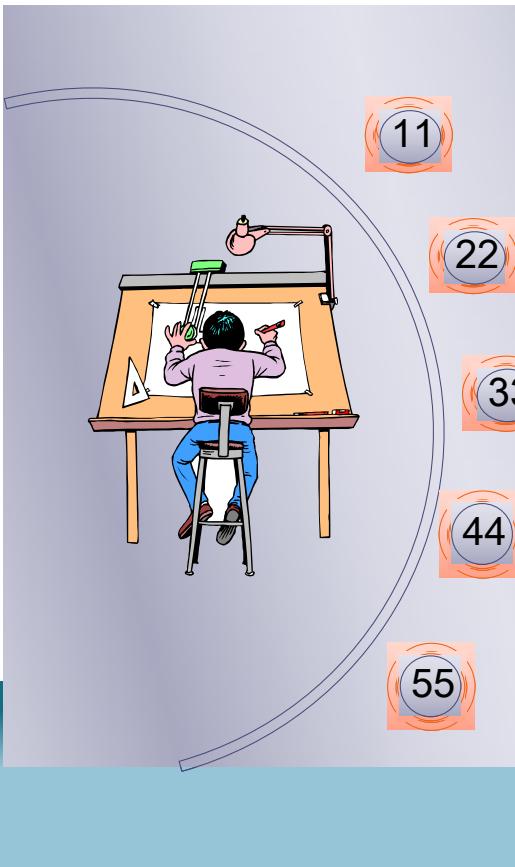


Architectures et Programmation Parallèles

Section :
FIA2-GL

Prof. : Tahar ALIMI

Plan



Introduction

Architectures parallèles : Types et formes

Modèles de programmation parallèles

Bénéfices du parallélisme

OpenMp : Open Multi-Processing

Introduction (1/2)

➤ Qu'est-ce que le parallélisme ?

- Exécuter plusieurs actions coordonnées en même temps.
- En informatique, le parallélisme consiste à mettre en œuvre des architectures [...] permettant de traiter des informations de manière simultanée, ainsi que les algorithmes spécialisés pour celles-ci.
- Ces techniques ont pour but de réaliser le plus grand nombre d'opérations en un temps le plus petit possible. (**Source : wikipedia**).

Introduction (2/2)

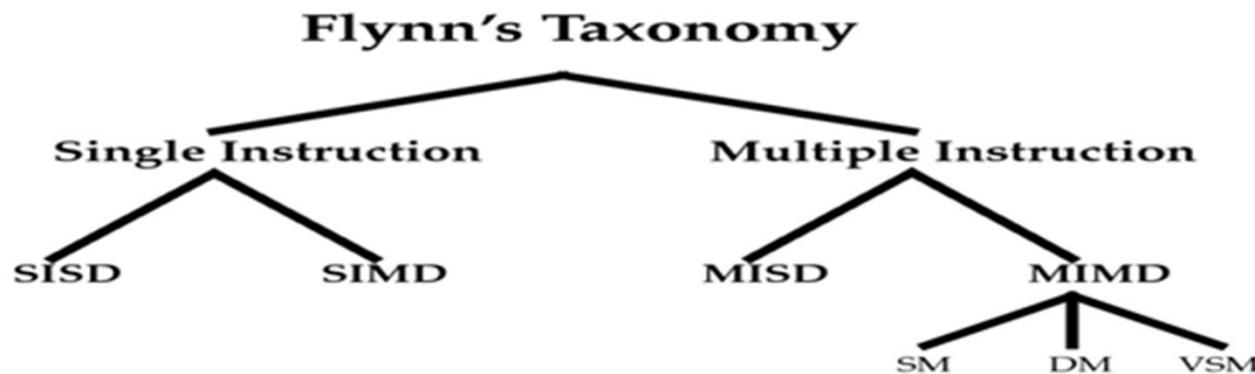
- ⌚ Les architectures parallèles sont les ordinateurs sur lesquels ce paradigme est utilisable ;
- ⌚ Les modèles de programmation parallèles sont les techniques de programmation qui permettent de l'exploiter.
- Temps d'exécution d'un programme séquentiel : $t_{\text{exéc}} = n_{\text{ instructions}} * t_{\text{ instruction}}$
- Temps d'exécution d'un programme parallèle : $t_{\text{exéc}} = (n_{\text{ instructions}} * t_{\text{ instruction}}) / p$
avec p : le nombre de threads
- “ Exemple : Vitesse

Un produit de matrices

- ⌚ Produit de deux matrices 10000×10000 : $C = AB$
- ⌚ Calcul de 10000 fois $c_{i,j} = \sum a_{i,k} * b_{k,j}$
- ⌚ Nombre d'opérations flotantes = $2 * 10^4 * 10^4 * 10^4 = 2.10^{12}$
- “ Temps d'exécution :
- ⌚ ENIAC : 2.10^9 s = 63 ans
- ⌚ PC de 1997 : 24 minutes
- ⌚ ASCI Red (1997) : 2 s
- ⌚ PC actuel : 0.2 s → 5 par seconde
- ⌚ TaihuLight : 0.02 ms → 50000 par seconde

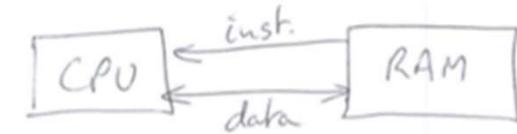
Architectures parallèles (1/3)

➤ Taxonomy de Flynn



➤ A) SISD : Simple Instruction – Simple Data

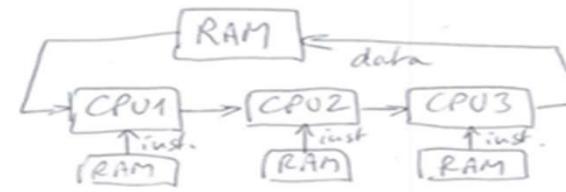
- “ Une seule unité de calcul, séquentielle
- “ Accédant à une seule donnée à la fois
- “ **Exemple :** ENIAC, un PC de 1982 (Intel8088)



Architectures parallèles (2/3)

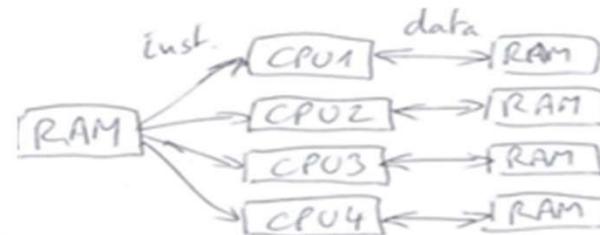
➤ B) MISD : Multiple Instructions – Simple Data

- “ Plusieurs unités de calcul, qui exécutent des instructions différentes
 - “ Mais accédant à une seule donnée à la fois
- Fonctionnement de type pipeline
- “ **Exemples :** pipeline d'instructions du cœur de processeur (RISC à 5 étages : 1988), pipeline graphique...



➤ C) SIMD : Single Instruction – Multiple Data

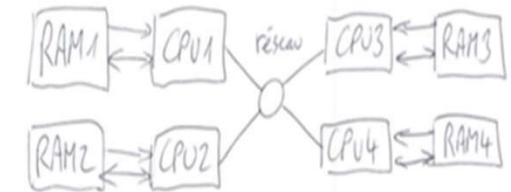
- “ Plusieurs unités de calcul, exécutant toutes la même instruction (SI)
 - “ Sur des jeux de données différents
- Fonctionnement de type vectoriel
- “ **Exemples :** MasPar (1990), instructions vectorielles (MMX, SSE, AVX), GPUs



Architectures parallèles (3/3)

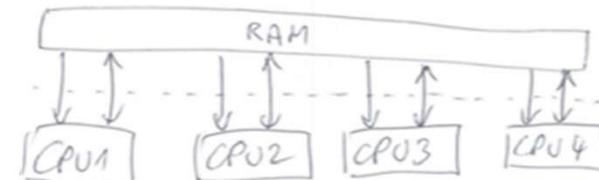
➤ D) MIMD : Multiple Instructions – Multiple data (Mémoire distribuée)

- “ Plusieurs machines SISD, reliées entre elles par un réseau
- “ Fonctionnement asynchrone, chaque machine a sa mémoire
- “ **Exemples :** IBM RS/6000 (1990), clusters de PCs



➤ E) MIMD : Multiple Instructions – Multiple data (Mémoire partagée)

- “ Plusieurs machines SISD partageant une mémoire unique
- “ **Exemples :** Cray X-MP (1983), SGI Challenge (1990), processeurs multi-coeurs



➤ F) MIMD-VSM (Virtually Shared memory) :

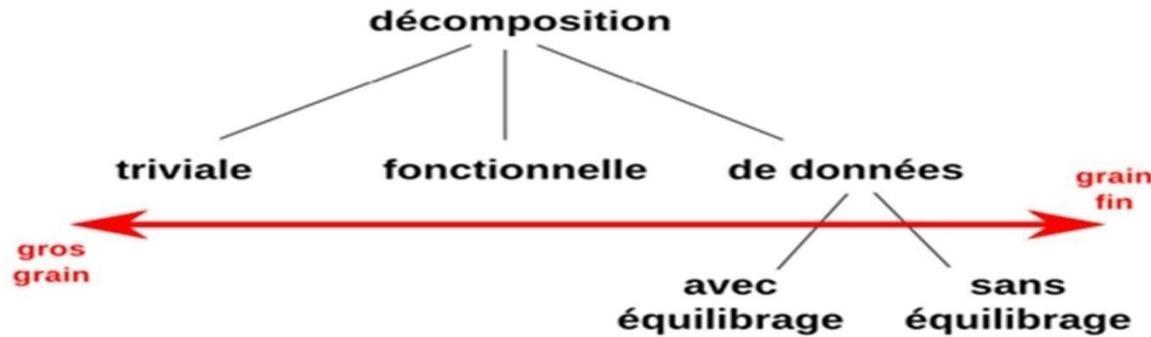
- “ Aussi appelée **NUMA** (Non-Uniform Memory Access) .
- “ Architecture physique MIMD à mémoire distribuée
- “ Mais... l'ensemble de la machine possède une vision globale de la mémoire
- “ Un processeur X peut accéder à la mémoire du processeur Y, sans l'interrompre, grâce à un réseau qui interconnecte les RAM !
 - **Exemples :** Cray T3D (1993), SGI Origin 2000 (1996), ...

Formes de parallélisme (1/1)

- **Parallélisation des bits** : Plus qu'un bit est exécuté par cycle d'horloge.
- **Parallélisation des instructions**: Les instructions d'une telle tâches sont exécutées simultanément.
- **Parallélisation des données** : C'est le fait de répartir des données sur des nœuds qui s'exécutent en parallèle. Généralement c'est le cas d'une boucle **loop**. Une tâche est subdivisée en sous-parties qui seront traitées.
- **Parallélisation des tâches** : C'est le fait de répartir (ou créer) des tâches qui doivent s'exécuter en parallèle. Il s'agit d'un parallélisme fonctionnel ou de contrôle. Deux tâches T1 et T2 sont exécutées séparément par différents processeurs.

Modèles de programmation (1/1)

➤ Granularité :



“ OpenMP **grain fin ou Fine-grain** (FG) : utilisation des directives OpenMP pour partager le travail entre les threads, en particulier au niveau des boucles parallèles.

“ **Remarque : Synchronisation des threads**

- Contrairement aux processus, les threads partagent leur vision de la mémoire (comme dans le modèle d'architecture MIMD-SM)

➤ **OpenMP :**

- “ Plusieurs tâches (threads) s'exécutent en parallèle
- “ C'est un modèle de programmation multitâches sur architecture à mémoire partagée.

Mesure de performances (1/1)

- L'accélération et l'efficacité sont deux métriques de performance parallèle
- **Accélération (Speedup):** Elle calcule l'évolution du temps d'exécution en fonction du nombre de processeur

$$S_p = \frac{T_s}{T_p}$$

Où

- “ T_s : Temps d'exécution du meilleur algorithme séquentiel
- “ T_p : temps d'exécution de l'algorithme parallèle sur p threads (p processeurs)

- **Efficacité :** elle mesure le taux d'occupation « utile » des processeurs

$$E_p = \frac{S_p}{p} = \frac{T_s}{p \cdot T_p}$$

Bénéfices de la parallélisation (1/1)

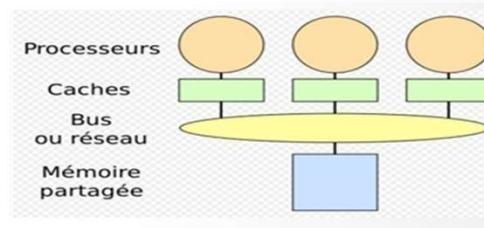
- Exécution plus rapide du programme (Généralement : gain en temps de restitution) en distribuant le travail sur différents cœurs de calcul.
- Résolution de problèmes avec un nombre de degré de libertés très élevé sur le domaine global (plus de ressources matérielles accessibles, notamment la mémoire)
- **Remarques importantes :**
 - ✓ Bien optimiser le code séquentiel avant de se lancer dans la parallélisation
 - ✓ Paralléliser un code séquentiel ne consiste pas à réécrire le code de calcul :
 - Ne pas dénaturer les algorithmes de calcul en termes de stabilité et de convergence
 - Garder le corps des instructions de calcul du code séquentiel
 - Modifier les boucles de calcul qui parcouruent cette fois des données locales de tableaux
 - De façon judicieuse placer les instructions de parallélisation au bon endroit pour que lorsqu'elles sont omises, on retrouve les instructions de calcul du code séquentiel
 - ✓ Bien s'assurer que le code parallèle sur plusieurs cœurs de calcul reproduise des résultats « identiques » ou le plus proches possible que ceux obtenus sur 1 cœur

OpenMP (1/1)

- OpenMP est un standard défini par OpenMP Architecture Review Board en 1998, définissant une interface de **programmation (API) parallèle**, pour les systèmes **multi-threads** à **mémoire partagée** disponible dans les langages C, C++ et Fortran, basé sur l'ajout de **directives** dans le code source
- OpenMP est implémenté dans les principaux compilateurs du marché (gcc, llvm, icc, visual studio, ...) en ajoutant une option de compilation
- **Modèle de programmation OpenMP**
 - ✓ C'est un modèle de programmation multitâches sur architecture à mémoire partagée
 - ✓ Plusieurs threads s'exécutent en parallèle
 - ✓ La mémoire est partagée (physiquement ou virtuellement)
 - ✓ Les communications entre tâches se font (Généralement) par lectures et écritures dans la mémoire partagée
 - ✓ Les processeurs multi-cœurs partagent une mémoire commune
 - ✓ Les tâches peuvent être attribuées à des cœurs distincts

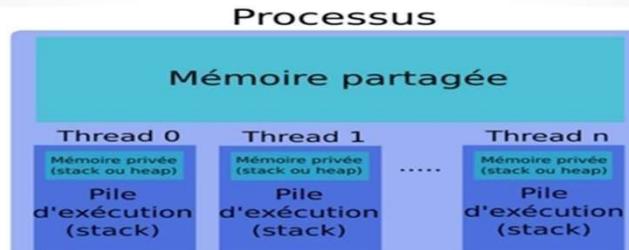
Mémoire partagée (1/1)

- ✓ Une même zone de la mémoire vive est accédée par plusieurs processus. C'est le comportement de la mémoire de threads issus d'un même processus
- ✓ Utilisation de la bibliothèque OpenMP (Open Multi-Processing) => En C/C++ :
`#include <omp.h>`



- ✓ **Symmetric Shared-memory Multiprocessors (SMP)** : architecture parallèle qui consiste à multiplier les processeurs identiques au sein d'un nœud, de manière à augmenter la puissance de calcul avec une mémoire unique partagée

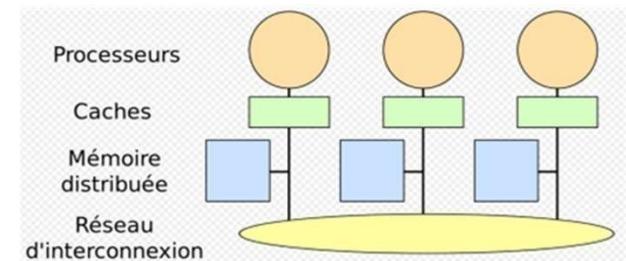
Schéma de principe OpenMP



Mémoire distribuée (1/1)

➤ Mémoire distribuée

- ✓ La mémoire d'un système informatique multiprocesseur est dite distribuée lorsque la mémoire est répartie en plusieurs nœuds, chaque portion n'étant accessible qu'à certains processeurs
- ✓ Un réseau de communication relie les différents nœuds



- ✓ Un système NUMA (Non Uniform Memory Access ou Non Uniform Memory Architecture, signifiant respectivement accès mémoire non uniforme et architecture mémoire non uniforme) est un système multiprocesseur dans lequel les zones mémoire sont séparées et placées en différents endroits
- ✓ Le même programme est exécuté par tous les processus selon le modèle SMPD (Single Program Multiple Data)
- ✓ Bonne règle : un seul processus par cœur
- ✓ Toutes les variables du programme sont privées et résident dans la mémoire locale allouée à chaque processus
- ✓ Des données sont échangées entre plusieurs processus via des appels utilisant la bibliothèque **MPI (Message Passing Interface)**

OpenMP - OpenMPI (1/1)

➤ Différence OpenMP – MPI

- OpenMP

- ✓ OpenMP (Open Multi- Processing) est utilisé sur des machines multiprocesseurs à mémoire partagée SMP \$ hared Memory Programming)
- ✓ OpenMP n'est utilisable que sur un seul nœuds de calcul

- MPI

- ✓ MPI (Message Passing Interface) est utilisé sur des machines multiprocesseurs à mémoire distribuée ou partagée DMP (Distributed Memory Programming), c-à-d que la mémoire est répartie sur plusieurs nœuds reliés par un réseau de communication
- ✓ L'échange de données se fait par "passage de message"

Utilité de OpenMP (1/1)

➤ **Avantage de la programmation OpenMP**

- ✓ Simplicité de mise en œuvre et facilité de programmation
- ✓ La parallélisation ne dénature pas le code ; une seule version du code à gérer pour la version séquentielle et parallèle
- ✓ On peut gagner en temps de calcul avec quelques directives OpenMP très faciles à implémenter
- ✓ Gestion de « threads » transparente et portable

➤ **Inconvénients de la programmation OpenMP**

- ✓ Problème de localité des données
- ✓ Mémoire partagée mais non hiérarchique
- ✓ Les surcoûts dus au partage du travail et à la création/gestion des threads peuvent se révéler importants, particulièrement lorsque la granularité du code parallèle est petite/faible (boucles de petites tailles de tableaux)
- ✓ Passage à l'échelle limité (efficacité sur plusieurs cœurs), parallélisme modéré : dans la pratique, on observe une extensibilité limitée des codes (en tout cas inférieure à celle du même code parallélisé avec MPI), même lorsqu'ils sont bien optimisés. Plus la granularité du code est fine, plus ce phénomène s'accentue

Environnement OpenMP (1/2)

➤ Préparation de l'environnement OpenMP

✓ Installer mingw64 : <https://sourceforge.net/projects/mingw-w64/>

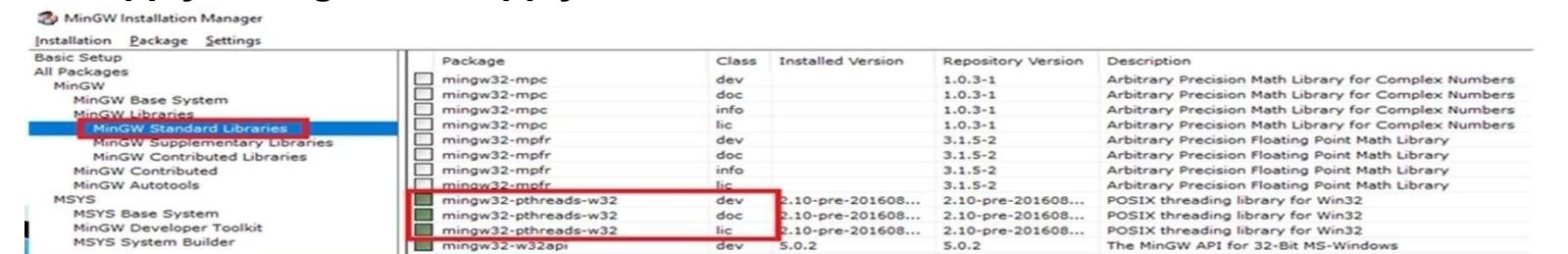
✓ Paramétrage de mingw64 :

- ◆ Lors de l'installation, elle s'affiche une boîte de dialogue de paramétrage, on marque (clic droit puis choix de "Mark for installation") le compilateur gcc/g++ (gcc pour C et g++ pour C++) à travers "Basic steup" «



Package	Class	Installed Version	Repository Version	Description
mingw-developer-tool...	bin		2013072300	An MSYS Installation for MinGW Developers (meta)
mingw32-base	bin		2013072200	A Basic MinGW Installation
mingw32-gcc-ada	bin		6.3.0-1	The GNU Ada Compiler
mingw32-nrc-fortran	bin		6.3.0-1	The GNU FORTRAN Compiler
mingw32-gcc-g++	bin		6.3.0-1	The GNU C++ Compiler
mingw32-gcc-objc	bin		6.3.0-1	The GNU Objective-C Compiler
msys-base	bin		2013072300	A Basic MSYS Installation (meta)

- ◆ A partir de "All Packages → MinGW Standard Libraries" on marque les trois options de "Pthreads" pour l'installation (mingw32-pthreads-w32 : dev, doc et lic) puis on choisit "Apply Changes → Apply "



Package	Class	Installed Version	Repository Version	Description
mingw32-mpc	dev	1.0.3-1	1.0.3-1	Arbitrary Precision Math Library for Complex Numbers
mingw32-mpc	doc	1.0.3-1	1.0.3-1	Arbitrary Precision Math Library for Complex Numbers
mingw32-mpc	info	1.0.3-1	1.0.3-1	Arbitrary Precision Math Library for Complex Numbers
mingw32-mpc	lic	1.0.3-1	1.0.3-1	Arbitrary Precision Math Library for Complex Numbers
mingw32-mpfr	dev	3.1.5-2	3.1.5-2	Arbitrary Precision Floating Point Math Library
mingw32-mpfr	doc	3.1.5-2	3.1.5-2	Arbitrary Precision Floating Point Math Library
mingw32-mpfr	info	3.1.5-2	3.1.5-2	Arbitrary Precision Floating Point Math Library
mingw32-mpfr	lic	3.1.5-2	3.1.5-2	Arbitrary Precision Floating Point Math Library
mingw32-pthreads-w32	dev	2.10-pre-201608...	2.10-pre-201608...	POSIX threading library for Win32
mingw32-pthreads-w32	doc	2.10-pre-201608...	2.10-pre-201608...	POSIX threading library for Win32
mingw32-pthreads-w32	lic	2.10-pre-201608...	2.10-pre-201608...	POSIX threading library for Win32
mingw32-w32api	dev	5.0.2	5.0.2	The MinGW API for 32-Bit MS-Windows

Environnement OpenMP (2/2)

- ✓ **Installer msys64** : <https://www.msys2.org/>
 - ✓ **Paramétrage de msys64** : (En se servant du terminal de msys64)
 - ◆ Installer emacs : Taper la commande **pacman -S emacs**
 - ◆ Pour pouvoir compiler et utiliser les ressources de développement, on installe quelques paquets de base : **pacman -S gitbase-develmingw-w64-i686-gtk3 mingw-w64-x86_64-gtk3 mingw-w64-i686-gcc mingw-w64-x86_64-gcc**
 - ◆ Mettre à jour le système de paquets : **pacman -Syu**
 - ✓ Vérifier sur le terminal de **msys64** si le compilateur **gcc** est installé : on tape **gcc --version**
- 💡 Si **gcc** n'est pas installé, on tape **pacman -S gcc** pour l'installer

Structure d'un programme OpenMP (1/2)

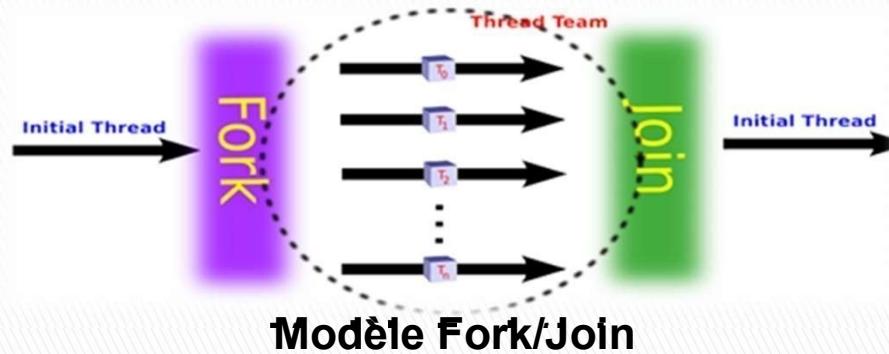
➤ **Structure d'un programme OpenMP**

- ◆ OpenMP est une librairie de programmation parallèle en SMP (Symetric Multi-Processors, or Shared Memory Processors). En effet, tous les threads partagent la mémoire et les données
- ◆ Toutes les fonctions OpenMP sont incluses dans la bibliothèque <omp.h>
- ◆ Un programme OpenMP peut contenir des sections qui sont séquentielles, d'autres qui sont parallèles. Généralement, il commence par une section séquentielle (File header)
- ◆ Lors de l'exécution, un seul thread est utilisé dans le bloc séquentiel, et plusieurs threads sont utilisés dans les sections parallèles
- ◆ Il y a un seul thread qui s'exécute tout le long du programme ; c'est le thread maître (master thread). Les sections parallèles provoquent la génération de nouveaux threads dits esclaves
- ◆ Pour paralléliser un bloc d'instructions, on utilise la directive **#pragma omp parallel**

Structure d'un programme OpenMP (2/2)

➤ Modèle Fork-Join

- ❖ **Fork** : c'est le début du bloc parallèle ; les tâches sont affectées aux threads
- ❖ **Join** : c'est la fin du bloc parallèle ; on joint les résultats élémentaires pour former le résultat final



Compilation – Exécution OpenMP (1/1)

➤ Compilation en OpenMP

- ◆ Soit **hello.c** un programme OpenMP qui affiche le message "**Hello !**" par divers threads. Pour l'exécuter, il suffit de lancer **ucrt64.exe** de **msys64** et taper : **gcc –fopenmp hello.c -o hello**

◆ **hello** est le .exe à exécuter pour générer le résultat attendu

➤ Exécution en OpenMP

- ◆ Revenons à l'exemple précédent, pour exécuter le fichier **hello.exe**, il suffit de taper **./hello**
- ◆ Un programme OpenMP est exécuté par un processus unique (sur un ou plusieurs cœurs)
- ◆ Les threads accèdent aux mêmes ressources que le processus
- ◆ Le gestionnaire de tâches du système d'exploitation affecte les tâches aux cœurs

OpenMP en pratique (1/25)

➤ Créer une zone parallèle

- ◆ Le bloc parallèle est exécuté par un nombre de threads égal – par défaut – au nombre de "cores" du processeur puisque nous n'avons pas défini le nombre de threads.
- ◆ Si un ordinateur contient 4 cores, le message **hello** sera affiché 4 fois

```
1 #include <stdio.h>
2
3 int main()
4 {
5     #pragma omp parallel
6     printf("Hello !\n") ;
7     return 0 ;
8 }
```

- ◆ Si on veut afficher l'identifiant de chaque thread (c'est un indice entier dont le premier thread admet 0 comme indexe) qui a affiché chaque message, on utilise la fonction **omp_get_thread_num()** de la bibliothèque **<omp.h>**
- ⌚ La fonction **omp_get_num_threads()** renvoie le nombre total de threads qui vont participer à l'exécution de la région parallèle

OpenMP en pratique (2/25)

```
1 #include <stdio.h>
2 #include<omp.h>
3 int main()
4 {
5     int id, nb ;
6     #pragma omp parallel
7     {
8         id = omp_get_thread_num();
9         nb = omp_get_num_threads();
10        printf("Je suis le thread numéro : %d parmi %d threads \n", id, nb) ;
11    }
12    return 0 ;
13 }
```

- 💡 Dans ce cas, puisque l'on n'a pas défini le nombre de threads de la région parallèle (par défaut, il est égal au nombre de cœurs), et pour des raisons d'optimisations, on va faire sortir l'instruction nb = `omp_get_num_threads()` de la région parallèle, mais dans ce cas on aura nb = 1. Donc on fait recours à utiliser la fonction = `omp_get_num_procs()` qui renvoie le nombre de cores du processeurs disponibles à être exploités .

```
1 #include <stdio.h>
2 #include<omp.h>
3 int main()
4 {
5     int id, nb ;
6     nb = omp_get_num_procs();
7     #pragma omp parallel
8     {
9         id = omp_get_thread_num();
10        printf("Je suis le thread numéro : %d parmi %d threads \n", id, nb) ;
11    }
12    return 0 ;
13 }
```

OpenMP en pratique (3/25)

➤ Variables privées et partagées

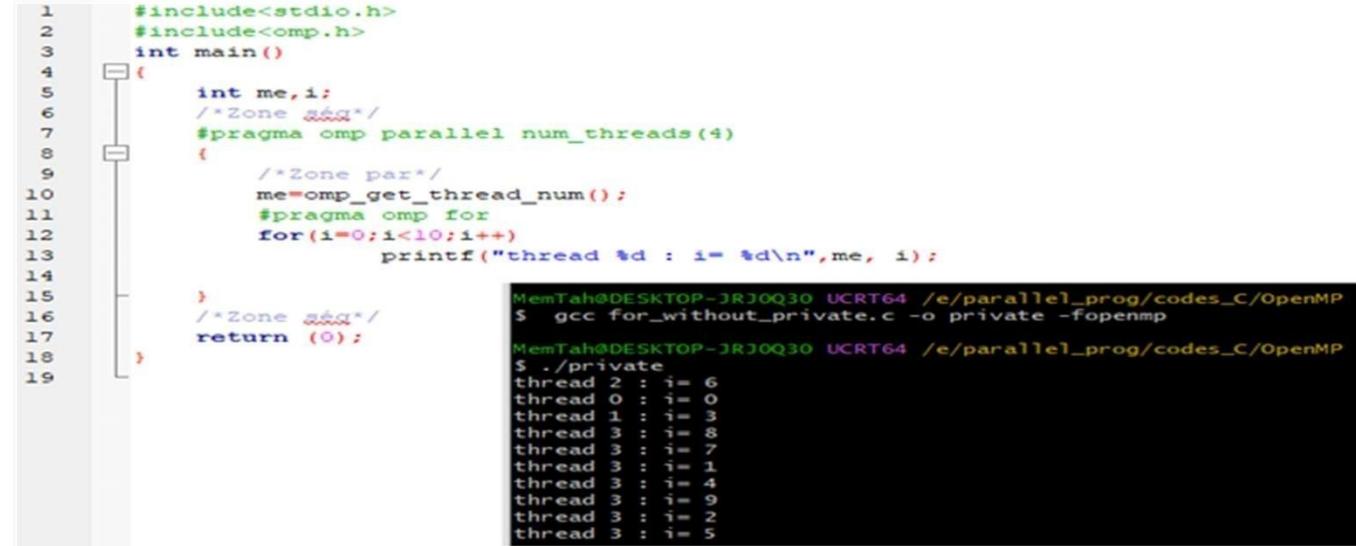
✓ Clauses private et shared

Dans une section parallèle, une variable peut être privée ou partagée :

- ◆ **private** : chaque thread possède sa propre copy (valeur) de la variable.
Une variable privée n'est pas initialisée et sa valeur n'est pas maintenue hors de la section parallèle
- ◆ Par défaut, les compteurs des boucles sont privés
- ◆ **shared**: une variable partagée est visible et accessible simultanément par tous les threads
- ◆ A l'exception des compteurs des boucles, toutes les autres variables d'une zone partagée sont aussi partagées
- ◆ Le type d'une variable (privée ou partagée) est indiqué après la directive qui parallélise un bloc

OpenMP en pratique (4/25)

- ◆ Soit l'exemple suivant dans lequel on a parallélisé la boucle **for** dans une section parallèle



```
1 #include<stdio.h>
2 #include<omp.h>
3 int main()
4 {
5     int me,i;
6     /*Zone séq*/
7     #pragma omp parallel num_threads(4)
8     {
9         /*Zone par*/
10        me=omp_get_thread_num();
11        #pragma omp for
12        for(i=0;i<10;i++)
13            printf("thread %d : i= %d\n",me, i);
14
15    }
16    /*Zone séq*/
17    return (0);
18 }
```

MemTah@DESKTOP-JRJ0Q30 UCRT64 /e/parallel_prog/codes_C/OpenMP
\$ gcc for_without_private.c -o private -fopenmp
MemTah@DESKTOP-JRJ0Q30 UCRT64 /e/parallel_prog/codes_C/OpenMP
\$./private
thread 2 : i= 6
thread 0 : i= 0
thread 1 : i= 3
thread 3 : i= 8
thread 3 : i= 7
thread 3 : i= 1
thread 3 : i= 4
thread 3 : i= 9
thread 3 : i= 2
thread 3 : i= 5

- ◆ Dans ce cas, les instructions de la boucle **for** seront réparties (partagées avec) sur les threads. Mais le problème est que l'on peut avoir un affichage incohérent et incompatible avec l'exécution réelle dans le processeur. Il s'avère qu'un thread ne s'est pas exécuté ou bien qu'il y en a une répartition non équilibrée des tâches sur les threads, mais en réalité c'est un jeu d'affichage dû au concurrence des threads. La solution est donc de définir la variable (me) comme **private**

OpenMP en pratique (5/25)

```
1 #include<stdio.h>
2 #include<omp.h>
3 int main()
4 {
5     int me,i;
6     /*Zone séq*/
7     #pragma omp parallel num_threads(4) private (me)
8     {
9         /*Zone par*/
10        me=omp_get_thread_num();
11        #pragma omp for
12        for(i=0;i<10;i++)
13            printf("thread %d : i= %d\n",me, i);
14    }
15    /*Zone séq*/
16    return (0);
17 }
18
19
```

```
MemTah@DESKTOP-JRJ0Q30 UCRT64 /e/parallel_prog/codes_C/OpenMP
$ gcc for_with_private.c -o private -fopenmp

MemTah@DESKTOP-JRJ0Q30 UCRT64 /e/parallel_prog/codes_C/OpenMP
$ ./private
thread 2 : i= 6
thread 1 : i= 3
thread 0 : i= 0
thread 3 : i= 8
thread 2 : i= 7
thread 1 : i= 4
thread 0 : i= 1
thread 3 : i= 9
thread 1 : i= 5
thread 0 : i= 2
```

OpenMP en pratique (6/25)

✓ Clause firstprivate

- ◆ Quand une variable est déclarée privée sa valeur n'est pas définie à l'entrée d'une région parallèle.
- ◆ Pour garder la dernière valeur affectée à une variable avant d'entrer dans la région parallèle il faut utiliser la clause : **firstprivate**
- ◆ Lorsqu'une variable privée est initialisée en dehors d'une région parallèle, sa valeur n'est pas prise en compte dans la région parallèle.
- ◆ La clause **firstprivate(a)** permet d'avoir la dernière valeur de la variable a avant d'entrer dans la région parallèle. Cette variable sera privée

```
//# include <iostream>
#include <stdio.h>
#include "omp.h"
//using namespace std;
int main ()
{
    int x =10;
    # pragma omp parallel firstprivate (x)
    {
        printf ("x=%d\n", x) ;
    }
}

//# include <iostream>
#include <stdio.h>
#include "omp.h"
//using namespace std;
int main ()
{
    int x =10;
    # pragma omp parallel private (x)
    {
        printf ("x=%d\n", x) ;
    }
}
```

\$./fp
x=10
x=10
x=10
x=10

\$./priv
x=7
x=0
x=0
x=7

OpenMP en pratique (7/25)

✓ Clause lastprivate

- La clause **lastprivate()** permet de donner la dernière valeur d'une variable dans une région parallèle, elle n'est utilisable qu'avec les directives : **for** et **section**

```
1 //# include <iostream>
2 # include <stdio.h>
3 # include "omp.h"
4 //using namespace std ;
5
6 int main ()
7 {
8     int i, a = 33 ;
9     # pragma omp parallel num_threads(4)
10    {
11        # pragma omp single
12        {
13            printf (" avant le for , a=%d \n",a) ;
14        }
15        # pragma omp for lastprivate (a)
16        for (i =0; i <10; i++)
17        {
18            a=i;
19        }
20    }
21    printf (" apres le for , a=%d\n",a);
22 }
```

s ./lp
avant le for , a=33
apres le for , a=9

- Dans l'exemple ci-dessus, sans la clause **lastprivate(a)** , la valeur de la variable **a** après la sortie de la boucle **for** change à chaque exécution du programme. Cette clause permet donc d'envoyer la valeur finale d'une variable calculée dans la boucle

OpenMP en pratique (8/25)

✓ Clause default

- ◆ Par défaut toutes les variables sont partagées. Le statut par défaut des variables peut être changé avec la clause **default()**
- ◆ Cette clause prend comme argument **shared** ou **none** en C/C++
- ◆ **default(None)** permet à l'utilisateur de spécifier le statut de toutes les variables utilisées dans la région parallèle

```
1 //# include <iostream>
2 #include <stdio.h>
3 #include "omp.h"
4 //using namespace std;
5
6 int main ()
7 {
8     int x=1 , y =10;
9     int num_thread ;
10
11    printf(" Dans la region sequentielle x= %d y= %d \n\n",x, y);
12    printf(" Dans le region parallele :\n" );
13
14    # pragma omp parallel num_threads(4) default ( none ) private (num_thread ,x,y)
15    {
16        num_thread = omp_get_thread_num () ;
17        x= 2 + num_thread ;
18        y= 2 + num_thread ;
19        printf ("      avec le thread %d x=%d y=%d \n", num_thread , x, y) ;
20    }
21
22    printf(" \nDans la region sequentielle x= %d y= %d \n" ,x,y);
23
24    return 0;
25 }
```

```
$ ./def
Dans la region sequentielle x= 1 y= 10
Dans le region parallele :
      avec le thread 0 x=2 y=2
      avec le thread 2 x=4 y=4
      avec le thread 1 x=3 y=3
      avec le thread 3 x=5 y=5
Dans la region sequentielle x= 1 y= 10
```

OpenMP en pratique (9/25)

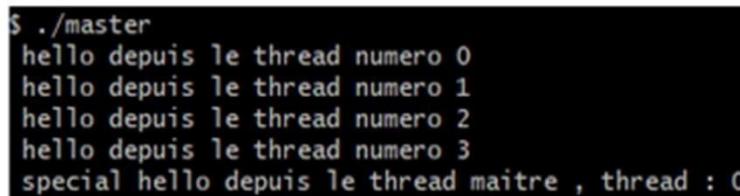
➤ Exécution exclusive

- Dans une région parallèle il est possible d'exécuter une partie du code avec un seul thread avec les directives master ou single

✓ Directive master

- ◆ Le code est exécuté avec le thread maître : thread numéro 0

```
1 # include <stdio.h>
2 # include "omp.h"
3 //using namespace std;
4 int main ()
5 {
6     int thread_num ;
7     # pragma omp parallel private ( thread_num )
8     {
9         thread_num = omp_get_thread_num () ;
10        printf (" hello depuis le thread numero %d \n", thread_num ) ;
11        # pragma omp master
12        {
13            printf (" special hello depuis le thread maître , thread : %d \n",thread_num ) ;
14        }
15    }
16    return 0;
17 }
```

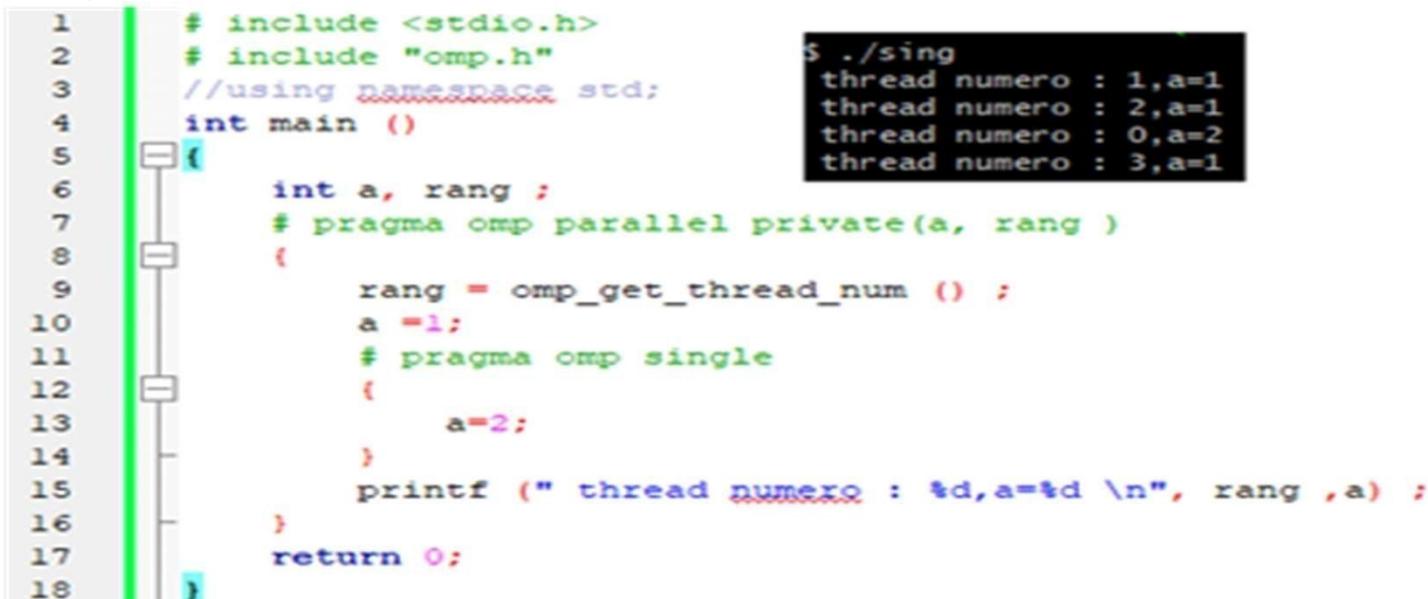


```
$ ./master
hello depuis le thread numero 0
hello depuis le thread numero 1
hello depuis le thread numero 2
hello depuis le thread numero 3
special hello depuis le thread maître , thread : 0
```

OpenMP en pratique (10/25)

✓ Directive single

- ◆ Le code est exécuté par le premier thread arrivé à la directive single.
- ◆ Le code poursuit son exécution uniquement lorsque tous les threads arrivent à la fin de la directive single



```
1 # include <stdio.h>
2 # include "omp.h"
3 //using namespace std;
4 int main ()
5 {
6     int a, rang ;
7     # pragma omp parallel private(a, rang )
8     {
9         rang = omp_get_thread_num () ;
10        a =1;
11        # pragma omp single
12        {
13            a=2;
14        }
15        printf (" thread numero : %d,a=%d \n", rang ,a) ;
16    }
17    return 0;
18 }
```

- ◆ Dans l'exemple ci-dessus, la valeur de a change pour un seul thread seulement, pour que cette valeur soit transmise à tous les autres threads il faut utiliser la clause **copyprivate**

OpenMP en pratique (11/25)

✓ Clause copyprivate()

- ◆ Cette clause n'est utilisable qu'à la fin de la directive single.
- ◆ Elle permet de transmettre à tous les threads la nouvelle valeur d'une variable privée affectée par un seul thread

```
1  # include <stdio.h>
2  # include "omp.h"
3  //using namespace std;
4  int main ()
5  {
6      int a, rang ;
7      # pragma omp parallel private(a, rang )
8      {
9          rang = omp_get_thread_num () ;
10         a =1;
11         # pragma omp single copyprivate (a)
12         {
13             a=2;
14         }
15         printf (" thread numero : %d,a=%d \n", rang ,a) ;
16     }
17     return 0;
18 }
```

\$./cop
thread numero : 2,a=2
thread numero : 1,a=2
thread numero : 0,a=2
thread numero : 3,a=2

OpenMP en pratique (12/25)

➤ for parallèle

- ◆ Pour paralléliser une boucle **for**, on utilise la directive **#pragma omp for**.
- ◆ Dans ce cas, on cherche normalement à répartir les instructions de la boucle sur les threads pour que chacun exécute un bloc (morceau) et renvoie un résultat élémentaire

```
1 //compute the sum of two arrays in parallel
2 #include <stdio.h>
3 #include <omp.h>
4 #define N 1000
5 int main(void) {
6     float a[N], b[N], c[N];
7     int i;
8
9     /* Initialize arrays a and b */
10    for (i = 0; i < N; i++) {
11        a[i] = i * 2.0;
12        b[i] = i * 3.0;
13    }
14
15    /* Compute values of array c = a+b in parallel. */
16    #pragma omp parallel shared(a, b, c) private(i)
17    {
18        #pragma omp for
19        for (i = 0; i < N; i++) {
20            c[i] = a[i] + b[i];
21            //printf ("%0.2f \n", c[i]);
22        }
23    }
24    // Print the result
25    for (i = 0; i < N; i++) {
26        printf ("%0.2f \n", c[i]);
27    }
28 }
```

```
M /e/parallel_prog/codes_C/OpenMP
MemTah@DESKTOP-JRJ0Q30 UCRT64 /e/parallel_prog/codes_C/OpenMP
$ gcc sum_2_vect.c -o sum -fopenmp
MemTah@DESKTOP-JRJ0Q30 UCRT64 /e/parallel_prog/codes_C/OpenMP
$ ./sum
0.00
5.00
10.00
15.00
20.00
25.00
30.00
35.00
40.00
45.00
50.00
55.00
60.00
65.00
70.00
75.00
80.00
85.00
90.00
95.00
100.00
```

OpenMP en pratique (13/25)

➤ Clause schedule

- Avec **schedule** , on peut fixer le nombre d'instructions d'un bloc parallélisé qui seront affecté à chaque thread (Le cas d'une boucle **loop** par exemple)

✓ Schedule(dynamic, N)

- ◆ OpenMP attribue une itération à chaque thread.
- ◆ Lorsque le thread termine l'exécution, il se verra attribuer la prochaine itération qui n'a pas encore été exécutée
- ◆ Le partage dynamique fonctionne sur la base du " **premier arrivé, premier servi** ".
- ◆ Deux exécutions avec le même nombre de threads pourraient (et produiraient très probablement) des répartitions ("espace d'itération" ou "threads") complètement différents, comme on peut facilement le vérifier

```
1 #include<stdio.h>
2 #include<omp.h>
3 int main()
4 {
5     int me,i;
6     /*Zone séq*/
7     #pragma omp parallel num_threads(4) private(me,i)
8     {
9         /*Zone par*/
10        me=omp_get_thread_num();
11        #pragma omp for schedule(dynamic,2) //Un bloc de 2 pour chaque thread à l'exécution
12        //#pragma omp for schedule(static,2)
13        for(i=0;i<10;i++)
14            printf("thread %d : i= %d\n",me, i);
15
16    }
17    /*Zone séq*/
18    return (0);
19 }
```

```
$ ./private
thread 3 : i= 6
thread 0 : i= 2
thread 1 : i= 4
thread 2 : i= 0
thread 3 : i= 7
thread 0 : i= 3
thread 1 : i= 5
thread 2 : i= 1
thread 3 : i= 8
thread 3 : i= 9
```

OpenMP en pratique (14/25)

✓ Schedule(static, N)

- ◆ Par défaut, OpenMP attribue statiquement des itérations de boucle aux threads.
- ◆ Lorsque le bloc parallèle `for` entré, il affecte à chaque thread l'ensemble d'itérations de boucle qu'il doit exécuter
- ◆ Le partage statique signifie que les blocs d'itérations sont répartis statiquement sur les threads d'exécution de manière circulaire (Round Robin).
- ◆ La bonne chose est que l'exécution OpenMP garantit – s'il y en a deux boucles distinctes avec le même nombre d'itérations – et que l'on procède à les exécuter avec le même nombre de threads en utilisant la partage statique, que chaque thread recevra exactement la même plage d'itérations dans les deux régions parallèles

```
1 #include<stdio.h>
2 #include<omp.h>
3 int main()
4 {
5     int me,i;
6     /*Zone séq*/
7     #pragma omp parallel num_threads(4) private(me,i)
8     {
9         /*Zone par*/
10        me=omp_get_thread_num();
11        // #pragma omp for schedule(dynamic,2) //Un bloc de 2 pour chaque thread à l'exécution
12        #pragma omp for schedule(static,2)
13        for(i=0;i<10;i++)
14            printf("thread %d : i= %d\n",me, i);
15
16    }
17    /*Zone séq*/
18    return (0);
19 }
```

```
$ ./private
thread 2 : i= 4
thread 1 : i= 2
thread 0 : i= 0
thread 3 : i= 6
thread 2 : i= 5
thread 1 : i= 3
thread 0 : i= 1
thread 3 : i= 7
thread 0 : i= 8
thread 0 : i= 9
```

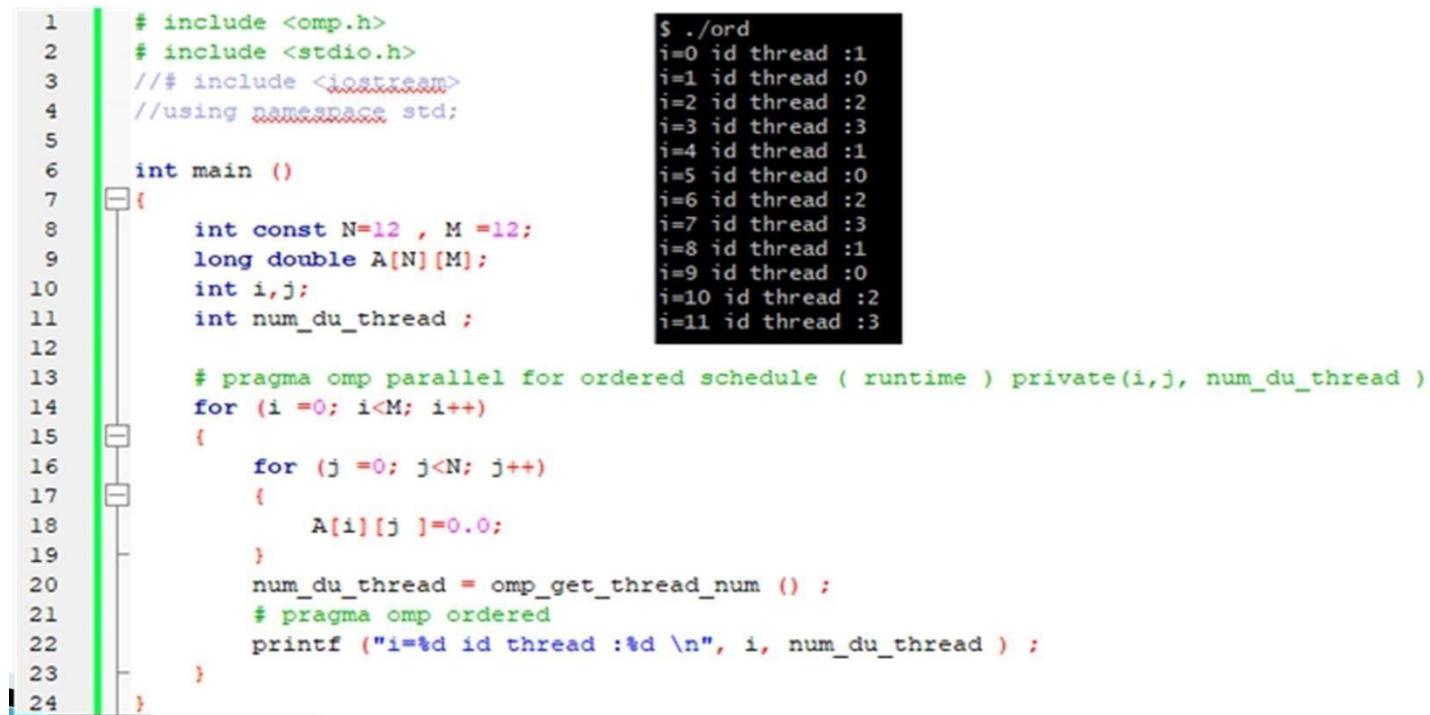
OpenMP en pratique (15/25)

✓ Schedule (guided)

- ◆ Les itérations sont subdivisées en morceaux qui diminuent de tailles successivement jusqu'au atteindre la plus petite taille possible

➤ Clause ordered

- ◆ Très utile pour le débogage, la clause ordered() permet l'exécution de la boucle dans l'ordre des indices



The image shows a terminal window with two panes. The left pane displays a C++ code snippet with line numbers from 1 to 24. The right pane shows the command \$./ord followed by the output of the program, which lists iteration indices i and their corresponding thread IDs.

```
#include <omp.h>
#include <stdio.h>
//# include <iostream>
//using namespace std;

int main ()
{
    int const N=12 , M =12;
    long double A[N][M];
    int i,j;
    int num_du_thread ;

    # pragma omp parallel for ordered schedule ( runtime ) private(i,j, num_du_thread )
    for (i =0; i<M; i++)
    {
        for (j =0; j<N; j++)
        {
            A[i][j]=0.0;
        }
        num_du_thread = omp_get_thread_num () ;
        # pragma omp ordered
        printf ("i=%d id thread :%d \n", i, num_du_thread ) ;
    }
}
```

```
$ ./ord
i=0 id thread :1
i=1 id thread :0
i=2 id thread :2
i=3 id thread :3
i=4 id thread :1
i=5 id thread :0
i=6 id thread :2
i=7 id thread :3
i=8 id thread :1
i=9 id thread :0
i=10 id thread :2
i=11 id thread :3
```

OpenMP en pratique (16/25)

➤ **Synchronisation des threads**

- OpenMP nous permet de spécifier comment synchroniser les threads

- **Règle d'or :**

“ **Tous les accès potentiellement conflictuels (au moins l'un d'entre eux est une écriture) aux variables partagées doivent être protégés (atomic, critical, ...)** ”

OpenMP en pratique (17/25)

✓ barrier

- ◆ Chaque thread attend que tous les autres threads d'une équipe aient atteint ce point
- ◆ On remarque dans l'exemple suivant que l'affichage à l'écran ne se fait pas dans l'ordre, le thread le plus rapide affiche le message en premier.
- ◆ L'ordre de l'affichage est aléatoire à chaque exécution du programme

```
1  #include<stdio.h>
2  #include<omp.h>
3
4  int main (int argc, char *argv[])
5  {
6      int th_id, nthreads;
7
8      #pragma omp parallel private(th_id)
9      {
10         th_id = omp_get_thread_num();
11         # pragma omp single
12         printf("Hello World from thread %d\n", th_id);
13
14         printf(" region parallel : hello depuis le thread numero \n", th_id);
15
16         # pragma omp master
17         {
18             printf("Master : Hello depuis le thread num :\n", th_id);
19         }
20
21     }
22
23 }
```

```
$ ./barrier
Hello World from thread 1
region parallel : hello depuis le thread numero
region parallel : hello depuis le thread numero
region parallel : hello depuis le thread numero
Master : Hello depuis le thread num :
region parallel : hello depuis le thread numero
```

OpenMP en pratique (18/25)

✓ Barrier (suite)

- En ajoutant la directive **#pragma omp barrier** juste avant la directive **#pragma omp master** le programme va s'exécuter dans l'ordre de son écriture

```
1  #include<stdio.h>
2  #include<omp.h>
3
4  int main (int argc, char *argv[])
5  {
6      int th_id, nthreads;
7
8      #pragma omp parallel private(th_id)
9      {
10         th_id = omp_get_thread_num();
11         # pragma omp single
12             printf("Hello World from thread %d\n", th_id);
13
14         printf(" region parallel : hello depuis le thread numero %d \n", th_id);
15
16         #pragma omp barrier
17         # pragma omp master
18         {
19             printf("Master : Hello depuis le thread num :%d\n", th_id);
20         }
21
22     }
23
24 }
```

\$./barrier
Hello World from thread 1
Hello World from thread 2
Hello World from thread 0
Hello World from thread 3
There are 4 threads

✓ ordered

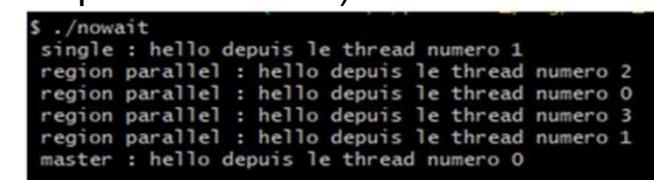
- Le bloc structuré est exécuté dans l'ordre dans lequel les itérations seraient exécutées dans une boucle séquentielle

OpenMP en pratique (19/25)

✓ Clause nowait()

- ◆ Spécifie que les threads qui terminent leurs tâches peuvent continuer sans attendre que tous les threads de l'équipe se terminent. En l'absence de cette clause, les threads rencontrent une barrière de synchronisation
- ◆ Dans certains cas, imposer une barrière implicite peut s'avérer inutile et ralentit un peu le code.
- ◆ Pour éviter cette attente, il faut utiliser la clause **nowait** qui vient directement après la fin d'une directive contenant une barrière de synchronisation implicite (à part la directive **parallel** qui contient une barrière implicite qui ne peut être levée)

```
1 # include <stdio.h>
2 # include <omp.h>
3 //using namespace std;
4 int main ()
5 {
6     # pragma omp parallel
7     {
8         int thread_num = omp_get_thread_num() ;
9         # pragma omp single nowait
10        {
11            printf (" single : hello depuis le thread numero %d \n", thread_num ) ;
12        }
13        printf (" region parallel : hello depuis le thread numero %d \n",thread_num ) ;
14
15        # pragma omp barrier
16        # pragma omp master
17        {
18            printf (" master : hello depuis le thread numero %d \n", thread_num ) ;
19        }
20    }
21 }
```



```
$ ./nowait
single : hello depuis le thread numero 1
region parallel : hello depuis le thread numero 2
region parallel : hello depuis le thread numero 0
region parallel : hello depuis le thread numero 3
region parallel : hello depuis le thread numero 1
master : hello depuis le thread numero 0
```

OpenMP en pratique (20/25)

➤ Opération cumulées sur une variable

✓ Directive reduction

- ◆ Cette clause est utilisée pour calculer une somme, un produit, un maximum... etc. de variables dont la valeur change avec les indices d'une boucle (par exemple) et chaque nouvelle valeur dépend de la valeur précédente.
- ◆ Cette clause prend la forme suivante : **# pragma omp for reduction (operation : variable)**
 - “ Dont :
 - “ **operation**: désigne l'opération de réduction utilisée (résumé dans les tableaux ci-dessous)
 - “ **variable** : désigne la variable sur quoi l'opération est effectuée

Opérations arithmétiques		
	Fortran	C/C++
sommation	+	+
soustraction	-	-
produit	*	*
division	/	non-utilisé

Opérations logiques		
	Fortran	C/C++
et	.and.	&&
ou	.or.	
équivalence	.eqv.	non-utilisé
non-équivalence	.neqv.	non-utilisé

Fonctions intrinsèques		
	Fortran	C/C++
maximum	max	non-utilisé
minimum	min	non-utilisé
et binaire	iand	&
ou inclusif binaire	ior	
ou exclusif binaire	ieor	^

OpenMP en pratique (21/25)

✓ Directive atomic

- ◆ La mise à jour de la mémoire (écriture ou lecture-modification-écriture) dans l'instruction suivante sera effectuée de manière atomique. Cela ne rend pas la déclaration entière atomique; seule la mise à jour de la mémoire est atomique.
- ◆ Un compilateur peut utiliser des instructions matérielles spéciales pour de meilleures performances que lors de l'utilisation de la critique
- ◆ Permet la mise à jour atomique d'un emplacement mémoire.
- ◆ Cette directive ne prend en charge aucune clause et s'applique sur l'instruction qui vient juste après.
- ◆ **atomic** a de meilleures performances que la directive **critical**

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 int main(void) {
5     int count = 0;
6     #pragma omp parallel shared(count)
7     {
8         #pragma omp atomic
9         count++; // count is updated by only a single thread at a time
10    }
11    printf("Number of threads: %d\n", count);
12 }
```

OpenMP en pratique (22/25)

✓ Directive critical

- ◆ **critical** s'applique sur un bloc de code, celui-ci ne sera exécuté que par un seul thread à la fois, une fois qu'un thread a terminé un autre thread peut donc avoir accès à cette région critique
- ◆ Le bloc de code inclus sera exécuté par un seul thread à la fois, et non exécuté simultanément par plusieurs threads.
- ◆ Il est souvent utilisé pour protéger les données partagées des conditions de concurrence
- ⌚ **atomic** et **reduction** sont plus restrictives mais plus performantes que **critical**

```
1 //example1.c: add all elements in an array in parallel
2 #include <stdio.h>
3 #include <omp.h>
4
5 int main() {
6
7     const int N=100;
8     int a[N];
9
10    //initialize
11    for (int i=0; i < N; i++)
12        a[i] = i;
13
14    //compute sum
15    int local_sum, sum;
16    #pragma omp parallel private(local_sum) shared(sum)
17    {
18        local_sum =0;
19
20        //the array is distributed statically between threads
21        #pragma omp for schedule(static,1)
22        for (int i=0; i< N; i++) {
23            local_sum += a[i];
24        }
25        //each thread calculated its LOCAL_SUM. All threads have to add to
26        //the global sum. It is critical that this operation is atomic.
27
28        #pragma omp critical
29        sum += local_sum;
30    }
31    printf("sum=%d should be %d\n", sum, N*(N-1)/2);
32 }
```

OpenMP en pratique (23/25)

➤ Sections parallèles : directive sections

- ◆ Une section parallèle n'est exécutée que par un seul et unique thread (tout comme single).
- ◆ La directive **sections** peut prendre les clauses **lastprivate** et **reduction** et la directive **single** est la seule à prendre la clause **copyprivate**
- ◆ La directive **sections** donne un bloc structuré à chaque thread.
- ◆ La directive **sections** a pour but de répartir l'exécution de plusieurs portions de code indépendantes (dans un bloc parallèle) sur différent threads
- ◆ La directive **sections** permet de découper un travail constitué de parties indépendantes. Une **section** est une portion de code exécutée par un seul thread
- ◆ Chaque partie sera réalisée par un thread qui peut éventuellement créer une région parallèle
- ◆ **SECTIONS** admet **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE**, et **REDUCTION** comme clauses

```
1  #include <stdio.h>
2  #include <omp.h>
3  #define NT 4
4  int main()
5  {
6      int c = 0;
7
8      omp_set_dynamic(0); // Sinon l'instruction suivante ne sera pas à lire
9      omp_set_num_threads(NT);
10
11     #pragma omp parallel
12         #pragma omp sections firstprivate(c)
13         {
14             #pragma omp section
15             {
16                 c++;
17                 printf("Thr%d:c=%d\n", omp_get_thread_num(), c);
18             }
19             #pragma omp section
20             {
21                 c++;
22                 printf("Thr%d:c=%d\n", omp_get_thread_num(), c);
23             }
24         }
25
26     return 0;
}
```

\$./sec
Thr2:c=1
Thr0:c=1

OpenMP en pratique (24/25)

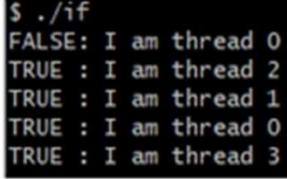
➤ Exécution parallèle conditionnelle

◆ Le **if** peut être utilisé pour désactiver/activer la parallélisation dans certains cas

◆ **Exemple 1 :** Le code suivant sera exécuté si l'environnement OpenMP est installé

```
# if def _OPENMP
    printf("Compiled by an OpenMP-compliant implementation.\n");
# endif
```

◆ **Exemple 2 :** Le premier bloc parallèle sera désactivé et affecté au thread principal(exécuté une seule fois)puisque la variable booléenne admet la valeur **False** (1==0 renvoie false). Alors que le deuxième bloc est exécuté par tous les threads esclaves puisque la valeur booléenne est à **True** (0==0 renvoie True)



```
1 #include <omp.h>
2 #include <stdio.h>
3 int main (void)
4 {
5     int t = (0 == 0); // true value
6     int f = (1 == 0); // false value
7     #pragma omp parallel if (f)
8     {
9         printf ("FALSE: I am thread %d\n", omp_get_thread_num());
10        #pragma omp parallel if (t)
11        {
12             printf ("TRUE : I am thread %d\n", omp_get_thread_num());
13         }
14     }
15 }
```

OpenMP en pratique (25/25)

- Exécution parallèle conditionnelle
- ◆ Exemple 3 : Parcours d'un arbre

Parcours séquentiel :

```
Void parcours()
{
    if(a_gauche())
        a_gauche()->parcours();
    if(a_droite())
        a_droite()->parcours();
    printf("%d",val);
}
```

Parcours parallèle OpenMP :

```
Void parcours()
{
    #pragma omp parallel sections
    {
        #pragma omp section
        if(a_gauche())
            a_gauche()->parcours();
        #pragma omp section
        if(a_droite())
            a_droite()->parcours();
    }
    printf("%d",val);
}
```

Merci pour
votre
attention

