



ትዕሊፎት ቅጽፊ ርጅፊ ርጅፊ ርጅፊ
ትዕሊፎት ቅጽፊ ርጅፊ ርጅፊ ርጅፊ

جامعة سيدي محمد بن عبد الله
كلية العلوم ظهر المهرار

Université Sidi Mohamed Ben Abdellah
Faculté des Sciences Dhar Mahraz



Parallel Computing

****BDSAS****

Projet :

Outils de programmation en traitement parallèle

Réalisé par : JOUAL Anouar
SADIQI Omar

Encadré par : Pr. Meknassi Mohammed

PLAN

- I. Les outils de programmation en traitement parallèle**
- II. Comparaison entre ces outils de programmation**
- III. L'importance de chacun de ces outils**

1. Les outils de programmation en traitement parallèle :

- **OpenMP : Open Multi-Processing**

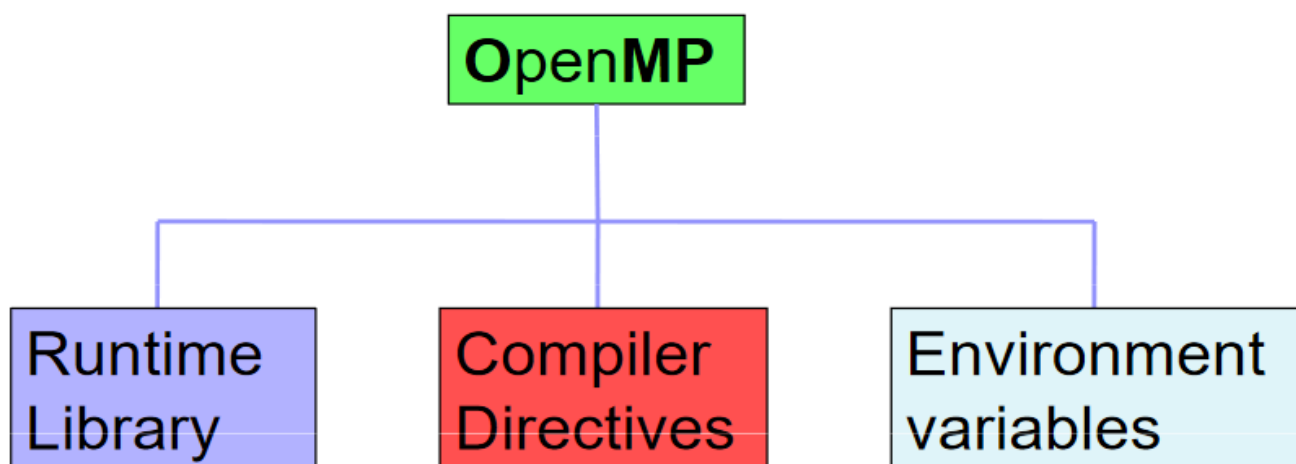
OpenMP (Open Multi-Processing) est une interface de programmation pour le calcul parallèle sur architecture à mémoire partagée. Cette API est supportée sur de nombreuses plateformes, incluant GNU/Linux, OS X et Windows, pour les langages de programmation C, C++ et Fortran. Il se présente sous la forme d'un ensemble de directives, d'une bibliothèque logicielle et de variables d'environnement.

OpenMP est portable et dimensionnable. Il permet de développer rapidement des applications parallèles à petite granularité en restant proche du code séquentiel.

La programmation parallèle *hybride* peut être réalisée par exemple en utilisant à la fois OpenMP et MPI.

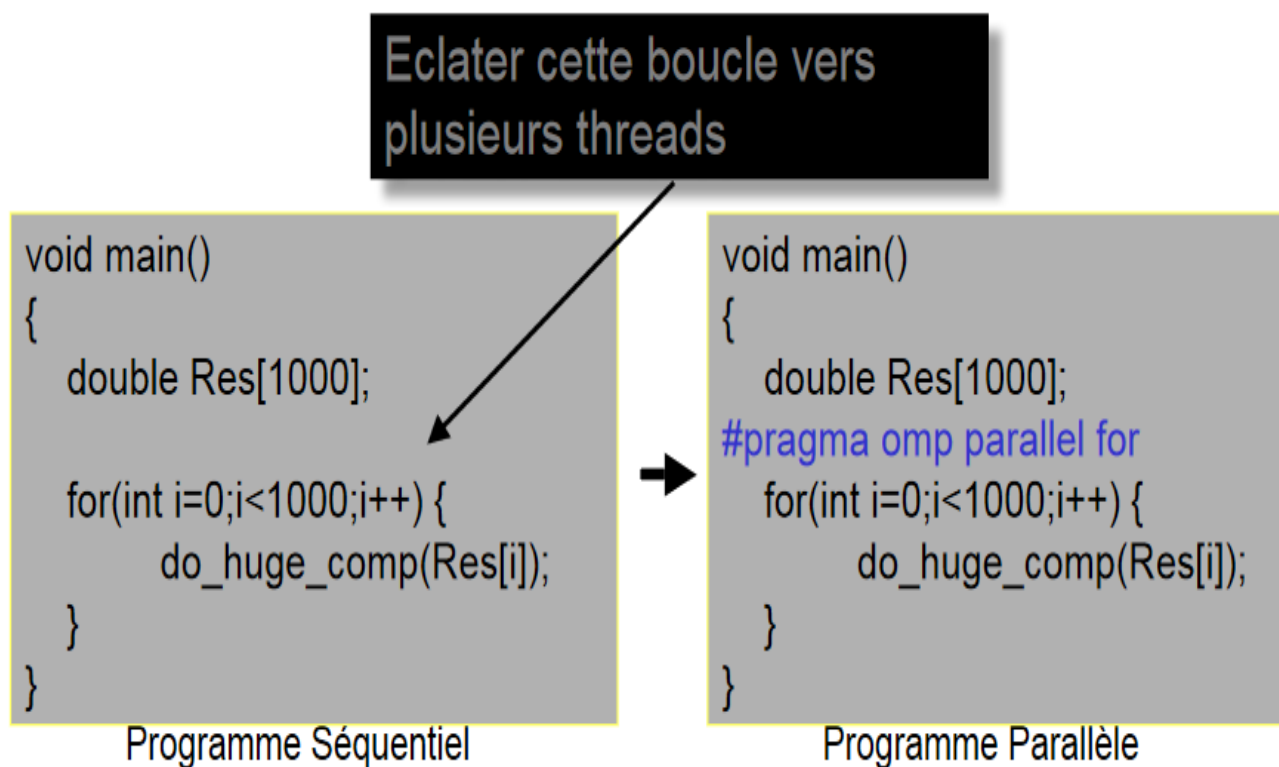
Le développement de la spécification OpenMP est géré par le consortium *OpenMP Architecture Review Board*.

✚ Structure d'OpenMP : Architecture logicielle :



OpenMP est utilisé «principalement» pour paralléliser les boucles:

- Trouver les boucles les plus coûteuses en temps
- Distribuer leurs itérations sur plusieurs threads



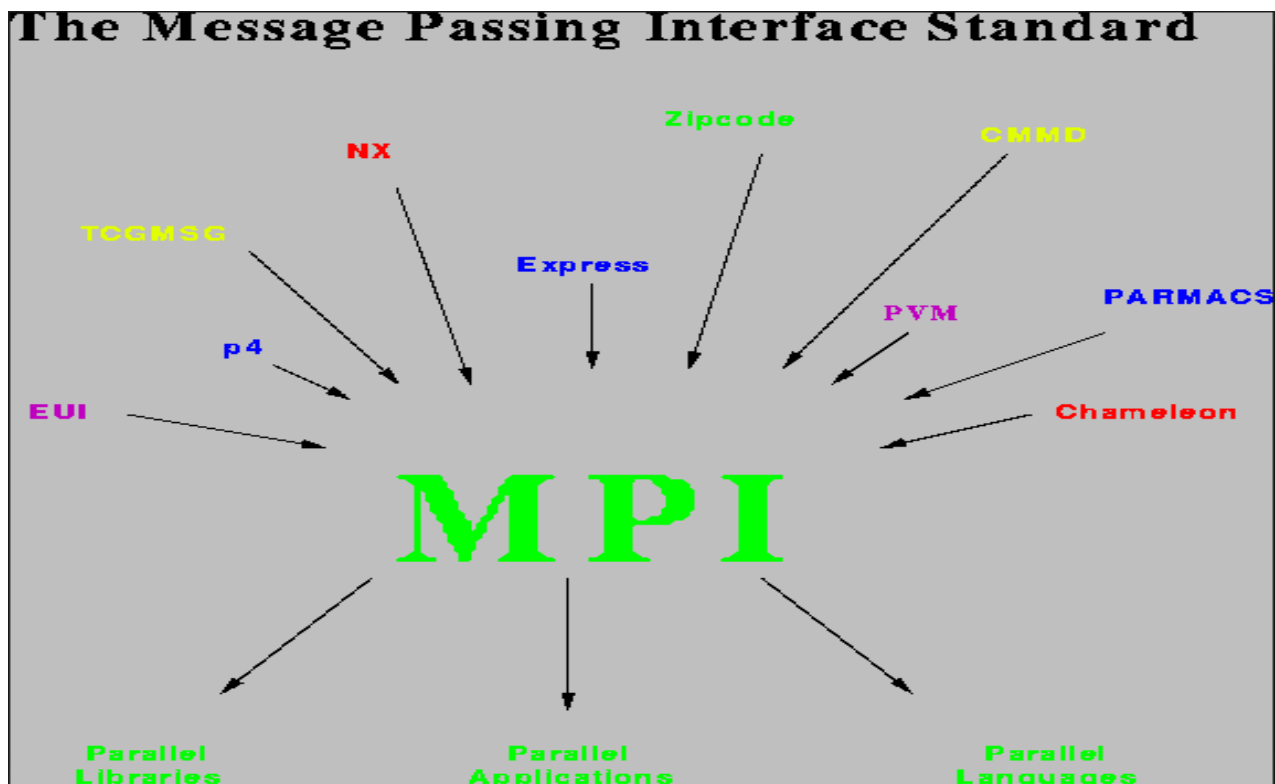
• MPI :

MPI (The Message Passing Interface), conçue en 1993-94, est une norme définissant une bibliothèque de fonctions, utilisable avec les langages C, C++ et Fortran. Elle permet d'exploiter des ordinateurs distants ou multiprocesseur par passage de messages.

Elle est devenue *de facto* un standard de communication pour des nœuds exécutant des programmes parallèles sur des systèmes à mémoire distribuée.

MPI a été écrite pour obtenir de bonnes performances aussi bien sur des machines massivement parallèles à mémoire partagée que sur des clusters d'ordinateurs hétérogènes à mémoire distribuée. Elle est disponible sur de très nombreux matériels et systèmes d'exploitation. Ainsi, MPI possède l'avantage par rapport aux plus vieilles bibliothèques de passage de messages d'être grandement portable (car MPI a été implémentée sur presque toutes les architectures de mémoires) et rapide (car chaque implémentation a été optimisée pour le matériel sur lequel il s'exécute).

- Plusieurs tâches (au sens Unix du terme) envoient et/ou reçoivent des données d'autres tâches.
- modèle SPMD (Single Program Multiple Data)
- Toutes les tâches MPI exécutent le même programme mais utilisent un sous-ensemble des données
- MPMD (Multi Program Multiple Data) est permis par les ajouts dans version MPI-2 de la norme (en dehors du cours)



MPI - types de systèmes parallèles :

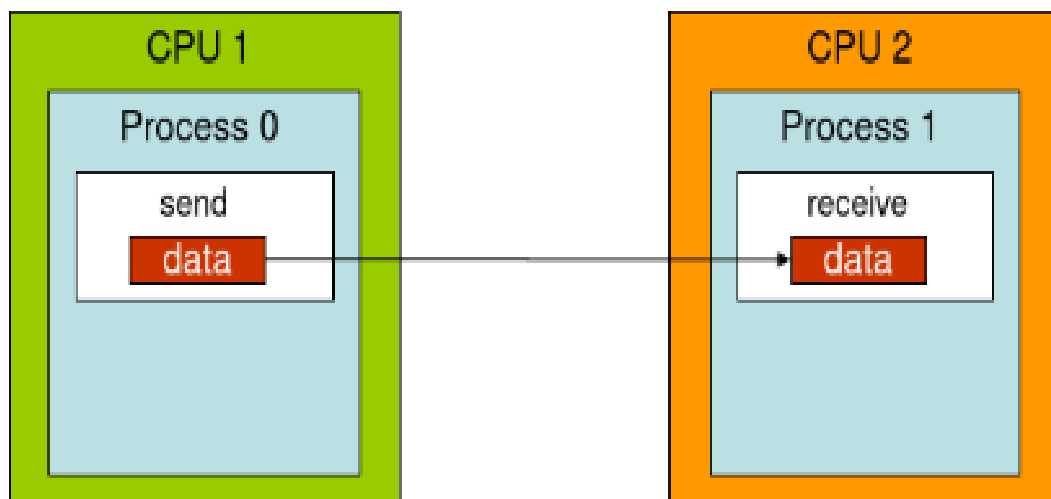
Systèmes à mémoire partagée

- Les cœurs d'un CPU partagent l'accès à la mémoire
- Programme dit multi-threadé, les threads partagent des variables localisées dans le même espace mémoire
- Coordination des threads par lecture/écriture des variables partagés

Systèmes à mémoire distribuée

- Programme constitués de plusieurs tâches (au sens Unix du terme, donc espace mémoire propre pas directement accessible aux autres tâches)
- Communication inter-tâches (coordination) via l'échange explicite de messages (via une interface réseau)
- Les ressources (données en mémoire) sont locales (ou privées) à une tâche
- le modèle par échange de message (message passing) est aussi valable sur architecture à mémoire partagée

Point-à-Point: MPI_Send / MPI_Recv



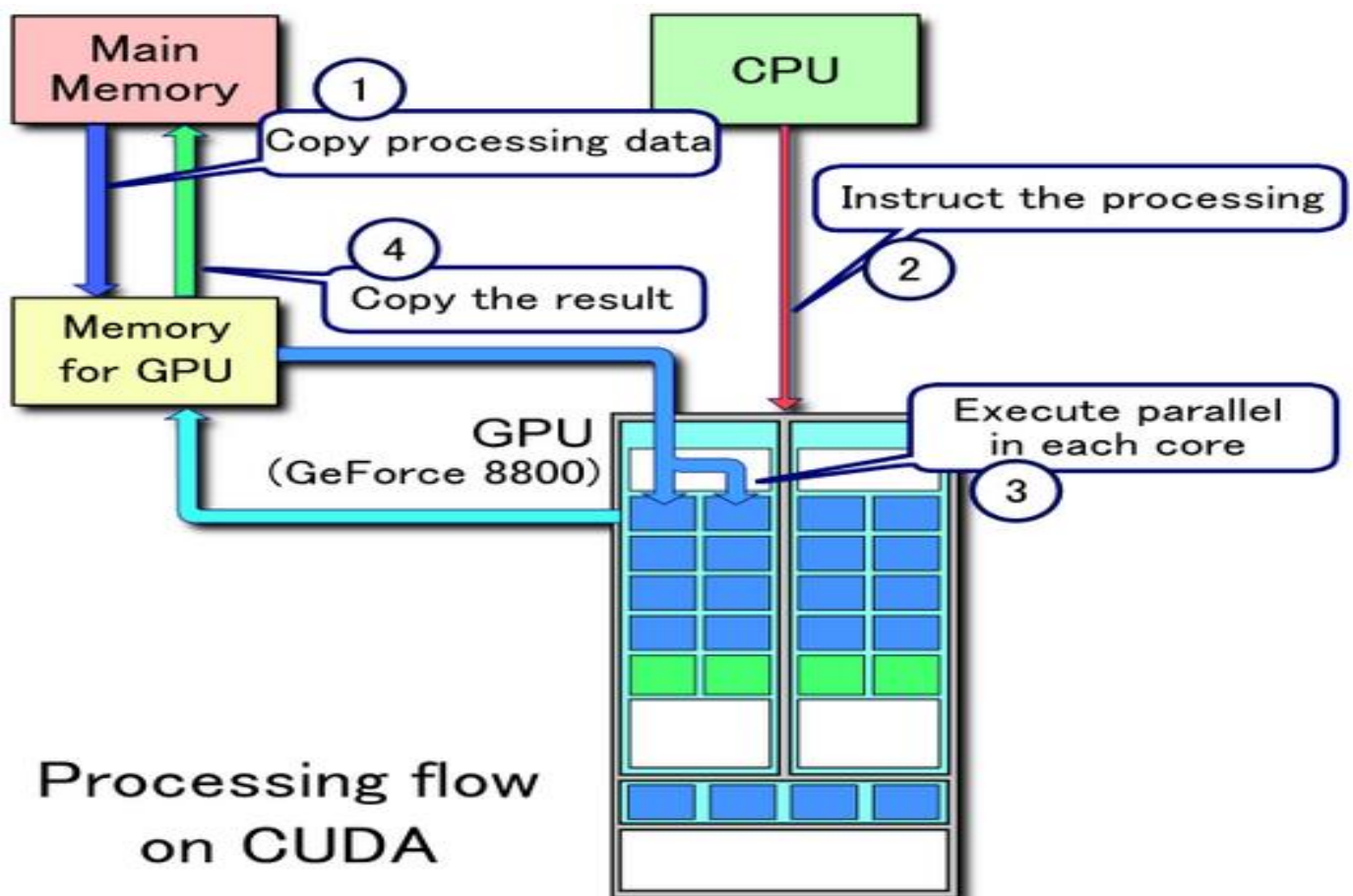
• CUDA :

CUDA (*Compute Unified Device Architecture*) est une technologie de GPGPU (*General-Purpose Computing on Graphics Processing Units*), c'est-à-dire qu'un processeur graphique (GPU) est utilisé pour exécuter des calculs généraux habituellement exécutés par le processeur central (CPU). CUDA permet de programmer des GPU en C. Cette technologie a été développée par Nvidia pour leurs cartes graphiques GeForce 8 Series, et utilise un pilote unifié utilisant une technique de *streaming* (flux continu). Le premier kit de développement pour CUDA a été publié le 15 février 2007¹.

CUDA présente plusieurs particularités par rapport à la programmation en C en proposant d'effectuer des calculs génériques sur GPU :

- Hiérarchisation explicite des zones de mémoire (privée, locale, globale) permettant d'organiser finement les transferts entre elles;
- Regroupement des *threads* en grilles de grilles : grille 1D, 2D ou 3D locale de threads pouvant se partager rapidement la mémoire locale. Les grilles locales sont ordonnées en grille globale permettant d'accéder à la mémoire globale.

Quelques réalisations combinent l'usage du langage Go, très orienté sur la programmation de processus concurrents et la gestion de mémoire sans fuites, avec celui de CUDA².

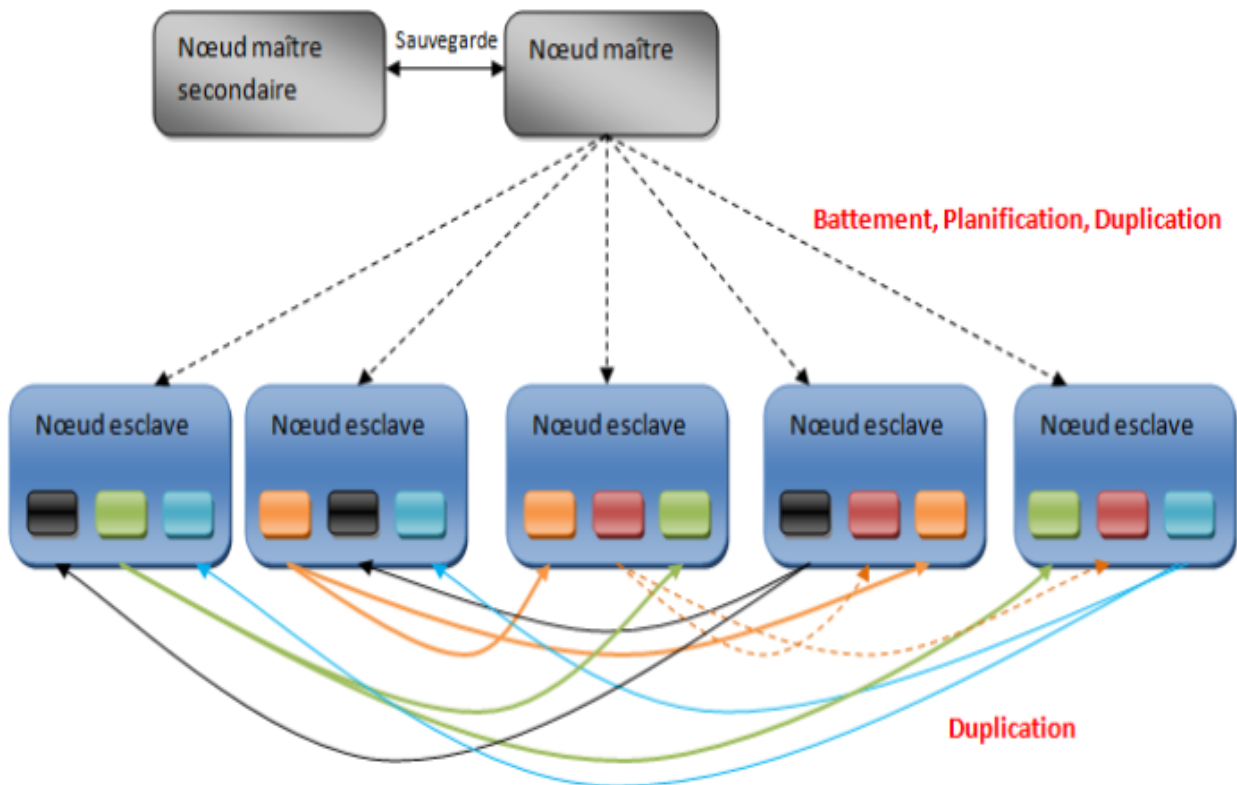


• Hadoop :

Hadoop est un framework libre et open source écrit en Java destiné à faciliter la création d'applications distribuées (au niveau du stockage des données et de leurs traitement) et échelonnables (scalables) permettant aux applications de travailler avec des milliers de nœuds et des pétaoctets de données. Ainsi chaque nœud est constitué de machines standard regroupées en grappe. Tous les modules de Hadoop sont conçus dans l'idée fondamentale que les pannes matérielles sont fréquentes et qu'en conséquence elles doivent être gérées automatiquement par le framework.

Hadoop a été inspiré par la publication de MapReduce, GoogleFS et BigTable de Google. Hadoop a été créé par Doug Cutting et fait partie des projets de la fondation logicielle Apache depuis 2009.

Le noyau d'Hadoop est constitué d'une partie de stockage: HDFS (Hadoop Distributed File System), et une partie de traitement appelé MapReduce. Hadoop fractionne les fichiers en gros blocs et les distribue à travers les nœuds du cluster. Pour traiter les données, Hadoop transfère le code à chaque nœud et chaque nœud traite les données dont il dispose. Cela permet de traiter l'ensemble des données plus rapidement et plus efficacement que dans une architecture supercalculateur plus classique^[réf. nécessaire] qui repose sur un système de fichiers parallèle où les calculs et les données sont distribués via les réseaux à grande vitesse.



2. Comparaison entre ces outils de programmation :

Comparaison générale :

MPI est un message passant le paradigme de parallélisme. Ici, vous avez une 'root machine' qui engendre des programmes sur toutes les machines dans son MONDE MPI. Tous les fils dans le système sont indépendants et par conséquent la seule façon(chemin) de communication entre eux est par des messages sur le réseau. La bande passante(la largeur de bande) de réseau et la sortie sont un du facteur le plus crucial dans la performance(prestation) de la mise en œuvre MPI. Idée : s'il y a juste un fil par machine et vous y avez beaucoup de cœurs, vous pouvez utiliser OpenMP le paradigme de mémoire partagé pour résoudre les sous-ensembles de votre problème sur une machine.

CUDA est un paradigme SMT de parallélisme. Il utilise l'état de l'art GPU l'architecture pour fournir parallelism. Un GPU contient (les blocs) marchant sur la même instruction d'une façon inflexible (Ceci est semblable au modèle de SIMD). D'où, si tous les fils dans votre système font beaucoup de même travail, vous pouvez utiliser CUDA. Mais la quantité(le montant) de mémoire(souvenir) partagée et la mémoire(le souvenir) globale(mondiale) dans un GPU est limitée et par conséquent vous ne devriez pas utiliser juste un GPU pour résoudre un problème énorme.

Hadoop est utilisé pour résoudre de grands problèmes sur le matériel(la quincaillerie) de marchandises utilisant la Carte Réduit le paradigme. D'où, vous ne devez pas vous inquiéter de la distribution de données ou la gestion de cas(d'affaires) de coin. Hadoop fournit aussi un système de fichiers HDFS pour stocker des données sur calcule des noeuds.

CUDA peut être très rapide, mais pour une sorte d'applications . Le transfert de données dans CUDA est souvent le goulot d'étranglement.

MPI est approprié pour l'environnement de groupe et le réseau de grande échelle d'ordinateurs.

OpenMP est plus approprié pour des systèmes multi-principaux. Donc c'est la vitesse dépend du nombre de cœurs.



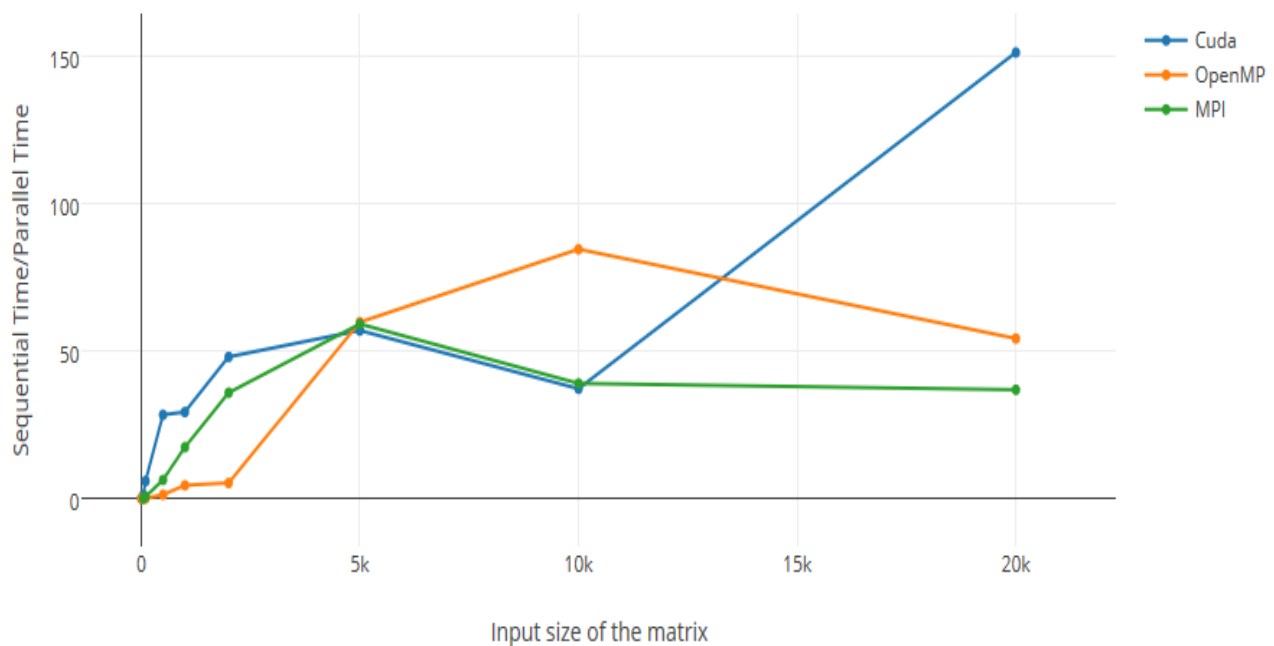
Hadoop, MPI et CUDA sont complètement orthogonal l'un à l'autre. D'où, il ne peut pas être juste de les comparer.

Quoique, vous puissiez toujours utiliser (CUDA + MPI) pour résoudre un problème utilisant un groupe de GPU'S. Vous avez toujours besoin d'un cœur simple pour exécuter la partie de communication du problème.

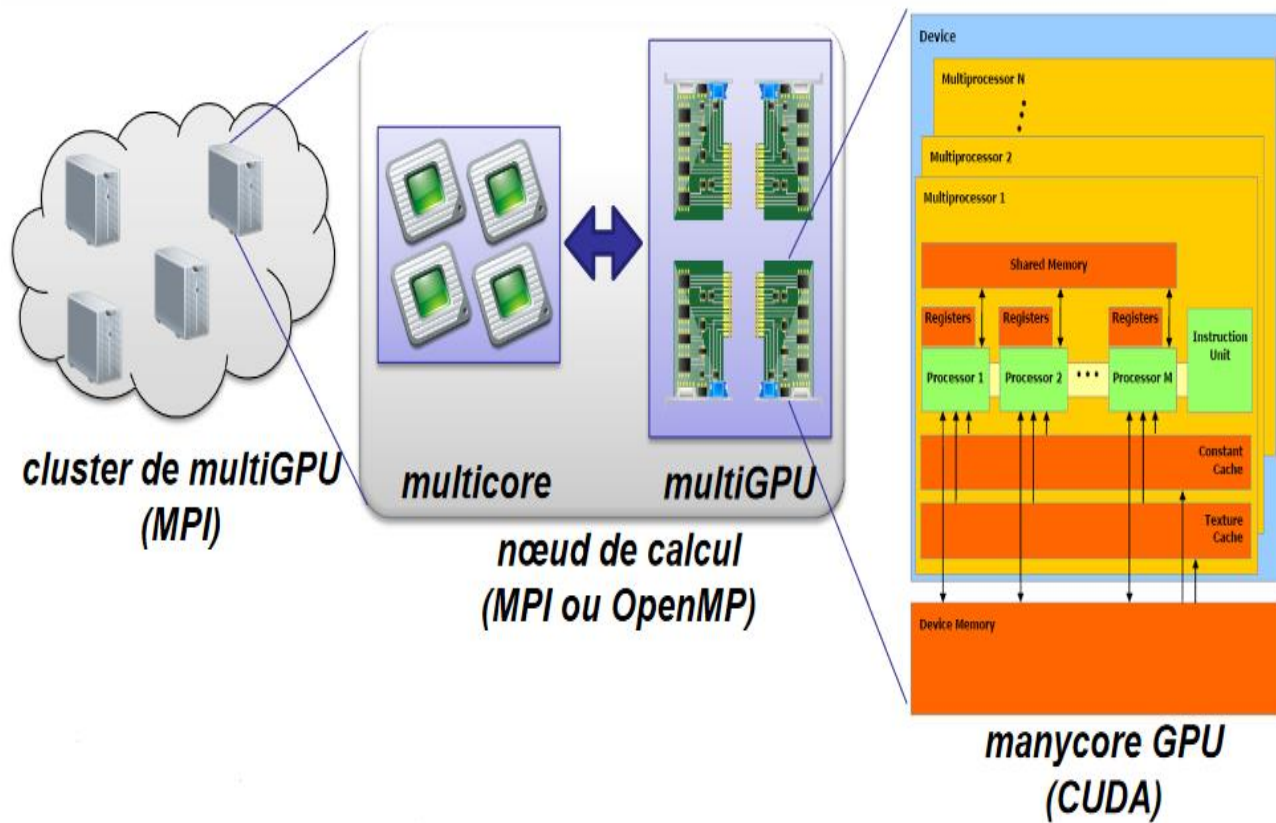
➤ **Comparaison entre MPI et OpenMP / Cuda :**

	MPI	OpenMP
<i>environnement</i>	processus	thread
<i>mémoire</i>	distribuée	partagée
<i>échange des données</i>	explicite (passage des messages)	implicite (mémoire partagée)
<i>synchronisation</i>	explicite	explicite / implicite
<i>attachement à une carte graphique</i>	rang du processus	identifiant du thread

Cuda v/s OpenMP v/s MPI speedup



➤ **Environnement MultiGPU : MPI/OpenMP/CUDA**



➤ OpenMP Vs MPI :

MPI pur Pro :

- Ordinateur portable à machines de mémoire(souvenir) distribuées et partagées.
- Balance(Échelles) au-delà d'un noeud
- Aucun problème de placement de données

OpenMP Pur Pro :

- Facile de mettre en œuvre le parallélisme
- Latence basse, haute bande passante (largeur de bande)
- Communication Implicite
- Granularité grossière et bonne
- Répartition de charge dynamique

MPI pur Con :

- Difficile de se développer et déboguer
- Haute latence, bande passante(largeur de bande) basse
- Communication explicite
- Grande granularité
- Répartition de charge difficile

OpenMP pur Con :

- Seulement sur Machines de mémoire partagée
- Échelle dans un noeud
- Problème de Placement de données possible
- Aucun ordre(commande) de fil spécifique

3. L'importance de chacun de ces outils :

a) OpenMP :

Modèle de programmation simple :

- Décomposition de données et communication manipulée(traitée) selon directives de compilateur

- Code source seul(simple) pour feuilletton(périodique) et codes parallèles
- Aucun commandant n'écrit du code en série
- Mise en œuvre portable
- Parallelization progressif
- Début de la partie la plus critique ou consommatrice de temps du code

➤ *Avantages d'OpenMP :*

- Espace d'adressage globale plus facile à utiliser
- Le partage des données entre les tâches est rapide
- Contrairement au modèle message passing, les directives openMP peuvent s'introduire dans le code de façon incrémentale ce qui permet une parallélisation incrémentale
- Même code pour les versions séquentielle et parallèle, les directives openMP sont introduites sous forme de lignes de commentaires qui ne seront prise en compte par le compilateur que lorsqu'on le demande.
- le code séquentiel n'est en général pas modifié lorsqu'il est parallélisé avec OpenMP, diminue la possibilité d'introduire des bugs

➤ Exemple avec OpenMP :

a. OpenMP approche classique : Fine grain (Grain fin)

Méthode la plus simple & commune pour utiliser openMP , en utilisant la directive Do pour partager le travail entre les threads (processus légers).

```
!$OMP PARALLEL DO PRIVATE(i,j) & !$OMP SHARED (residu)
do j= 1, ny
do i= 1, nx
residu(i,j)=0.d0
enddo
enddo
!$OMP END PARALLEL DO
```

b. 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;
}
```

b) MPI :

- Fonctionne sur un très grand nombre de configuration
- Librairie complète pour le calcul parallèle
- Débogage « facile » par rapport à du SMP
- Faciliter la programmation par envoi de message
- Portabilité des programmes parallèles
- Utilisable avec une architecture
- heterogene
- Cible stable pour des outils/langages de plus haut niveau

Exemple :



Exemple de programmation MPI

Communications Send-Recv

Standard-Bloquant (MPI_Send – MPI_Recv)

```
/* RELAXATION LOOP
OI = 0;  NI = 1;
for (cycle = 0; cycle < NbCycle; cycle++) {
    /* - Computation loop
    ...
    /* - Frontier exchange
    if (NbPE > 1) {
        if (Me == 0) {
            MPI_Send(...,Me+1,...);  MPI_Recv(...,Me+1,...);
        } else if (Me == NbPE - 1) {
            MPI_Send(...,Me-1,...);  MPI_Recv(...,Me-1,...);
        } else {
            MPI_Send(&V[NI][1][1],...,Me-1,...);
            MPI_Send(&V[NI][SmallBoard][1],...,Me+1,...);
            MPI_Recv(&V[NI][0][1],...,Me-1,...);
            MPI_Recv(&V[NI][SmallBoard+1][1],...,Me+1,...);
        }
    }
    /* - Index switch
    OI = NI; NI = 1-NI;
}
```

Simple mais */
peu portable
peu efficace */

*/

*/

c) CUDA :

Avantages GPU :

- I - Rapide pour exécuter des instructions en parallèle (gain 100 x max).
- I - Excellent pour les gros travaux.
- I - Lit les instructions de manière asynchrone; retourne le contrôle au CPU après réception du code.
- I - Offre une grande puissance de calcul à faible coût.

CUDA est une architecture de traitement parallèle développée par NVIDIA permettant de décupler les performances de calcul du système en exploitant la puissance des processeurs graphiques (GPU).

Alors que des millions de GPU compatibles avec CUDA ont été vendus, des milliers de développeurs de logiciels, de scientifiques et de chercheurs utilisent CUDA dans une grande gamme de domaines, incluant notamment le traitement des images et des vidéos, la chimie et la biologie par modélisation numérique, la mécanique des fluides numérique, la reconstruction tomodensitométrique, l'analyse sismique, le ray tracing et bien plus encore.

Le calcul informatique a évolué en passant du traitement central exclusif des CPU vers les capacités de co-traitement offertes par l'association du CPU et du GPU. Pour permettre ce nouveau paradigme informatique, NVIDIA a conçu l'architecture de traitement parallèle CUDA, aujourd'hui incluse dans les GPU GeForce, ION Quadro, et Tesla en offrant ainsi une base matérielle significative aux développeurs d'applications.

Du côté du grand public, la plupart des applications majeures de traitement vidéo, incluant des produits d'Elemental Technologies, MotionDSP et LoiLo, Inc, sont ou seront bientôt accélérées par CUDA.

Du côté de la recherche scientifique, CUDA a été reçu avec enthousiasme. CUDA permet par exemple d'accélérer AMBER, un programme de simulation de dynamique moléculaire utilisé par plus de 60 000 chercheurs du public et du privé afin d'accélérer la découverte de nouveaux médicaments pour l'industrie pharmaceutique

➤ Example: Host Code

```
// allocate host memory
unsigned int numBytes = N * sizeof(float)
float* h_A = (float*) malloc(numBytes);

// allocate device memory
float* d_A = 0;
cudaMalloc((void**)&d_A, numbytes);

// copy data from host to device
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

// execute the kernel
increment_gpu <<< N/blockSize , blockSize >>>(d_A, b, N);

// copy data from device back to host
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// free device memory
cudaFree(d_A);
```