

High Performance Computing et calcul parallèle

D. Cornu,
UTINAM / Observatoire de Besançon

Master 2 - P2N & PICS

Automne 2019

Crédits et remerciement à Benoit Semelin, LERMA (OBSPM)

Supports pour le cours

Tous les supports de cours (slides, codes, exercices, corrections, ...) sont disponibles sur GitHub et seront mis à jour progressivement :

Deyht/HPC_M2

Quelques commandes utiles :

```
(en 313c) export http_proxy="http://..."  
sudo apt-get install git  
git clone https://github.com/Deyht/HPC_M2  
git pull
```

Pour ne pas risquer de perdre votre travail en cas de mise à jour copiez ces fichiers dans un autre répertoire !

Ressources supplémentaires

Des cours et des exercices de Fortran ainsi que d'autres langages et surtout de clacul parallel OpenMP et MPI peuvent etres trouvés sur le site :

http://www.idris.fr/formations/supports_de_cours.html

Par ailleurs des supports de cours de Fortran (dont un polycopié reprennant les bases), de C, et de Shell (commandes unix) sont disponibles sur la page du cours de MNI de Jacques Lefrere :

<http://wwwens.aero.jussieu.fr/lefrere/master/mni/>

Organisation du cours

Cours 1 : Introduction des concepts du HPC. Cours et exercices sur OpenMP.

Cours 2 : Exercices OpenMP, présentation du projet NBody pour la suite du cours.

Cours 3 à 5 : Projet NBody, objectifs différents P2N et PICS, à adapter au rythme de chacun

Cours 5 et 6 : Cours et exercices sur MPI. Poursuite du projet en OpenMP ou passage du projet en MPI.

P2N uniquement : *Cours et exercices sur CUDA (programmation GPU) en Décembre, après le cours Machine Learning.*

Performance séquentielle

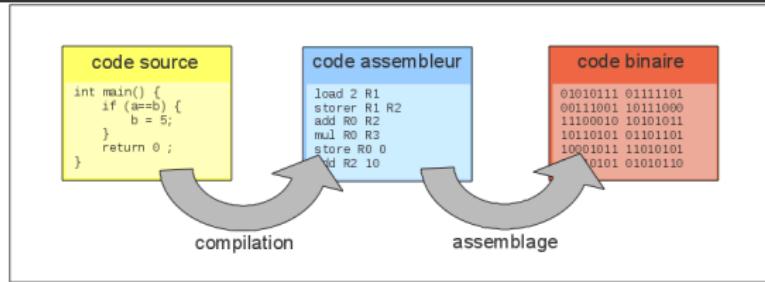
Exercice 1 :

- Déclarer 2 tableaux de réels tridimensionnels (dimensions de départ $128 \times 128 \times 128$), alloués dynamiquement sur le "tas/heap"
- En remplir un tel que : $A(i, j, k) = \text{real}(i + j + k)$
- Remplir l'autre tableau tel que :

$$B(i, j, k) = \sum_{dx=i-2}^{i+2} \sum_{dy=j-2}^{j+2} \sum_{dz=k-2}^{k+2} A(dx, dy, dz) \quad (1)$$

- Il faut donc se limiter à i, j, k variant de 3 à 126
- Afficher $B(\text{size}/2, \text{size}/2, \text{size}/2)$
- Compiler le programme à votre habitude, et l'exécuter avec la commande "time ./mon_programme"

Optimisation séquentielle : Compilation



Rappel : La compilation consiste à **réinterpréter le language** de programmation (haut niveau) vers un language plus élémentaire pour la machine : code assembleur puis "**language machine**" binaire. C'est aussi à ce moment là que les différents fichiers d'un même programme, ainsi que les librairies associées, sont mis en commun pour former un seul exécutable.

Le compilateur joue un rôle primordial dans **l'optimisation des performances**. C'est ce qui fait la force des languages compilés (C, Fortran, ...) face aux languages interprétés (Python (intermédiaire), MATLAB, Mathematica, ...)

Options de compilation

Basiques :

- -Wall : affiche toutes les erreurs détectables par le compilateur
- -fcheck=bounds : vérifie les dépassemens de tableaux
- -v : messages plus détaillés (verbose)
- -g : permet la construction de messages de débug (avec gdb)

Options de compilation

Basiques :

- -Wall : affiche toutes les erreurs détectables par le compilateur
- -fcheck=bounds : vérifie les dépassemens de tableaux
- -v : messages plus détaillés (verbose)
- -g : permet la construction de messages de débug (avec gdb)

Optimization :

- -On : optimisation selon n
 - -O0 : réduit le temps de compilation
 - -O1 : accélère légèrement l'exécution et réduit la taille de l'exécutable
 - -O2 : accélère l'exécution
 - **-O3 : accélère beaucoup la vitesse d'exécution** au prix d'un temps de compilation élevé et d'un usage mémoire plus important
 - -Os : optimise la taille de l'exécutable
 - -Ofast : optimisation supérieure à O3 mais ne garantie plus l'intégrité des résultats (à utiliser avec prudence)
 - -Og : optimise les performances sans impacter la capacité de débugage

Le compilateur contient des centaines d'options ...

Le fonctionnement de la mémoire machine

Les ordinateurs utilisent uniquement la base 2 (binaire), représentée sous la forme de bits (0 ou 1)

Un entier est **encodé de manière exacte** le plus souvent sur 32 bits

n	signe	2^6	2^5	2^4	2^3	2^2	2^1	2^0
-128	1	0	0	0	0	0	0	0
-127	1	0	0	0	0	0	0	1
...
-1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0
+1	0	0	0	0	0	0	0	1
...
+127	0	1	1	1	1	1	1	1

TABLE 2.1 – Représentation des entiers relatifs sur 8 bits.

Le fonctionnement de la mémoire machine

Les ordinateurs utilisent uniquement la base 2 (binaire), représentée sous la forme de bits (0 ou 1)

Un entier est **encodé de manière exacte** le plus souvent sur 32 bits

Les réels quant à eux sont représentés en "**virgule flottante**"

$$r = sb^em = sb^e \sum_{i=1}^q p_i b^{-i} \quad (2)$$

- $s = \pm 1$ le signe
- b la base (décimale, binaire, hexa,...)
- e un exposant entier
- m est la mantisse

Le fonctionnement de la mémoire machine

Les ordinateurs utilisent uniquement la base 2 (binaire), représentée sous la forme de bits (0 ou 1)

Un entier est **encodé de manière exacte** le plus souvent sur 32 bits

Les réels quant à eux sont représentés en "**virgule flottante**"

$$r = sb^em = sb^e \sum_{i=1}^q p_i b^{-i} \quad (2)$$

- $s = \pm 1$ le signe
- b la base (décimale, binaire, hexa,...)
- e un *exposant* entier
- m est la *mantisso*

Le nombre de bit est alors séparé entre la représentation de la *mantisso* soit sa précision relative, et *l'exposant* fixant le domaine de valeurs.

C'est cette définition qui explique que les calculs sont plus **précis** lorsque les valeurs sont proches de 1 car plus de bits sont alors dédiés à la représentation de la mantisse !

Random-Access Memory *ou* RAM

Random-Access Memory ou RAM



Random-Access Memory ou RAM

La mémoire "volatile" RAM est une architecture de mémoire physique qui permet de lire ou écrire des données de manière équivalente **quelle que soient leurs emplacement physiques.** (à mettre en opposition avec les disques durs, dvd, bandes magnétiques, etc ...).

Puisque les données peuvent être n'importe où, il est alors nécessaire de **conserver une "adresse"** pour s'y retrouver.

Lors de **l'allocation d'un tableau** une seule adresse est enregistrée, celle du début du tableau.

Les tableaux multidimensionnels sont stockés **en une seule dimension** dans la RAM ! L'ordre dans lequel les éléments sont stockés est alors primordial.

Row-Major / Col-Major

Row-major order

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

C/C++, Python, ...
L'indice rapide est à droite

Column-major order

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Fortran, ...
L'indice rapide est à gauche

"Cache Miss"

Nous reviendrons en détail sur l'architecture d'un processeur plus loin. Pour autant, il faut maintenant expliquer que les processeurs disposent de "RAM" interne en faible quantité mais bien plus rapide appelé "**caches**".

A chaque calcul, le processeur va copier **la "zone" mémoire** contenant la variable concernée dans ses caches.

Si les opérations suivantes concernent **les éléments suivants en mémoire**, alors le cache n'est recopié dans la RAM qu'une fois toutes les opérations locales effectuées.

Si les opérations ne sont pas continues en mémoire, alors chacune d'entre elles est précédée d'une copie dans le cache puis suivie d'une mise à jour de la RAM. On parle alors de **Cache Miss**.

Fonctionnement voulu

Cas de cache miss

Performance séquentielle

Exercice 1 - partie 2 :

- Reprendre l'exercice précédent et tenter d'appliquer des directives d'optimisation, en particulier **-O3**.
- Assurez vous que les indices des boucles sont dans le bon ordre pour éviter les cache miss.
- Avec ces optimisations, le gain de temps peu atteindre **un facteur 10 !** Estimez votre facteur de gain de temps "speedup".
- Augmentez la taille des matrices ET/OU augmentez le padding. Observez le comportement.

Calcul Parallèle

Qu'est ce que le calcul parallel

Utiliser plusieurs processeurs en même temps pour réaliser un calcul

Avantages

- **Vitesse** : dans la cas optimum, le temps de calcul est divisé par le nombre de processeurs.
- **Taille des données** : on peut partager la mémoire entre plusieurs processeurs.
- **Précision** : la division en petites taches similaires permet de conserver une meilleure précision.
- **Coûts** : il est souvent bien moins cher de produire de nombreux processeurs lents que un seul processeur très rapide.

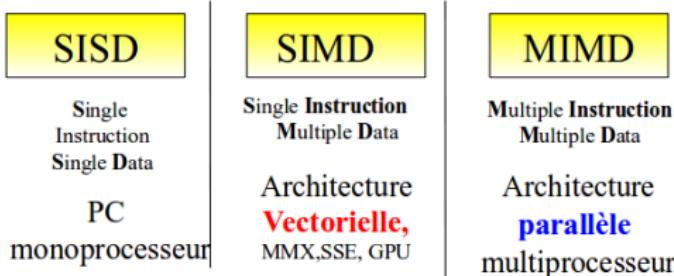
Désavantages

- **Partage** : des tâches laissé au contrôle du développeur.
- **Echange** : d'informations entre les coeurs difficile et demande un type d'infrastructure spécifique.
- **Language** : il peut être spécifique à certaines architectures.
- **Efficacité** : réduite. Chaque machine fonctionnera de manière moins efficace que avec des tâches purement indépendantes.

L'utilisation appropriée d'une machine de calcul permet de minimiser son coût écologique.

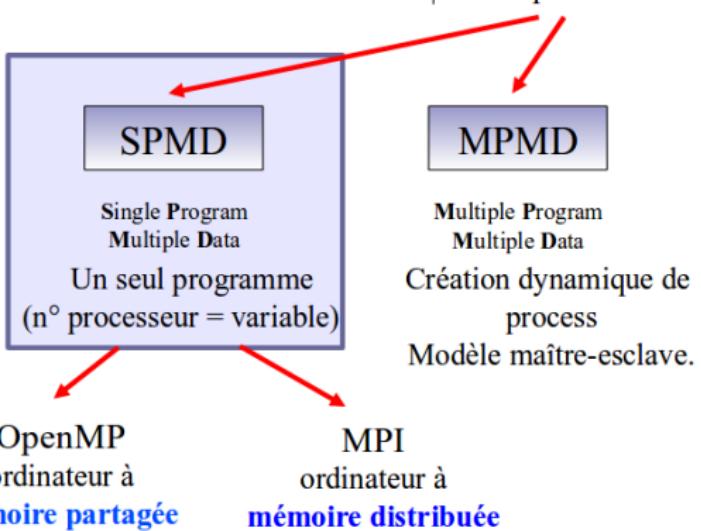
Méthodes de parallélisation

Architecture matérielle:



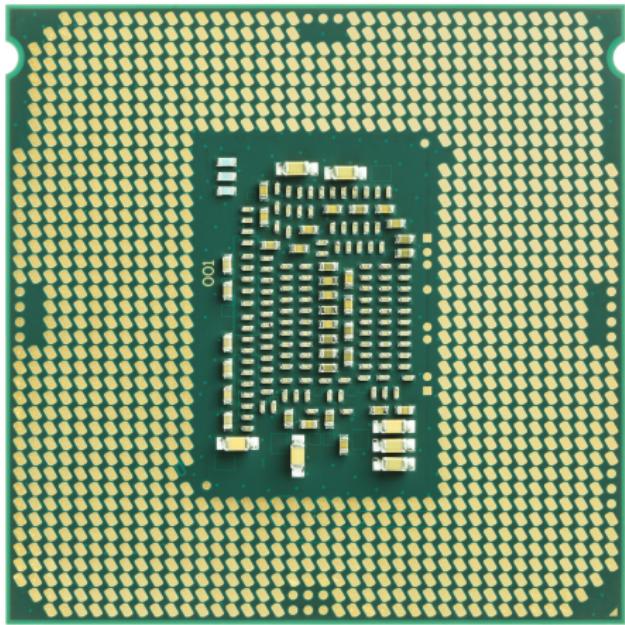
Modèle de programmation:

Le plus utilisé →

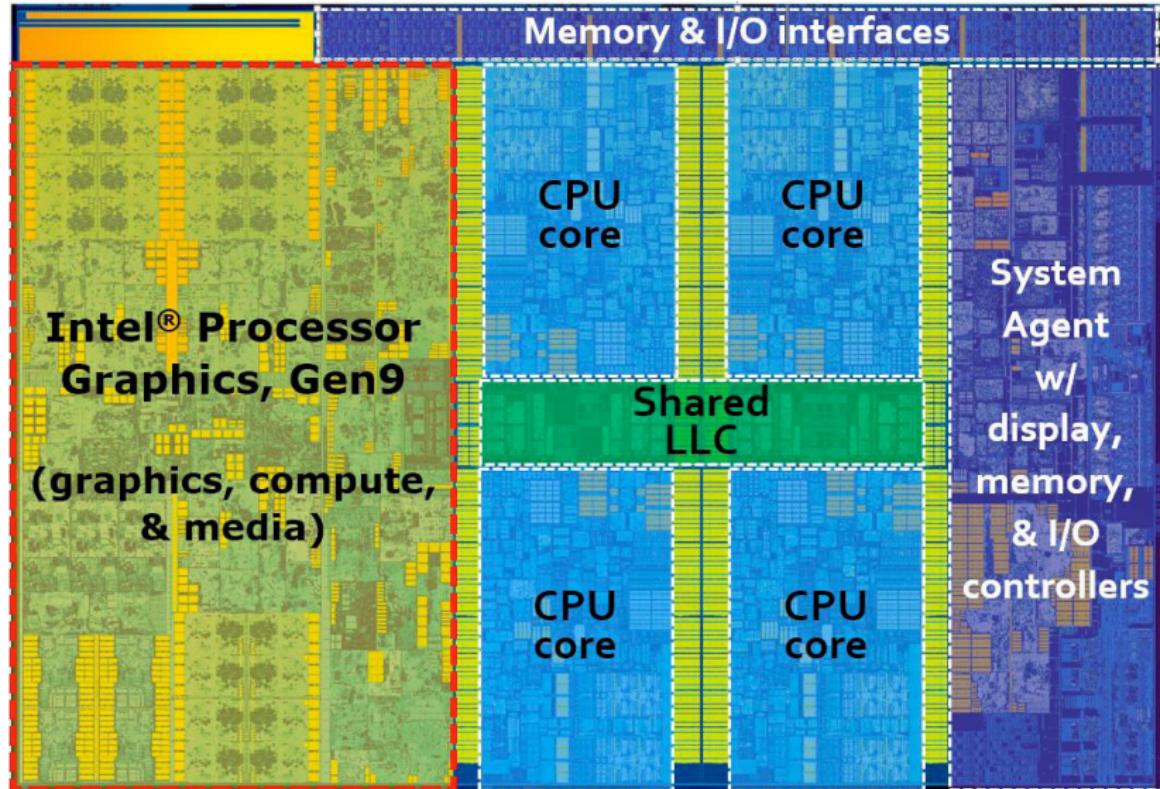


Outils de parallélisation:

Architecture CPU (Compute Processing Unit)



Architecture CPU (Compute Processing Unit)

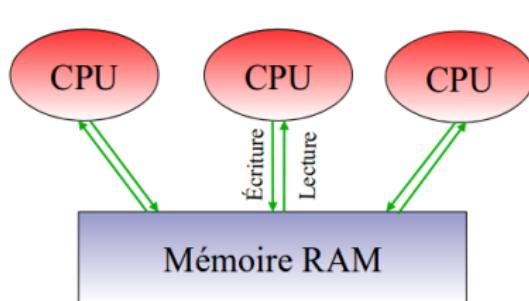


Architecture CPU (Compute Processing Unit)

Caractéristiques Intel i7 - 6700 k :

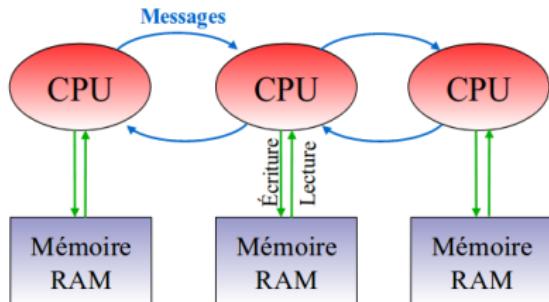
- 1.75 milliards de transistors
- 4 coeurs physiques / 8 coeurs logiques - "Hyperthreading"
- fréquence : 4GHz (turbo à 4.2GHz)
- Caches : L3-8 Mo (partagée), L2-256Ko (/ cœur), L1-32Ko (/ cœur)
- finesse de gravure : 14 nanomètres
- Max ram : 64 Go

Mémoire Partagée Vs Distribuée



Les processeurs partagent la même mémoire

- Pas ou peu de surcoût lié à la parallélisation
- Possibles conflits d'accès à la mémoire
- Matériel coûteux. Une seule machine avec de nombreux coeurs
- Le plus souvent un nombre de coeurs < 128



Chaque processeur dispose de sa propre mémoire et ne peut pas accéder à celle des autres

- Surcoût pour l'échange des messages
- Matériel plus accessible, mais coût d'architecture
- Sensible à la performance du système de communication
- Nombre de coeurs ≈ illimité

Mémoire Partagée Vs Distribuée

Les plus gros calculateurs ont une architecture qui **mélange les deux méthodes**. De nombreuses machines multi-processeurs sont reliées entre elles par des réseaux complexes et extrêmement performants pour former d'immenses clusters de calcul.

Exemple - Jean Zay (GENCI-CINES) : Actuellement en déploiement prévu pour la 14e place mondial :

Remplace Ada et Turing et sera contemporain à Occigen et Joliot Curie (respectivement 90e et 47e)

- 40 coeurs par noeud (2 xeon 20 coeurs) et 192 Go ram
- Plus de 1500 Noeuds classiques HPC
- 261 noeuds convergés CPU et GPU avec l'ajout de 4 Nvidia V100 32Go
- Puissance de calcul totale de 14 Petaflops (14×10^{15} floating-point opérations par secondes)
- 343 To de mémoire vive totale et 1 Po d'espace de stockage SSD.

Comprendre les outils

Plusieurs niveaux auxquels effectuer la parallélisation

- Langages ou extension
 - CUDA (GPU)*
 - OpenCL(divers)*
- Bibliothèques
 - Message Passing Interface MPI*
 - PosixThreads
- Directives de compilation
 - OpenMP*
 - Accélération GPU
- Compilateur autonome
 - Fortran/C inclu dans le compilateur

*Méthodes usuelles pour le calcul scientifique

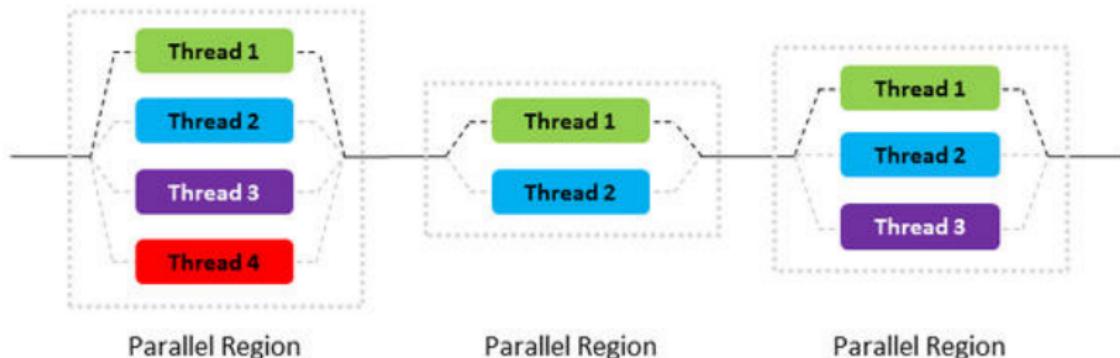
Open Multi-Processing

Introduction

OpenMP se caractérise par des **instructions de compilation** qui seront interprétées par le "pré-processeur". Les interfaces communes sont en Fortran, C et C++.

Les premiers standards d'OpenMP datent de 1997 et sont encore mises à jour très régulièrement aujourd'hui.

C'est l'une des méthodes les plus répandues de parallélisation, à la fois pour sa simplicité et ses performances.



Syntaxe

!\$OMP DO SCHEDULE(DYNAMIC,500)

On y distingue :

- **La sentinelle** : qui permet de dire au pré-processeur qu'il s'agit d'une directive OpenMP.
- **La directive** : qui décrit l'action à effectuer.
- **Une clause** : optionnelle, qui permet de modifier le comportement de la directive. Il peut y en avoir plusieurs et l'ordre n'a pas d'importance.

Zone parallèle

Une zone parallèle est commencée avec la directive
!\$OMP PARALLEL

puis est ensuite terminée avec
!\$OMP END PARALLEL

Les fonctions :

`integer :: omp_get_num_threads, omp_get_thread_num`

Permettent respectivement de récupérer le nombre total de threads dans une zone parallèle, et d'obtenir le numéro du thread en cours (de 0 à N-1).

Compilation et nombre de threads

Pour coder un programme parallel il convient toujours de commencer par le code séquentiel, et de l'optimiser en partie.

On peut ensuite placer les directives OpenMP nécessaires.

La compilation s'effectue alors avec l'option supplémentaire "**-fopenmp**" (En effet OpenMP est directement pré-intégré avec la majorité des compilateurs).

Le nombre de coeurs à utiliser pour les programmes OpenMP est alors défini au niveau du système via la **commande (dans la console)** :
`export OMP_NUM_THREADS=N`

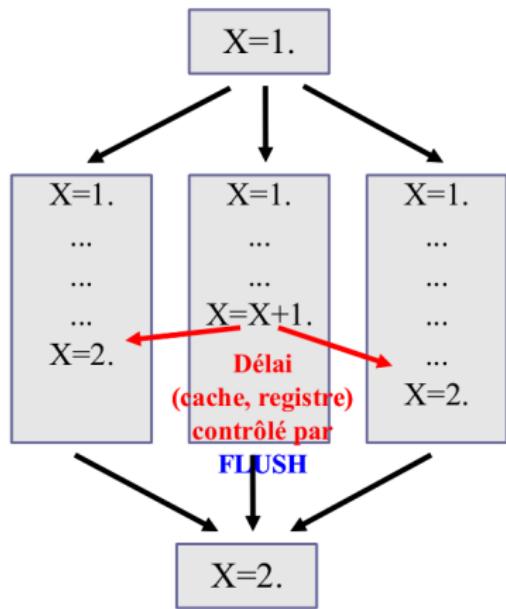
On peut alors lancer le programme comme d'habitude, les zones parallèles vont s'exécuter avec les N threads définis précédemment (on peut changer le nombre de threads sans recompiler).

Exercice 2 : Hello World

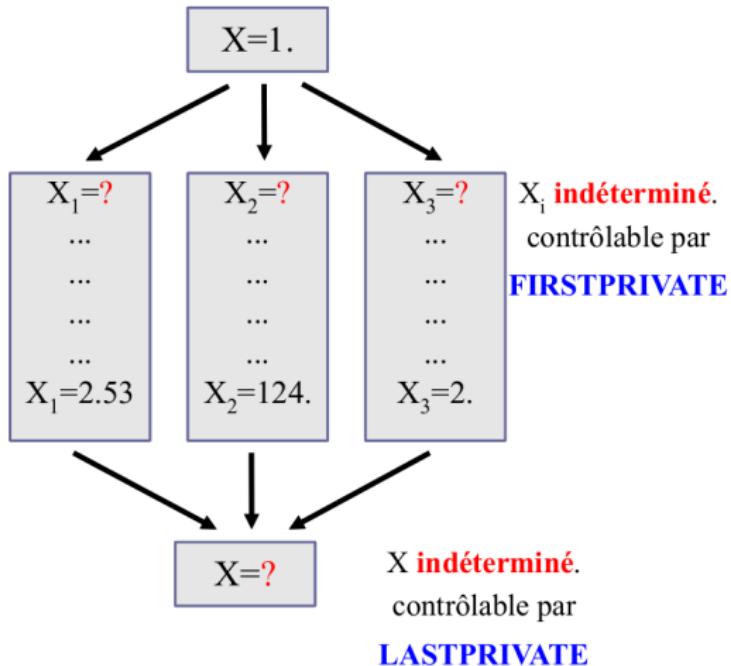
- Ecrivez un simple programme qui définit une zone parallèle et fait afficher à l'intérieur de celle-ci le rang du thread (sans déclarer de variable).
- Compilez à l'aide des explications précédentes et exécutez votre programme à plusieurs reprises pour des nombres de threads différents.
- Que constatez-vous ?

Les variables

Variable **SHARED**



Variable **PRIVATE**



X indéterminé.
contrôlable par
LASTPRIVATE

Les variables

Ce sont des clauses, à ajouter après la directive **PARALLEL**

SHARED(var1,var2,...) :

La variable est partagée entre tous les threads, sa valeur est commune.

C'est le statut par défaut !

Il faut s'assurer de la synchronisation des coeurs si ces variables sont éditées !(directive FLUSH)

PRIVATE(var1,var2,...) :

Chaque thread a sa propre copie de la variable.

L'espace mémoire est dupliqué.

Les variables déclarées à l'intérieur d'une zone parallèle sont automatiquement private !

Clauses annexes :

FIRSTPRIVATE(var1,var2,...) : conserve la valeur précédent la zone parallèle

LASTPRIVATE(var1,var2,...) : la valeur du thread exécutant la dernière mise à jour est conservée

Exercice 3 : partage des variables

- Ecrire un programme qui déclare une zone parallèle dans laquelle chaque thread effectue une boucle et affiche à chaque itération le rang du thread et l'indice courant.
- Chaque thread doit effectuer cette opération de manière indépendante.
- Observez le comportement de la sortie en fonction de la clause des variables en jeu. Affichez la même sortie en dehors de la zone parallèle.
- Vous pouvez ajouter à ce programme une variable shared qui sera éditée une fois par chacun des coeurs, et observer sa valeur à la sortie.

Threads VS Cores

Les threads définis jusqu'à présent sont des jeux d'instructions parallèles. Néanmoins, rien n'interdit de définir un **nombre de threads supérieur au nombre de coeurs physiques** sur le processeur.

Cette pratique n'a que peu d'intérêt en calcul puisque plus de threads engendre un nombre d'instructions plus élevé.

Il n'y aura à priori aucun speedup, voire une perte de performances.

Néanmoins les CPU modernes disposent d'une technologie nommée **Hyper Threading**. Celle-ci permet à un cœur physique de traiter **deux files d'instructions** simultanément et ainsi de rendre simultanées des tâches différentes : Calcul, mouvement mémoire, instruction SIMD...

Cette technologie permet d'obtenir un speedup correct pour un nombre de threads allant jusqu'au double de coeurs physiques sur le CPU.

Partage du travail

La façon la plus commune de répartir le travail entre les threads est de leur attribuer des morceaux **d'une boucle indépendante** (formalisme SIMD).

```
1 !$OMP DO
2 do i=1, N
3 ...
4 end do
5 !$OMP END DO
```

Cette directive doit être placée **à l'intérieur d'une zone parallèle**. La variable de la boucle sera automatiquement définie comme private et initialisée à la valeur de départ qui répartit le travail **équitablement entre les threads**.

Exercice 4 : "loop workshare"

- Reprendre le code du premier exercice sur l'optimisation séquentielle.
- Déclarez une zone parallèle avec les bonnes variables en private
- Parallélisez la boucle do extérieure
- S'assurez de la stabilité du résultat affiché avec la version séquentielle
- Exécutez le programme plusieurs fois avec des nombres de threads différents
- Tracez le speedup pour votre machine (si possible avec hyper threading)

Stratégie de répartition

La clause **SCHEDULE(strategie,N)** permet de contrôler la répartition des itérations d'une boucle entre les différents threads.

- **STATIC** : comportement par défaut, chaque thread reçoit N itérations, les threads s'attendent avant de passer à la répartition suivante. Si N n'est pas spécifié, alors les threads se partagent de manière égale les itérations.
- **DYNAMIC** : Chaque thread reçoit N itérations, mais les threads ne s'attendent pas. Un nouveau groupe N est immédiatement donné à un thread qui a terminé l'étape précédente (par défaut N = 1)
- **GUIDED** : Les paquets sont de taille exponentielle décroissante avec pour minimum N.
- **RUNTIME** : La stratégie est définie au moment de l'exécution avec la variable OMP_SCHEDULE.

Essentiel car rien ne garantie que les coeurs soient équivalents, ou que leurs charges de travail soient identiques (boucle avec conditions internes).

Performance parallèle stratégique

Exercice 4 - partie 2 : stratégies

- Reprendre l'exercice précédent et définir une stratégie.
- Tentez différentes stratégies avec différentes valeurs de N.
- Tracez le speedup à nombre de threads constant en faisant varier N, pour plusieurs stratégies.

Opération "Atomique"

Lorsque l'on édite une variable **SHARED** dans un boucle parallèle il faut faire attention à ce que plusieurs coeurs ne mettent pas à jour la variable simultanément.

La directive **!\$OMP ATOMIC** permet de spécifier que l'opération qui suit doit être effectuée par un seul thread à la fois.

Exercice 5 : Calcul de π par intégrale de Riemann

- Ecrire un programme qui calcul l'intégrale de Riemann telle que :

$$\pi = \int_0^1 \frac{4}{1 + x * x} \quad (3)$$

- Utilisez une boucle conjointement avec la directive **ATOMIC** pour paralléliser cette somme.
- Tracez le speedup pour cet algorithme. Réfléchissez à des méthodes d'optimisation.

Boucle avec réduction

La clause **REDUCTION(operator :var)** permet de définir une opération sur une variable qui devra être effectuée par tous les coeurs une fois la boucle terminée. Cette opération peut être +,-,*,.OR.,.AND.,.MAX,.MIN,...

Exercice 5 - partie 2 : Réduction

- Utilisez cette méthode pour simplifier le programme de calcul d'intégrale de π .
- Comparez les résultats obtenus entre les différentes méthodes, tracez les speedup.

Autres directives OpenMP

- **SECTIONS / END SECTIONS** : définit plusieurs zones de codes qui peuvent être exécutés simultanément. Nombre de threads égale au nombre de sections.
- **WORKSHARE / END WORKSHARE** : doit être capable de répartir de manière autonome un ensemble d'instructions contenues dans la zone. Fonctionne rarement.
- **SINGLE / END SINGLE** : la zone définie est exécutée une seule fois par le premier thread qui y arrive. Par défaut les autres threads attendent sauf si **NOWAIT**.
- **MASTER/ END MASTER** : la zone définie est exécutée une seule fois par le thread "master" numéro 1. Les autres sautent la zone et n'attendent pas.

Synchronisation des Threads

- **BARRIER** : Synchronisent les threads. Ils s'arrêtent tous à cette directive avant de repartir ensemble. Implicitement incluse dans END PARALLEL, END DO, END SECTIONS, END SINGLE, END WORKSHARE.
- **FLUSH(var1,var2,...)** : le thread met à jour les variables spécifiées (utile uniquement si variable shared). Il vide les tampons, les caches, les registres et vérifie si un autre thread n'a pas modifié la variable. Implicitement incluse dans BARRIER, CRITICAL, END CRITICAL, END DO, END SECTIONS, END SINGLE, END WORKSHARE, ORDERED, END ORDERED, PARALLEL, END PARALLEL.

Exercice 6 : "Nested loop"

Tentez de paralléliser le programme "nested_loop.f90" en utilisant la directive **FLUSH**. Vous aurez sûrement besoin de la clause **NO WAIT** sur l'une des boucles.

Ce qui n'a pas été traité

- Les types de données définis par l'utilisateur.
- Les opérations de réduction définies par l'utilisateur.
- L'utilisation des verrous ou "Mutex", définir mieux l'inter-bloquage (peu utile en calcul scientifique).
- Les zones parallèles imbriquées.
- Limites physiques de OpenMP.
- Comparaison avec l'auto parallélisation de Fortran.
- etc ...

Projet : Nbody

Projet pour le reste des séances : Réaliser un programme qui calcule l'interaction gravitationnelle exacte en 3D entre N particules de masse égale ($1/N$).

Pour cela vous définirez comme condition initiale pour vos particules une position aléatoire dans une sphère de rayon 1, ainsi qu'une rotation solide selon l'axe z de norme 1 par unité de temps.

Vous définirez un pas de temps à ajuster en fonction du résultat de vos premières simulations, et un temps maximum pour définir la durée de la simulation.

Méthode type Leap Frog : A chaque itération calculez la position $x_{i+1/2}$, l'accélération $a_{i+1/2}$ et ainsi la vitesse v_{i+1} .

Identifiez des grandeurs physiques conservées à calculer afin de vérifier la stabilité de l'intégrateur puis de la parallélisation.

Parallelisez cette algorithme de la façon qui vous paraît la plus simple en utilisant OpenMP. Tracez les speedup, identifiez les meilleures stratégies. Votre calcul est-il optimal ?