques (rendering, jeux vidéo, etc.)
- Systèmes d'exploitation (Linux, Android, etc.)
- et bien d'autres...

Frèquence maximale d'un CPU

Qu'est-ce la fréquence maximale d'un CPU en 2020?



Intel Core i9-10900K, Comet Lake

- 5.3 GHz
- La performance crête? 5.3 Gflops/s?
- Non! ≈ 1.7 Tflops/s (1700 Gflops/s) Comment?
- Grace au parallisme deadans (10 cœurs, chacun avec 2 unités véctorielles AVX256)

- 1 Introduction
- 2 Calcul parallèle, pourquoi?
- 3 Concepts du calcul parallèle**
- 4 Types de parallélisme
- 5 Architecture Parallèle
- 6 Programmation Parallèle
- 7 Autour du développement de programmes parallèles

Moore's Law – The number of transistors on integrated circuit chips (1971-2018)

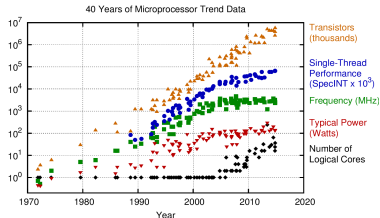
Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products –



- ## Moore's Law

Dennard Scaling

Si la dimension de gravure est réduite par -30% (0.7x) dans chaque génération, ceci donne



Dennard Scaling

- Une baisse de zone du circuit de 50%
- Une baisse de latence de 30%
- En conséquence, une hausse de fréquence de 40% ($10/7 \approx 1.4$).
 - N'est plus possible car effets quantiques!
 - Recourir à soit multi-cœurs, soit gros-cœurs.
- $\text{energie} \sim \text{frequence}^2$
 - Utiliser plus de cœurs à une fréquence basse
→ plus de performances dans la même fenêtre énergétique

Accélération et efficacité

À quel point un programme parallèle s'exécute plus rapidement? À quel point les ressources de calcul sont bien exploitées?

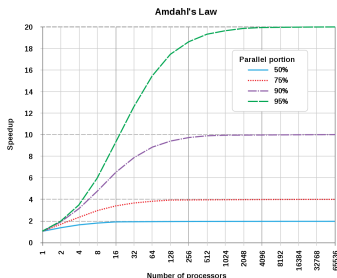


Cray-1 Supercomputer (1975) with peak performance of $\approx 160\text{Mflops/s}$

- Le temps d'exécution séquentiel: $T(1)$
- Le temps d'exécution sur N processeurs: $T(N)$
- **Accélération**: $S(N) = T(1)/T(N)$
- **Efficacité**: $E(N) = S(N)/N$

La loi d'Amdahl

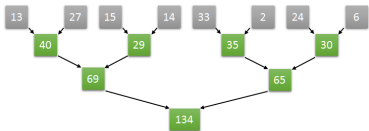
À quel point un programme pourrait-il s'exécuter plus rapidement? Quelle est la limite?



La loi d'Amdahl

- Introduite en 1967 par Gene Amdahl
- Soit s et p la fraction **séquentielle** et **parallélisable** d'un programme ($s + p = 1.0$).
- Accélération atteignable est bornée par
$$A(N) = \frac{T(1)}{T(N)} \leq \frac{T(1)}{pT(1)/N + sT(1)} = \frac{1}{p/N + s}$$
- **Strong scaling:** Scalabilité parallèle pour un problème de taille fixe
- Désespoir pour le calcul parallèle?
 - Typiquement S diminue avec la taille du problème.

Exemple: Somme d'un tableau



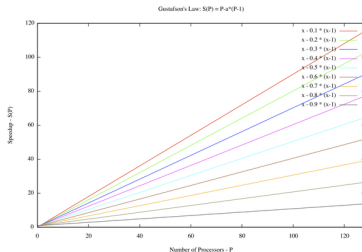
Somme en parallèle

$$\sum_{i=0}^{N-1} A[i]$$

- $N - 1$ additions
- Pour $N = 8$, 7 additions dont 3 séquentielles, $S \leq 7/3 = 2.34$
- Pour $N = 16$, 15 additions dont 4 séquentielles, $S \leq 15/4 = 3.75$
- Dans le cas général, $N - 1$ additions dont $\log N$ séquentielle, $S \leq (N - 1)/\log N$

La loi de Gustafson

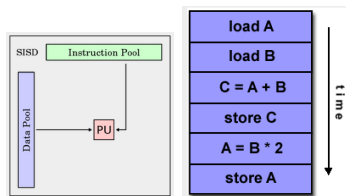
Combien d'accélération pourrait-on obtenir si la taille du problème augmente avec le nombre de processeurs?



La loi de Gustafson

- $S(N) = N + (1 - N)s$
- Amdahl: Il pourrait être difficile accélérer un calcul avec plus de processeurs
- Gustafson: Il est possible d'effectuer plus de calcul aussi rapidement avec plus de processeurs
- Lequel est plus pertinent en pratique?
- **Weak scaling:** Scalabilité parallèle par rapport à la taille du problème augmentante.

Single Instruction, Single Data (SISD)

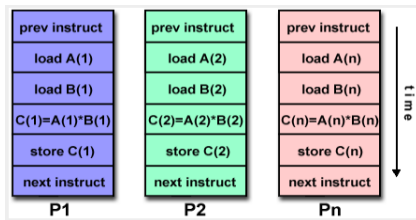
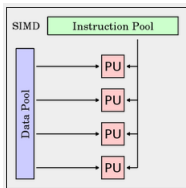


Processeur SISD

- Une machine complètement séquentielle
- Instruction “single”: Une seule instruction est exécutée à chaque cycle d'horloge
- Donnée “single”: Un seul flux de donnée est utilisée
- Exécution déterministe
- Les plus anciens des ordinateurs (i.e., cours d'architecture)

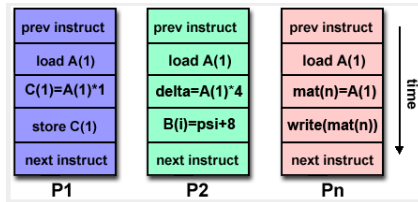
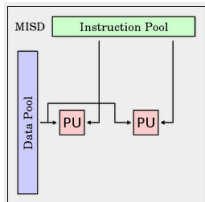
Single Instruction, Multiple Data (SIMD)

- 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 les problème assez réguliers comme les traitement d'images



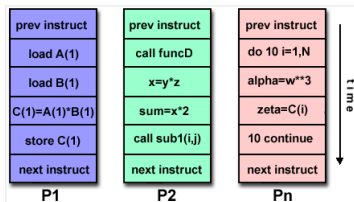
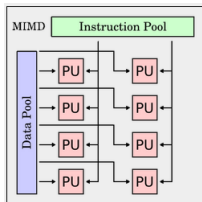
Multiple Instruction, Single Data (MISD)

- 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 algorithmes de cryptographies essayant de déchiffrer un seul message codé coded message



Multiple Instruction, Multiple Data (MIMD)

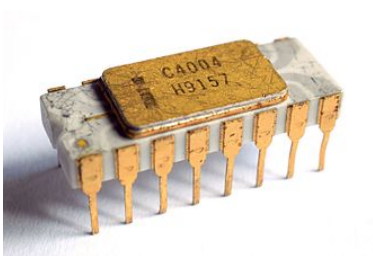
- Un exemple de machine parallèle
- Instruction Multiple : chaque unité de calcul peut exécuter un flux d'instructions différent
- Données Multiple : chaque processeur peut opérer sur un flux de données différent
- L'exécution peut être synchrone or asynchrone, déterministe or non-déterministe
- La majorité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



- 1 Introduction
- 2 Calcul parallèle, pourquoi?
- 3 Concepts du calcul parallèle
- 4 Types de parallélisme**
- 5 Architecture Parallèle
- 6 Programmation Parallèle
- 7 Autour du développement de programmes parallèles

Parallélisme au niveau du bit

Il s'agit de la taille de mot (word-size) du processeur.



- Pertinent pour les années 1970..1986.
- La taille de mot: 4-bit \rightarrow 8-bit \rightarrow 16-bit \rightarrow 32-bit
- Convergée en 64-bit aujourd'hui.

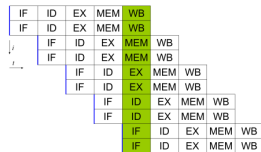
Intel C4004, processeur 4-bit (1971)

Parallélisme au niveau des instructions (ILP)

Permet d'exécuter des instructions indépendantes simultanément.



Exécution sans pipeline



Exécution superscalaire avec pipeline

- $a = b + c; c = d + e;$
 $f = a + c; g = c - a;$
- Possibilité 1: Pipeline. Différents étapes de deux instructions indépendantes peuvent être exécutées dans le pipeline en même temps.
- Possibilité 2: Exécution superscalaire: Des instructions indépendantes peuvent utiliser plusieurs unités d'exécution (i.e. ALU) dans un processeur **superscalaire**.
- La plupart des processeurs modernes sont superscalaires.

Data Parallelism

Effectuer les mêmes opérations indépendentes des tableaux de données.

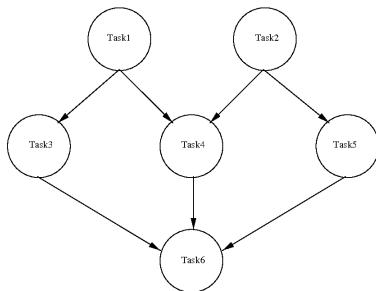
index	0	1	2	3	4	5	6	7
Array A	1	2	3	1	4	1	6	7
				+				
B	1	2	3	1	5	2	6	1
				=				
C	2	4	6	2	9	3	12	8

Addition des deux tableaux

- Convient au paradigme SIMD.
- Matériel bien adapté pour une exécution efficace (unité vectorielle CPU et GPU)
- Usage très répandu en calcul scientifique (matrices, vecteurs), graphique (rendering), traitement d'images et de signaux (filtres).

Task Parallelism

Il s'agit de l'exécution des tâches indépendentes dans un programme.



Task parallelism

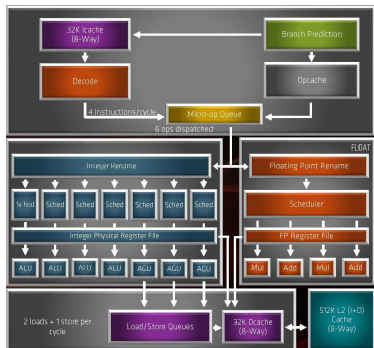
- Exemples des tâches indépendentes?
 - Appels aux fonctions: $\mathbf{a} = \mathbf{f}(\mathbf{x}); \mathbf{b} = \mathbf{g}(\mathbf{y});$
 - Blocs de code indépendentes dans une même fonction.
 - Exécution multiple d'un même programme avec différents paramètres (i.e. simulation).
- Chaque tâche peut s'exécuter sur une unité de calcul séparée.
- Il faut respecter les dépendences s'il y en a.
 - Équilibrage de charge?
 - Tout un domaine de recherche (ordonnancement).

- 1 Introduction
- 2 Calcul parallèle, pourquoi?
- 3 Concepts du calcul parallèle
- 4 Types de parallélisme
- 5 Architecture Parallèle**
- 6 Programmation Parallèle
- 7 Autour du développement de programmes parallèles

CPU

“Central Processing Unit”, unité de calcul générale consistant à

- Plusieurs unités d'exécutions (cœurs)
- Plusieurs niveaux de mémoire (registres, L1, L2, L3, RAM)
- Plusieurs ports d'exécution dans chaque cœur (ALUs, unités vectorielles)
- Unités vectorielles (AVX2, AVX512, Arm Neon, ...)
- Exécution de quelques threads (1-4) simultanément
- Capable d'exploiter le parallélisme au niveau des instructions (micro-op buffer, renumérotation d'instructions, renommer les registres, ...)
- Une partie considérable du circuit est consacrée au ILP + cache



L'architecture Zen 2

GPU

“Graphical Processing Unit”, unité de calcul spécifique vectorielle consistant à



L'architecture Nvidia Ampere

- Plusieurs unités d'exécutions (symmetric multiprocesseurs (SM))
- Plusieurs niveaux de mémoire (registres, mémoire partagée, L1, L2, RAM)
- Plusieurs unités vectorielles (2-4) larges (16-32 flottants) dans chaque SM
- Exécution de centaines de threads simultanément
- Grand tableau de registres (65K)
- Échange de threads très rapide
- La plupart du circuit est consacrée aux unités vectorielles
- Parallélisation prend l'effort

Supercalculateur / Cluster

Un ensemble de machines (CPU+GPU) connecté



Jolio Curie supercalculateur,
300K cœurs CPU, 1024 GPUs

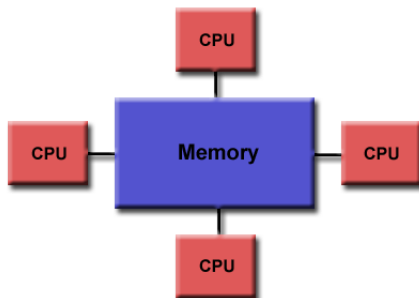
- Connexion par un réseau avec une topologie particulière (anneau, grille, torus, clique, etc.)
- Bibliothèques de communications adaptée à la topologie
- Capable de s'adresser à des problèmes de très grande taille
- Aujourd'hui, à l'échelle d'exaflops (10^{18} opérations flottants par seconde)

Mémoire partagée : caractéristiques générales

- Tous les processeurs peuvent accéder 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

Mémoire partagée : Uniform Memory Access (UMA)

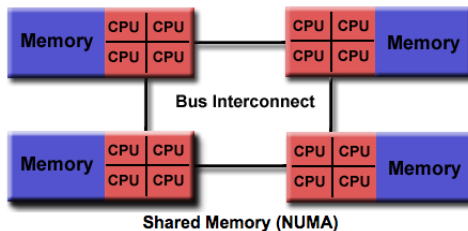
- Présenté par les machines SMP (Symmetric Multiprocessor)
- Processeurs identiques
- Temps d'accès à la mémoire égal pour tous les processeurs



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 accéder 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 Advantages :

- 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 :

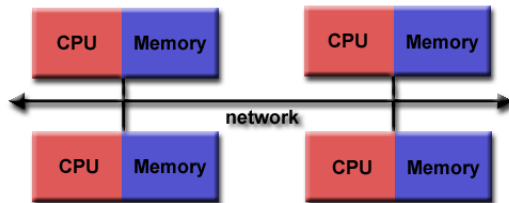
- Le manque de scalabilité entre la mémoire et les CPUs : ajouter plus de CPUs augmentera l'utilisation du bus partagé et pour les systèmes à cache cohérent, cela augmentera l'effort de gestion de la cohérence entre le cache et la mémoire
- C'est la responsabilité du programmeur de faire les synchronisations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

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 processeurs doivent être effectuées

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 Advantages:

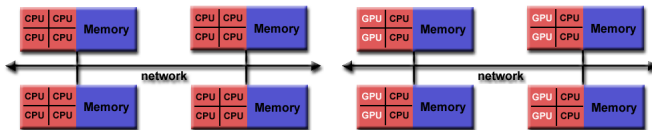
- La mémoire est scalable avec le nombre de processeurs
- Chaque processeur accède à sa mémoire rapidement sans ni interference avec les autres processeurs ni de coût additionnel pour maintenir une cohérence globale du cache

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 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 neud de calcul partage le même espace mémoire
- nécessitent des communications pour échanger les données entre les noeuds



- 1 Introduction
- 2 Calcul parallèle, pourquoi?
- 3 Concepts du calcul parallèle
- 4 Types de parallélisme
- 5 Architecture Parallèle
- 6 Programmation Parallèle**
- 7 Autour du développement de programmes parallèles

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**)

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 :

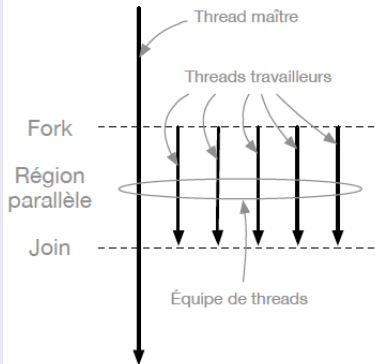
- **Sockets** bibliothèque, bas niveau
- **MPI Message Passing Interface** le standard pour les architectures à mémoire distribuée, le code parallèle est en général très différent du code séquentiel (**Plus de détails dans les cours à venir**)

Modèle Thread

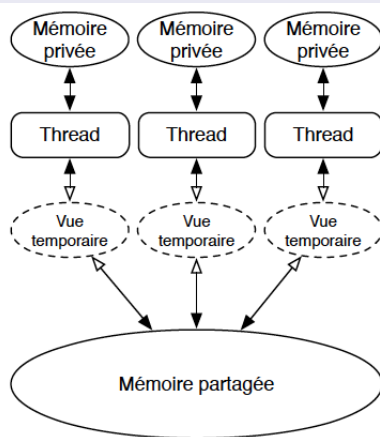
- IEEE POSIX Threads (PThreads)
 - Une API Standard UNIX, existe aussi sous Windows
 - Plus de 60 fonctions: *pthread_create*, *pthread_join*, *pthread_exit*, ...
- OpenMP
 - Une interface plus haut niveau, basée sur
 - des directives compilateur
 - des fonctions bibliothèques
 - un runtime
 - Une orientation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

OpenMP

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)

Produit scalaire : Séquentiel

```
#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 = 0 ; 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 );
```

Produit scalaire : Pthreads

```
#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;
}
```

```
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

```
#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

```
#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;

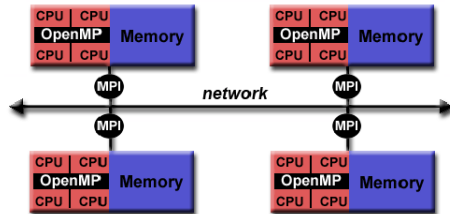
    // Initialization
    sum_local = 0.;
    for (size_t i = 0; i < SIZE; i++) {
        a[i] = i * 0.5;
        b[i] = i * 2.0;
    }

    // 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);
}
```

Modèle Hybride

- Plusieurs processus MPI, chacun gérant 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



Outline

- 1 Introduction
- 2 Calcul parallèle, pourquoi?
- 3 Concepts du calcul parallèle
- 4 Types de parallélisme
- 5 Architecture Parallèle
- 6 Programmation Parallèle
- 7 Autour du développement de programmes parallèles**

Plan

- 1 Introduction
- 2 Calcul parallèle, pourquoi?
- 3 Concepts du calcul parallèle
- 4 Types de parallélisme
- 5 Architecture Parallèle
- 6 Programmation Parallèle
- 7 Autour du développement de programmes parallèles

Un modèle de performance ; le roofline modèle

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

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

où **l'intensité opérationnelle** est le nombre d'opération flottantes effectuée par byte de DRAM transféré

Un modèle de performance ; le roofline modèle

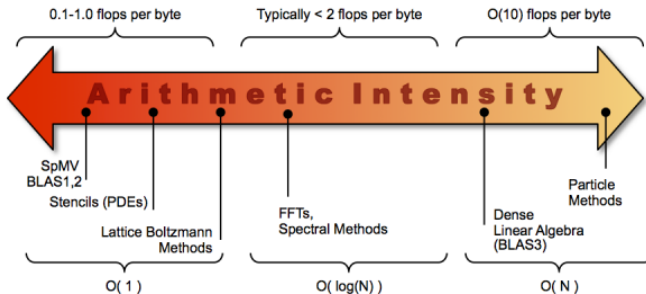
La performance qu'on peut atteindre (Gflop/s) est bornée par

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

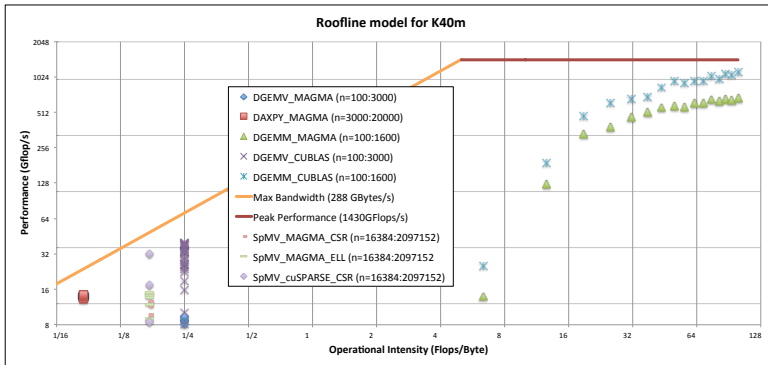
où **l'intensité opérationnelle** est le nombre d'opération flottantes effectuée par byte de DRAM transféré

- dépend de l'algorithme et de l'architecture cible
- produit matrice-vecteur dense : $I_{dgemv} \leq \frac{1}{4}$
- produit matrice-vecteur creux : $I_{SpMV} \leq \frac{1}{6}$, dépend du format du stockage (CSR ici)

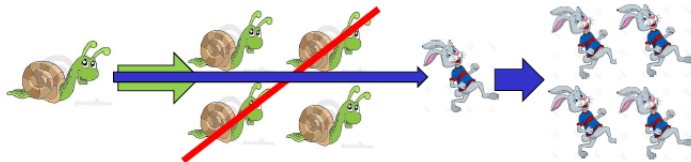
le roofline modèle : l'intensité opérationnelle



Modèle de performance pour NVIDIA Tesla K40



Où commencer ? : Optimisation des noyaux



Performance au niveau du coeur

- Réduire les défauts de cache : blocking, tiling, 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
- Re-structurer le programme ou utiliser/développer 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, C++ Parallel STD, ...)

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
- ...

References

Contact

Oguz Kaya

Université Paris-Saclay and LRI, Paris, France

oguz.kaya@lri.com

www.oguzkaya.com