OpenMP

Emmanuel Grolleau
Observatoire de Paris – LESIA – Service d'Informatique Scientifique

Inspiré de :
IDRIS_OpenMP_cours.pdf
Jalel Chergui,Pierre-François Lavallée,Etienne Gondet
Introduction au parallélisme
Gérald Monard

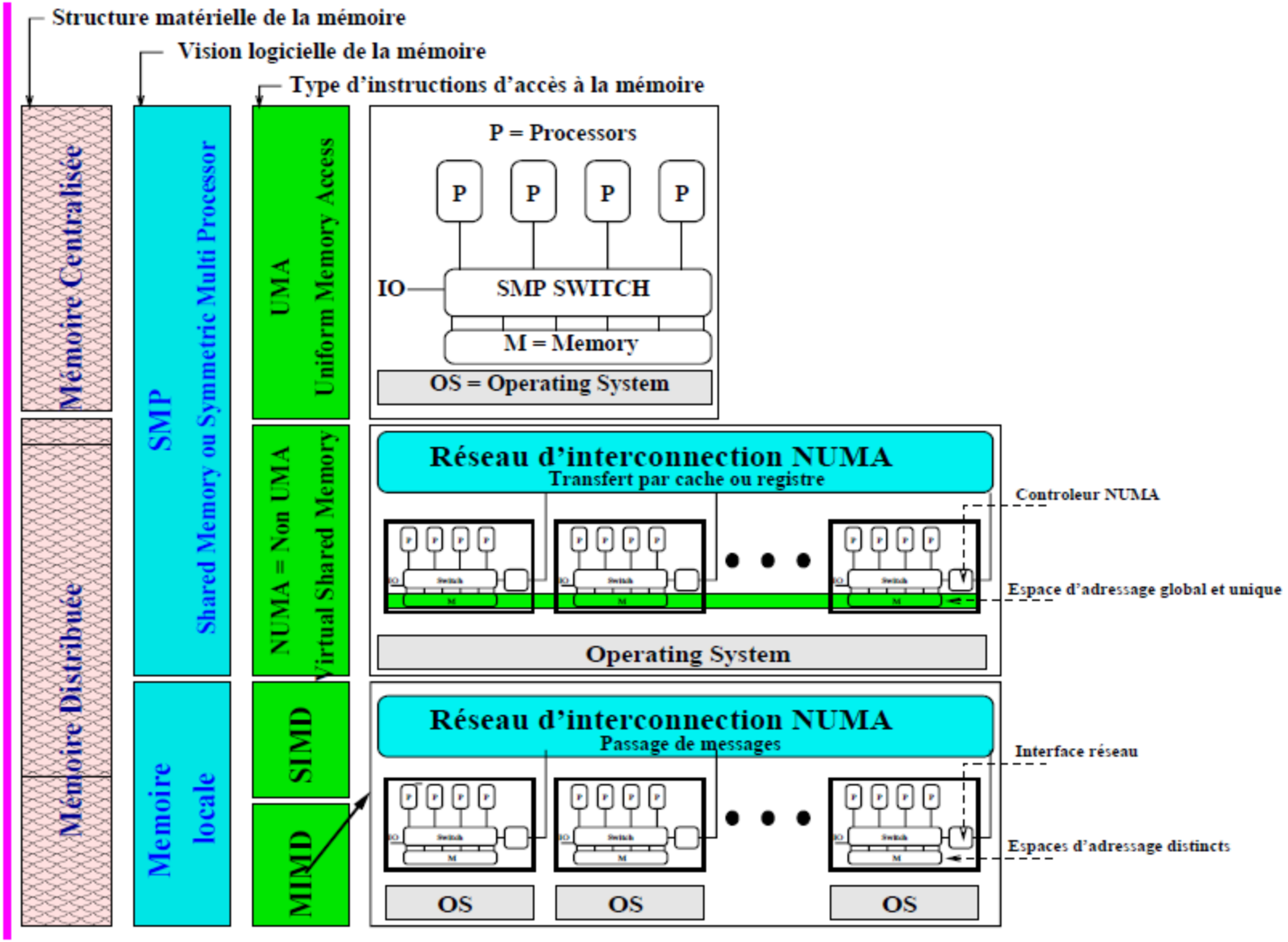
Sommaire

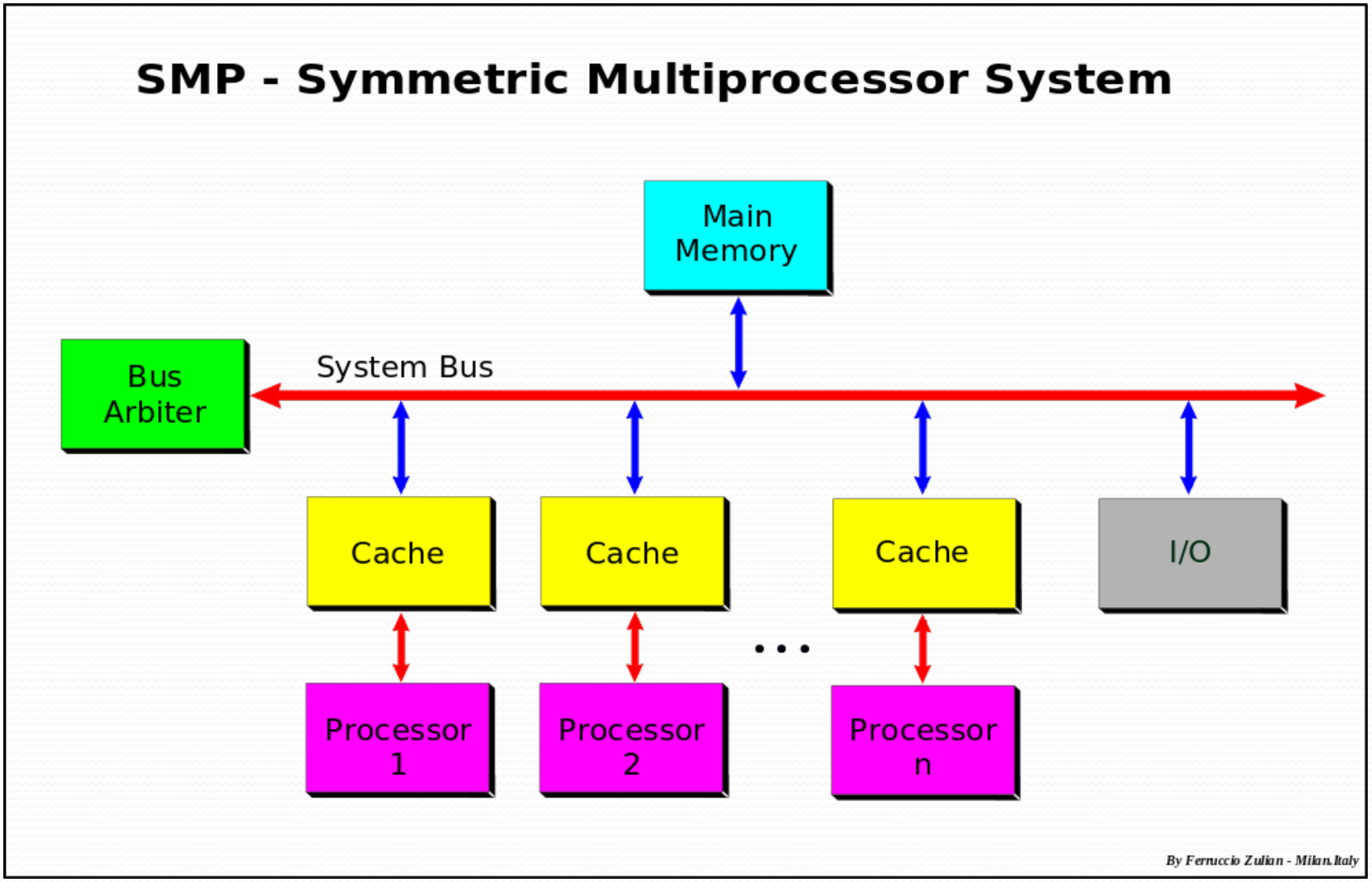
Notions de calcul parallèle
Présentation d'OpenMP
Structure d'un programme
Boucle for
- Schedule Static
- Schedule Dynamic
- Problème de dépendance
- Opération de réduction
- Opération de réduction : Atomic

Portion de code parallèle
- Exécution exclusive
- Single
- Master
- Sections
- Sections Nowait
- Synchronisation Barrier

Appeler du C (OMP) dans Python
Appeler du C (OMP) dans IDL

1.3.2 Les architectures adaptées



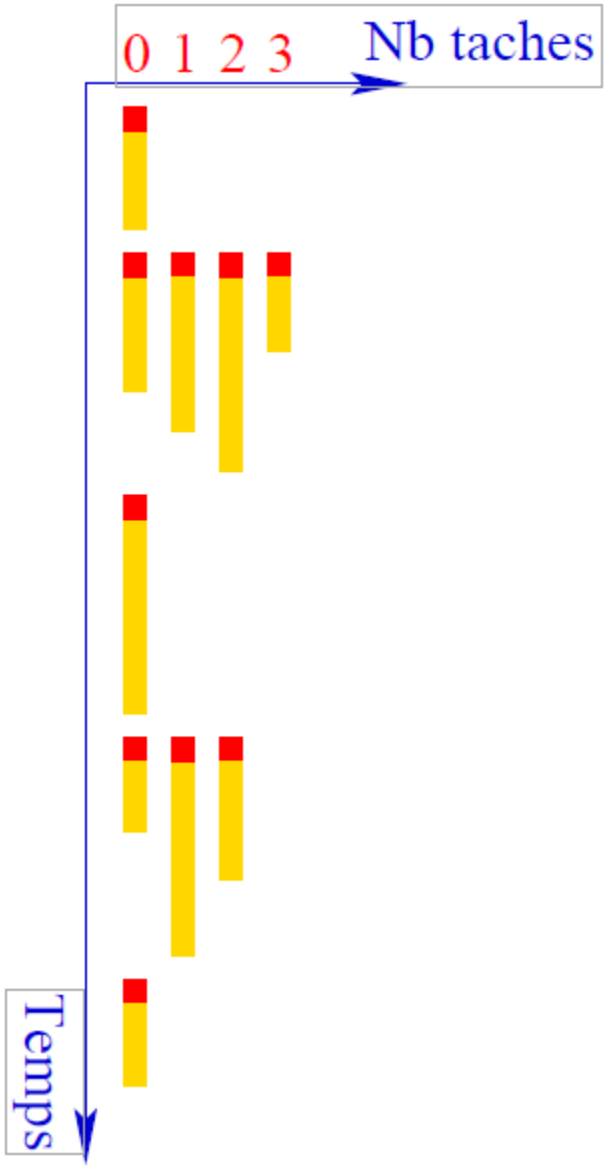

SMP - Symmetric Multiprocessor System
Main Memory
Bus Arbiter
System Bus
Cache
Cache
Cache
I/O
Processor 1
Processor 2
Processor n
By Ferruccio Zulian - Milan Italy

Deux standards : POSIX Threads et OpenMP.

- **POSIX Threads**
  - Basé sur une bibliothèque de routines.
  - Nécessite un codage parallèle explicite
  - Standard IEEE (1995)
  - Langage C seulement
  - Parallélisme très explicite (bas niveau) et oblige le programmeur à être très attentif au détail de programmation.
- **Open MP (**Open Multi-Processing)
  - Basé sur des directives de compilation.
  - Peut utiliser du code séquentiel
  - Supporté par un consortium de vendeurs de compilateurs et de fabriquants de matériels.
  - Portable, multiplateformes (inclus UNIX et Windows)
  - Existe pour C, C++ et Fortran
  - Peut être très simple à manipuler.

Structure d'un programme

- Un programme OpenMP est une alternance de régions séquentielles et de régions parallèles.
- Une région séquentielle est toujours exécutée par la tâche maître, celle dont le rang vaut 0.
- Une région parallèle peut être exécutée par plusieurs tâches à la fois.
- Les tâches peuvent se partager le travail contenu dans la région parallèle.



Gain de la parallélisation

- **Speed-up ou accélération**. => pour un code parallélisé, le rapport entre le temps d'exécution séquentiel et le temps d'exécution parallèle d'une même tâche.

$$\text{Speed-up}(n) = \frac{\text{Temps séquentiel}}{\text{Temps parallèle (sur } n \text{ processeurs)}}$$

Gain de la parallélisation

- **Loi d'Amdahl**

Soit P la fraction parallèlisable d'un programme et S sa fraction non-parallèlisable (séquentielle).

$$\text{speed-up}(n) = \frac{P + S}{\frac{P}{n} + S} = \frac{1}{P/n + S}$$

$P+S = 1$

Gain de la parallélisation

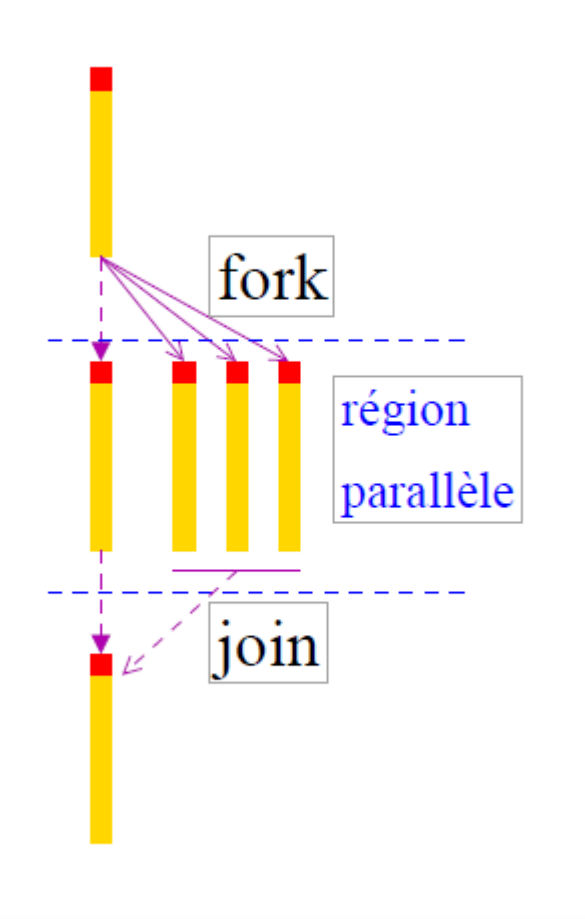
	speed-up		
Nn coeurs	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.99	99.02

- Si 50% du code est parallèlisable (cas classique), on ne pourra avoir un gain meilleur que diviser le temps d'exécution par 2.
- Avec 10 cœurs, on obtiendra déjà un gain de 1.82

Structure d'un programme

- Il est à la charge du développeur d'introduire des directives OpenMP dans son code
- A l'exécution du programme, le système d'exploitation construit une région parallèle sur le modèle « fork and join ».
- A l'entrée d'une région parallèle, la tâche maître crée/active (fork) des processus « fils » (processus légers) qui disparaissent/s'assoupissent en fin de région parallèle (join) pendant que la tâche maître poursuit seule l'exécution du programme jusqu'à l'entrée de la région parallèle suivante.

Structure d'un programme



OpenMP : Concepts généraux

- Un programme OpenMP est exécuté par un processus unique
- Ce processus active des processus légers (threads) à l'entrée de région parallèle
- Chaque thread exécute une tâche composée d'un ensemble d'instruction
- Pendant l'exécution d'une tâche, une variable peut être lue et/ou modifiée en mémoire.

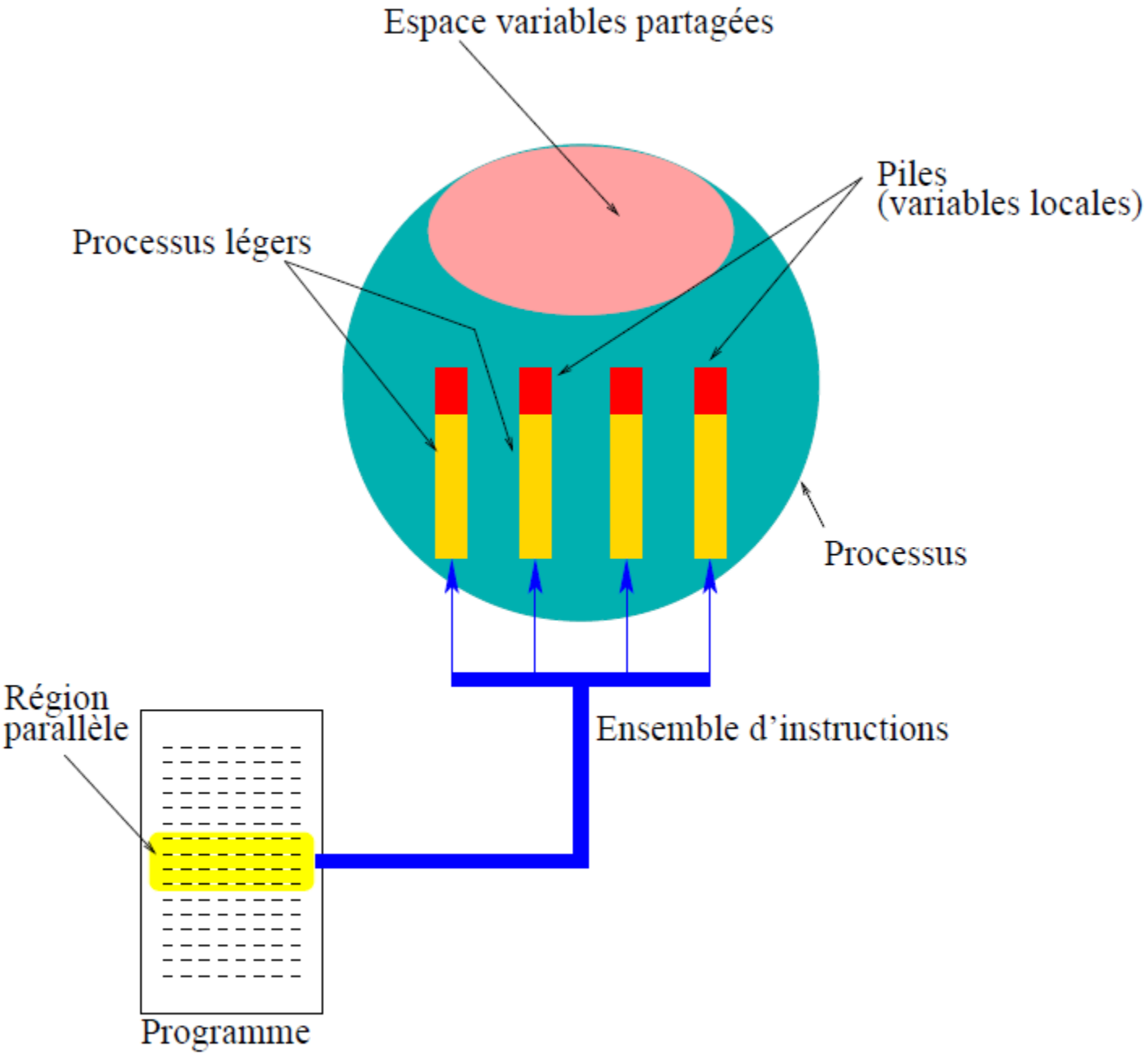
Exemple de directives

```
#include <omp.h>
...

#pragma omp parallel
{
        //Code parallèlisé
}
```

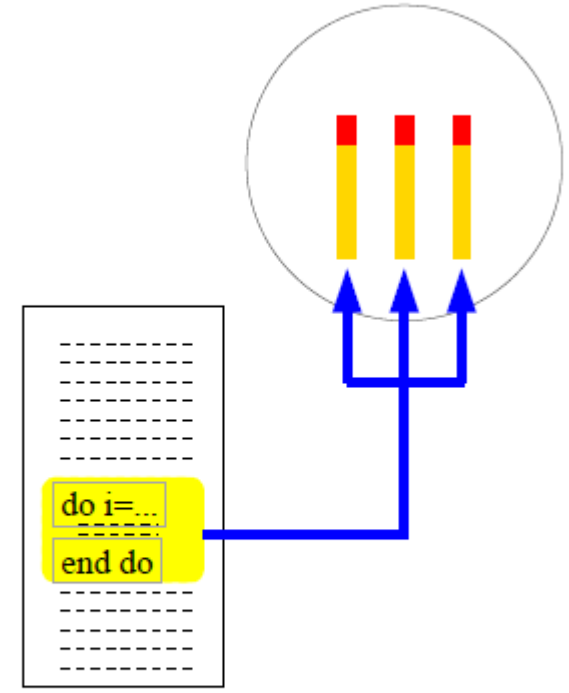
Variables

- Une variable peut être :
  - définie dans la pile (stack) : espace mémoire local d'un processus léger; on parle alors de **variable privée**
  - définie dans un espace de mémoire partagé par tous les processus légers; on parle alors de **variable partagée**.


Espace variables partagées
Piles
(variables locales)
Processus légers
Processus
Région
parallèle
Ensemble d'instructions
Programme

Partage du travail

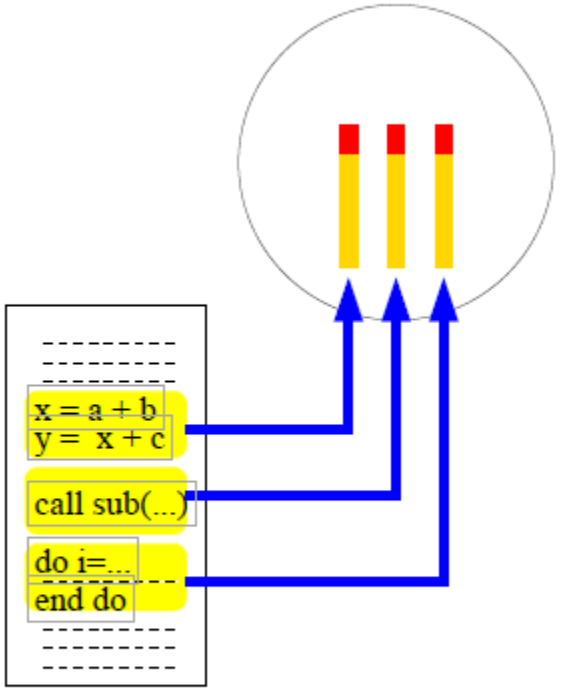
- Le partage du travail consiste essentiellement à :
  - **exécuter une boucle par répartition des itérations entre les tâches;**
  - exécuter plusieurs sections de code mais une seule par tâche;
  - exécuter plusieurs occurrences d'une même procédure par différentes tâches



Boucle parallèle
(Looplevel parallelism)

Partage du travail

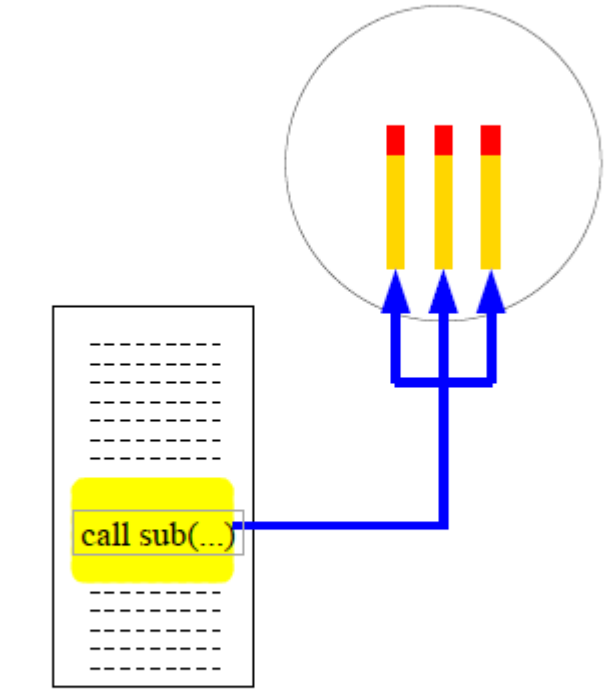
- Le partage du travail consiste essentiellement à :
  - exécuter une boucle par répartition des itérations entre les tâches;
  - **exécuter plusieurs sections de code mais une seule par tâche;**
  - exécuter plusieurs occurrences d'une même procédure par différentes tâches



Sections parallèles

Partage du travail

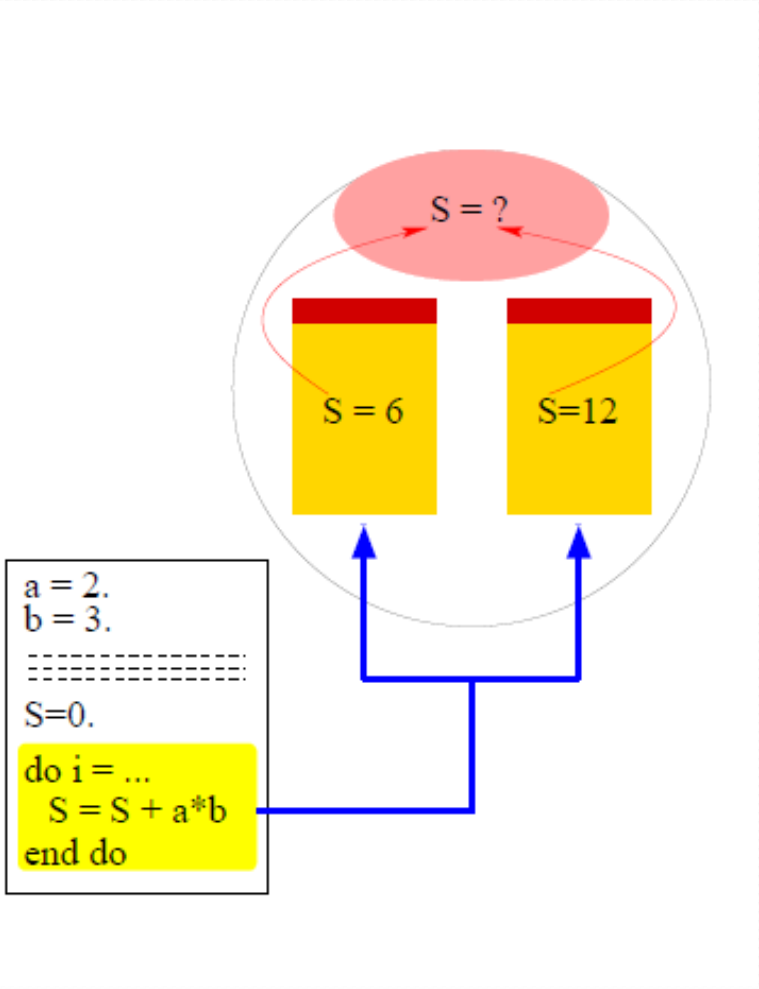
- Le partage du travail consiste essentiellement à :
  - exécuter une boucle par répartition des itérations entre les tâches;
  - exécuter plusieurs sections de code mais une seule par tâche;
  - **exécuter plusieurs occurrences d'une même procédure par différentes tâches**



Procédure parallèle (orphaning)

Synchronisation

- Il est parfois nécessaire d'introduire une synchronisation entre les tâches concurrentes pour éviter, par exemple, que celles-ci modifient dans un ordre quelconque la valeur d'une même variable partagée.



Construction d'une région parallèle

- Dans une région parallèle, par défaut, le statut des variables est partagé.
- Il est interdit d'effectuer des « branchements » (ex. GOTO) vers l'intérieur ou vers l'extérieur d'une région parallèle
- Il existe une barrière implicite de synchronisation en fin de région parallèle.

Sommaire

Notions de calcul parallèle
Présentation d'OpenMP
Structure d'un programme
Boucle for
- Schedule Static
- Schedule Dynamic
- Problème de dépendance
- Opération de réduction
- Opération de réduction : Atomic

Portion de code parallèle
- Exécution exclusive
- Single
- Master
- Sections
- Sections Nowait
- Synchronisation Barrier

Appeler du C (OMP) dans Python
Appeler du C (OMP) dans IDL

Boucle for : répartition des itérations entre les tâches

```
#include <omp.h>
...
#pragma omp parallel for
for (i= 1; i< 10; i++)
{
        //Boucle parallèlisée, région parallèle
}
```

```
// Par defaut les variables sont partagees
#pragma omp parallel for
for (index=0; index < 10; index++)
{
        sleep(index);
        printf("Number of threads = %d\n", omp_get_num_threads());
        printf("Index %d traite par le thread : %d\n",index,omp_get
}
```

Boucle for : Schedule Static

- static.c : #pragma omp parallel for **schedule(static)**

```
#pragma omp parallel for schedule(static)
for (index=0; index < 10; index++)
{
        sleep(index);
        //printf("Number of threads = %d\n", omp_get_n
        printf("Index %d traite par le thread : %d\n",
}
```

Boucle for : Schedule Static

```
grolleau@biruni2:~/dev/omp$ ./static.exe
Index 0 traite par le thread : 0
Index 1 traite par le thread : 0
Index 2 traite par le thread : 0
Index 5 traite par le thread : 1
Index 3 traite par le thread : 0
Index 4 traite par le thread : 0
Index 6 traite par le thread : 1
Index 7 traite par le thread : 1
Index 8 traite par le thread : 1
Index 9 traite par le thread : 1
Temps d'execution : 35 s
```

Boucle for : SCHEDULE

- La répartition SCHEDULE(STATIC) consiste à répartir avant le début de la boucle toutes les itérations sur chacun des cœurs.
- La répartition SCHEDULE(DYNAMIC) consiste à répartir les paquets au fur et à mesure qu'un cœur se libère.

Boucle for : Schedule Dynamic

- Dynamic.c : #pragma omp parallel for **schedule(dynamic)**

```
#pragma omp parallel for schedule(dynamic)
for (index=0; index < 10; index++)
{
        sleep(index);
        //printf("Number of threads = %d\n", omp_get_n
        printf("Index %d traite par le thread : %d\n",
}
```

Boucle for : Schedule Dynamic

```
grolleau@biruni2:~/dev/omp$ ./dynamic.exe
Index 0 traite par le thread : 0
Index 1 traite par le thread : 1
Index 2 traite par le thread : 0
Index 3 traite par le thread : 1
Index 4 traite par le thread : 0
Index 5 traite par le thread : 1
Index 6 traite par le thread : 0
Index 7 traite par le thread : 1
Index 8 traite par le thread : 0
Index 9 traite par le thread : 1
Temps d'execution : 25 s
```

Boucle for : critical

- *critical*: the enclosed code block will be executed by only one thread at a time, and not simultaneously executed by multiple threads. It is often used to protect shared data fromrace conditions.

Boucle for : variable privée/paratgée

- **#pragma omp parallel for default(none) firstprivate(i,j,sum) shared(m,n,a,b,c)**

Boucle for
Problème de dépendance

- Problème de dépendance arrière : boucle non parallèlisable : **tab[index]=tab[index-1];**
- For_dependance.c

```
#pragma omp parallel for
for (index=1; index < TAB_SIZE; index++)
{
        //Dependance arriere
        //Boucle non parallellisable, sauf avec ordered
        tab[index]=tab[index-1];
        printf("Index %d traite par le thread : %d\n",in
}
```

Boucle for
Problème de dépendance

```
fin de la boucle for
tab[0]=0
tab[1]=0
tab[2]=0
tab[3]=2
tab[4]=2
tab[5]=4
tab[6]=4
tab[7]=6
tab[8]=6
tab[9]=8
```

Sommaire

Notions de calcul parallèle
Présentation d'OpenMP
Structure d'un programme
Boucle for
- Schedule Static
- Schedule Dynamic
- Problème de dépendance
- **Opération de réduction**
- Opération de réduction : Atomic

Portion de code parallèle
- Exécution exclusive
- Single
- Master
- Sections
- Sections Nowait
- Synchronisation Barrier

Appeler du C (OMP) dans Python
Appeler du C (OMP) dans IDL

Boucle for
Opération de réduction

- Réduction : le résultat final est issu d'une opération (+,*?-,/) à partir de chaque résultat des itérations.

```
int resultPlus=0,resultMult=1;
```

```
for (index=0; index < nbIter; index++)
{
        sleep(1);
        resultPlus += 1;
        resultMult *= 2;
}
```

Boucle for
Opération de réduction

- No_reduction.c
- #pragma omp parallel for default(shared)

```
#pragma omp parallel for default(shared)
for (index=0; index < nbIter; index++)
{
        sleep(1);
        resultPlusPar += 1;
        resultMultPar *= 2;
        printf("Index %d traite par le thread : %d\n"
}
```

```
grolleau@biruni2:~/dev/omp$ ./no_reduction.exe
Index 16 traite par le thread : 16
Index 10 traite par le thread : 10
Index 19 traite par le thread : 19
Index 15 traite par le thread : 15
Index 17 traite par le thread : 17
Index 18 traite par le thread : 18
Index 7 traite par le thread : 7
Index 8 traite par le thread : 8
Index 14 traite par le thread : 14
Index 5 traite par le thread : 5
Index 6 traite par le thread : 6
Index 12 traite par le thread : 12
Index 1 traite par le thread : 1
Index 9 traite par le thread : 9
Index 2 traite par le thread : 2
Index 11 traite par le thread : 11
Index 3 traite par le thread : 3
Index 4 traite par le thread : 4
Index 13 traite par le thread : 13
Index 0 traite par le thread : 0
fin de la boucle for
Result + attendu = 20, obtenu = 18
Result * attendu = 1048576, obtenu = 262144
Temps d'execution normal : 20 s, temps d'execution parallele : 1 s
```

Boucle for
Opération de réduction

#pragma omp parallel for default(shared)
reduction(+:resultPlusPar) reduction(*:resultMultPar)

```
#pragma omp parallel for default(shared) reduction(+:resultPlusPar) reduction(*:resultMultPar)
for (index=0; index < nbIter; index++)
{
        sleep(1);
        resultPlusPar += 1;
        resultMultPar *= 2;
        printf("Index %d traite par le thread : %d\n",index,omp_get_thread_num());
}
```

```
grolleau@biruni2:~/dev/omp$ ./reduction.exe
Index 18 traite par le thread : 18
Index 16 traite par le thread : 16
Index 4 traite par le thread : 4
Index 9 traite par le thread : 9
Index 13 traite par le thread : 13
Index 2 traite par le thread : 2
Index 5 traite par le thread : 5
Index 10 traite par le thread : 10
Index 0 traite par le thread : 0
Index 3 traite par le thread : 3
Index 11 traite par le thread : 11
Index 6 traite par le thread : 6
Index 17 traite par le thread : 17
Index 7 traite par le thread : 7
Index 14 traite par le thread : 14
Index 8 traite par le thread : 8
Index 15 traite par le thread : 15
Index 12 traite par le thread : 12
Index 19 traite par le thread : 19
Index 1 traite par le thread : 1
fin de la boucle for
Result + attendu = 20, obtenu = 20
Result * attendu = 1048576, obtenu = 1048576
Temps d'execution normal : 20 s, temps d'execution parallele : 1 s
```

Boucle for
Opération de réduction : atomic

- #pragma omp **atomic**
- La directive ATOMIC assure qu'une variable partagée est lue et modifiée en mémoire par une seule tâche à la fois.
- Son effet est local à l'instruction qui suit immédiatement la directive.

Boucle for
Opération de réduction : atomic

```
#pragma omp parallel for default(shared)
for (index=0; index < nbIter; index++)
{
        sleep(1);
        resultPlusPar += 1;
        /* The ATOMIC directive specifies that a spe
         * updated atomically, rather than letting m
        /* Valable uniquement pour la clause qui sui
        #pragma omp atomic
        resultMultPar *= 2;
        printf("Index %d traite par le thread : %d\n
}
```

```
grolleau@biruni2:~/dev/omp$ ./no_reduction_atomic.exe
Index 18 traite par le thread : 18
Index 8 traite par le thread : 8
Index 1 traite par le thread : 1
Index 5 traite par le thread : 5
Index 16 traite par le thread : 16
Index 12 traite par le thread : 12
Index 2 traite par le thread : 2
Index 6 traite par le thread : 6
Index 9 traite par le thread : 9
Index 17 traite par le thread : 17
Index 13 traite par le thread : 13
Index 15 traite par le thread : 15
Index 10 traite par le thread : 10
Index 7 traite par le thread : 7
Index 3 traite par le thread : 3
Index 14 traite par le thread : 14
Index 19 traite par le thread : 19
Index 4 traite par le thread : 4
Index 0 traite par le thread : 0
Index 11 traite par le thread : 11
fin de la boucle for
Result + attendu = 20, obtenu = 15
Result * attendu = 1048576, obtenu = 1048576
Temps d'execution normal : 20 s, temps d'execution parallele : 1 s
```

Boucle for
Opération de réduction : atomic

- Autres exemples :
  - Fichier variables.c et variables_atomic.c

- Fin première séance

Sommaire

Notions de calcul parallèle
Présentation d'OpenMP
Structure d'un programme
Boucle for
- Schedule Static
- Schedule Dynamic
- Problème de dépendance
- Opération de réduction
- Opération de réduction : Atomic

Portion de code parallèle
- Exécution exclusive
- Single
- Master
- Sections
- Sections Nowait
- Synchronisation Barrier

Appeler du C (OMP) dans Python
Appeler du C (OMP) dans IDL

Portion de code parallèle
Exécution exclusive

- Exécution exclusive : Code exécuté une fois à l'intérieur d'un code parallèle
- Il arrive que l'on souhaite exclure toutes les tâches à l'exception d'une seule pour exécuter certaines portions de code incluses dans une région parallèle.
- Pour se faire, OpenMP offre deux directives SINGLE et MASTER.

Portion de code parallèle
Exécution exclusive : Clause SINGLE

- #pragma omp **single**

- La construction SINGLE permet de faire exécuter une portion de code par une et une seule tâche sans pouvoir indiquer laquelle.
- En général, c'est la tâche qui arrive la première sur la construction SINGLE mais cela n'est pas spécifié dans la norme.
- Toutes les tâches n'exécutant pas la région SINGLE attendent, en fin de construction (}), la terminaison de celle qui en a la charge, à moins d'avoir spécifié la clause NOWAIT

Portion de code parallèle

Exécution exclusive : Clause SINGLE

```
#pragma omp parallel
{
        printf("Avant single execute par le thr
        sleep(1);
        /* The SINGLE directive specifies that
d in the team. */
        #pragma omp single
        {
                printf("Dans single DEBUT execu

                sleep(3);
                printf("Dans single FIN execute
        }

        printf("Apres single traite par le thre
}
```

Portion de code parallèle
Exécution exclusive : Clause SINGLE

```
grolleau@biruni2:~/dev/omp$ ./single.exe
Number of threads = 1
Avant single execute par le thread : 0
Avant single execute par le thread : 2
Avant single execute par le thread : 3
Avant single execute par le thread : 1
Dans single DEBUT execute par le thread : 2
Dans single FIN execute par le thread : 2
Apres single traite par le thread : 2
Apres single traite par le thread : 3
Apres single traite par le thread : 1
Apres single traite par le thread : 0
```

Portion de code parallèle
Exécution exclusive : Clause MASTER

- #pragma omp **master**

- La construction MASTER permet de faire exécuter une portion de code par la tâche maître seule.
- Cette construction n'admet aucune clause.
- Il n'existe aucune barrière de synchronisation ni en début (MASTER) ni en fin de construction (}).

Portion de code parallèle
Exécution exclusive : Clause MASTER

```
#pragma omp parallel
{
        printf("Avant master execute par le thread : %d
        sleep(1);
        /* The MASTER directive specifies a region that
of the team.
         * All other threads on the team skip this sect
        #pragma omp master
        {
                printf("Dans master DEBUT execute par l

                sleep(3);
                printf("Dans master FIN execute par le
        }

        printf("Apres master traite par le thread : %d\
}
```

Portion de code parallèle
Exécution exclusive : Clause MASTER

```
grolleau@biruni2:~/dev/omp$ ./master.exe
Number of threads = 1
Avant master execute par le thread : 0
Avant master execute par le thread : 1
Avant master execute par le thread : 3
Avant master execute par le thread : 2
Dans master DEBUT execute par le thread : 0
Apres master traite par le thread : 1
Apres master traite par le thread : 2
Apres master traite par le thread : 3
Dans master FIN execute par le thread : 0
Apres master traite par le thread : 0
```

Portion de code parallèle
Sections

- #pragma omp **sections**, #pragma omp **section**
- Une section est une portion de code exécutée par une et une seule tâche.
- Plusieurs portions de code peuvent être définies par l'utilisateur à l'aide de la directive SECTION au sein d'une construction SECTIONS.
- Le but est de pouvoir répartir l'exécution de plusieurs portions de code indépendantes sur les différentes tâches.
- Par défaut, il y a un point de synchronisation à la fin de l'ensemble des SECTIONs

Portion de code parallèle Sections

```
#pragma omp parallel
{
        #pragma omp sections
        {

                //  Each SECTION is executed once by a thread in the
                #pragma omp section
                {
                        printf("Section 1 traitee par le thread : %d
                        sleep(5);
                        printf("Fin section 1\n");
                }

                #pragma omp section
                {
                        printf("Section 2 traitee par le thread : %d
                        sleep(2);
                        printf("Fin section 2\n");
                }

        }
```

Portion de code parallèle

Sections

```
grolleau@biruni2:~/dev/omp$ ./section.exe
Section 1 traitee par le thread : 2
Section 2 traitee par le thread : 0
Fin section 2
Fin section 1
Hors section traite par le thread : 2
Hors section traite par le thread : 1
Hors section traite par le thread : 3
Hors section traite par le thread : 0
```

Portion de code parallèle
Sections Nowait

- #pragma omp **sections nowait**
- Avec la clause NOWAIT : il n'y a plus de point de synchronisation à la fin de l'ensemble des SECTIONS

```
#pragma omp sections nowait
{

        //  Each SECTION is executed once by a thread in the
        #pragma omp section
        {
                printf("Section 1 traitee par le thread : %d\
                sleep(5);
                printf("Fin section 1\n");
        }

        #pragma omp section
        {
                printf("Section 2 traitee par le thread : %d\
                sleep(2);
                printf("Fin section 2\n");
        }
```

Portion de code parallèle
Sections Nowait

```
grolleau@biruni2:~/dev/omp$ ./section_nowait.exe
Section 1 traitee par le thread : 0
Section 2 traitee par le thread : 3
Hors section traite par le thread : 1
Hors section traite par le thread : 2
Fin section 2
Hors section traite par le thread : 3
Fin section 1
Hors section traite par le thread : 0
```

Sommaire

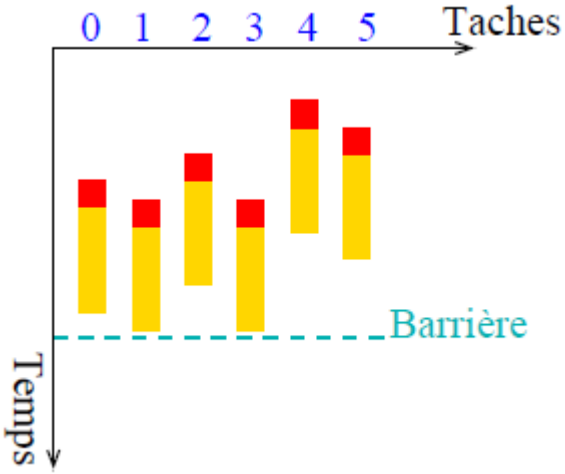
Notions de calcul parallèle
Présentation d'OpenMP
Structure d'un programme
Boucle for
- Schedule Static
- Schedule Dynamic
- Problème de dépendance
- Opération de réduction
- Opération de réduction : Atomic

Portion de code parallèle
- Exécution exclusive
- Single
- Master
- Sections
- Sections Nowait
- **Synchronisation Barrier**

Appeler du C (OMP) dans Python
Appeler du C (OMP) dans IDL

Portion de code parallèle
Synchronisation : clause BARRIER

- #pragma omp **barrier**
- La directive BARRIER synchronise l'ensemble des tâches concurrentes dans une région parallèle.
- Chacune des tâches attend que toutes les autres soient arrivées à ce point de synchronisation pour reprendre, ensemble, l'exécution du programme.



Portion de code parallèle

Synchronisation : clause BARRIER

```
#pragma omp parallel
{
        printf("Portion avant le sleep executee par le thread : %
        /* Chaque thread se repose le temps de son indice */
        sleep(omp_get_thread_num()+1);

        printf("Portion avant la bariere executee par le thread :

        /* The BARRIER directive synchronizes all threads in the
         * When a BARRIER directive is reached, a thread will wai
         * until all other threads have reached that barrier. */
        #pragma omp barrier
        printf("Portion apres la bariere executee par le thread :
}
```

Portion de code parallèle
Synchronisation : clause BARRIER

```
grolleau@biruni2:~/dev/omp$ ./barrier.exe
Portion avant le sleep executee par le thread : 0
Portion avant le sleep executee par le thread : 3
Portion avant le sleep executee par le thread : 2
Portion avant le sleep executee par le thread : 1
Portion avant la bariere executee par le thread : 0
Portion avant la bariere executee par le thread : 1
Portion avant la bariere executee par le thread : 2
Portion avant la bariere executee par le thread : 3
Portion apres la bariere executee par le thread : 3
Portion apres la bariere executee par le thread : 2
Portion apres la bariere executee par le thread : 0
Portion apres la bariere executee par le thread : 1
```

Sommaire

Notions de calcul parallèle
Présentation d'OpenMP
Structure d'un programme
Boucle for
- Schedule Static
- Schedule Dynamic
- Problème de dépendance
- Opération de réduction
- Opération de réduction : Atomic

Portion de code parallèle
- Exécution exclusive
- Single
- Master
- Sections
- Sections Nowait
- Synchronisation Barrier

Appeler du C (OMP) dans Python
Appeler du C (OMP) dans IDL

Appeler du C (OMP) dans Python

- Principe : construire un module d'extension de Python en C via le package python distutils (inclus dans python 2.0)
- https://docs.python.org/2/extending/building.html

1. Ecrire un fichier setup.py
2. Ecrire le fichier C correspondant
3. Exécuter : python setup.py build
4. Exécuter example.py

Appeler du C (OMP) dans Python

- Setup.py

```
from distutils.core import setup, Extension

module1 = Extension('pycopenmp',
                    define_macros = [('MAJOR_VERSION', '1'),
                                     ('MINOR_VERSION', '0')],
                    extra_compile_args = ['-fopenmp'],
                    extra_link_args = ['-lgomp'],
                    sources = ['py_c_omp.c','prime_number.c'])

setup (name = 'Python C-OpenMP',
       version = '1.0',
       description = 'This is a demo package for C & Python & OpenMP',
       author = 'E. Grolleau',
       author_email = 'emmanuel.grolleau@obspm.fr',
       url = 'http://docs.python.org/extending/building',
       ext_modules = [module1])
```

Appeler du C (OMP) dans Python

- Principe : construire un module d'extension de Python en C via le package python distutils (inclus dans python 2.0)
- https://docs.python.org/2/extending/building.html

1. Ecrire un fichier setup.py
2. Ecrire le fichier C correspondant
3. Exécuter : python setup.py build
4. Exécuter example.py

Appeler du C (OMP) dans Python

- py_c_omp.c
- **#include <Python.h>**
- **#include <omp.h>**

```
static char pycopenmp_primes_doc[] =
"primes(number)\n\
\n\
Returns the number of primes between 1 and N.";

static PyMethodDef pycopenmp_methods[] = {
        {"primes", pycopenmp_primes, METH_VARARGS, pycopenmp_primes_doc},
        {NULL, NULL}
};

PyMODINIT_FUNC
initpycopenmp(void)
{
        (void)Py_InitModule3("pycopenmp",pycopenmp_methods, pycopenmp_doc );
}
```

```
static PyObject*
pycopenmp_primes(PyObject *self, PyObject *args)
{
        int number;
        int result;
        //if (!PyArg_UnpackTuple(args, "add", 2, 2, &a, &b)) {

        // On fixe le nb de thread a utiliser
        omp_set_num_threads(48);

        if (!PyArg_ParseTuple(args, "i", &number ))
        {
                return NULL;
        }

        double start = omp_get_wtime( ); //start the timer

        result = prime_number(number); // Calcul

        double stop = omp_get_wtime( ); //stop the timer

        double dif = stop - start; // stores the difference in dif
        printf("(Time : %.2lf s)", dif);

        return Py_BuildValue("i",result);
}
```

Appeler du C (OMP) dans Python

- Prime_number.c

```
# pragma omp for reduction ( +:total )
  for ( i = 2; i <= n; i++ )
  {
    prime = 1;

    for ( j = 2; j < i; j++ )
    {
      if ( i % j == 0 )
      {
        prime = 0;
        break;
      }
    }
    total = total + prime;
  }
```

Appeler du C (OMP) dans Python

- Principe : construire un module d'extension de Python en C via le package python distutils (inclus dans python 2.0)
- https://docs.python.org/2/extending/building.html

1. Ecrire un fichier setup.py
2. Ecrire le fichier C correspondant
3. Exécuter : python setup.py build
   - => créer un répertoire build avec pycopenmp.so
4. Exécuter example.py

Appeler du C (OMP) dans Python

```
#!/usr/bin/python

# python example.py &
# ps exmo user,pid,tid,tt,psr,pcpu,stat,tmout,f,wchan:12,comm
# htop
from pycopenmp import primes

 # Un nombre premier est un entier naturel qui admet exactement deux diviseurs distincts
 # entiers et positifs (qui sont alors 1 et lui-meme).
 # Ainsi, 1 n est pas premier car il n a qu un seul diviseur entier positif ;
 # 0 non plus car il est divisible par tous les entiers positifs.

print "C : Nombre de nombre premier de 1 a 1  attendu = 0, obtenu = ", primes(1)
print "C : Nombre de nombre premier de 1 a 10 attendu = 4, obtenu = ", primes(10)
print "C : Nombre de nombre premier de 1 a 100 attendu = 25, obtenu = ", primes(100)
print "C : Nombre de nombre premier de 1 a 1000 attendu = 168, obtenu = ", primes(1000)
print "C : Nombre de nombre premier de 1 a 10,000 attendu = 1229, obtenu = ", primes(10000)
print "C : Nombre de nombre premier de 1 a 100,000 attendu = 9592, obtenu = ", primes(100000)
print "Nombre de nombre premier de 1 a 300,000 attendu = 25997, obtenu = ", primes(300000)
#print "Nombre de nombre premier de 1 a 1,000,000 attendu = 78498, obtenu = ", primes(1000000)

# Puremant Python
```

Sommaire

Notions de calcul parallèle
Présentation d'OpenMP
Structure d'un programme
Boucle for
- Schedule Static
- Schedule Dynamic
- Problème de dépendance
- Opération de réduction
- Opération de réduction : Atomic

Portion de code parallèle
- Exécution exclusive
- Single
- Master
- Sections
- Sections Nowait
- Synchronisation Barrier

Appeler du C (OMP) dans Python
Appeler du C (OMP) dans IDL

Appeler du C (OMP) dans IDL

- Principe :

1. Récupérer le fichier idl_export.h
2. Ecrire un fichier C « ayant conscience d'IDL »
3. Compiler
4. Appeler la fonction C compilée dans IDL

Appeler du C (OMP) dans IDL

- Principe :

1. Récupérer le fichier idl_export.h

   /usr/local/exelis/idl83/external/include/idl_export.h

2. Ecrire un fichier C « ayant conscience d'IDL »
3. Compiler
4. Appeler la fonction C compilée dans IDL

Appeler du C (OMP) dans IDL

- Principe :

1. Récupérer le fichier idl_export.h
2. Ecrire un fichier C « ayant conscience d'IDL »
3. Compiler
4. Appeler la fonction C compilée dans IDL

Appeler du C (OMP) dans IDL

- idl_c_omp.c

```
#include <stdio.h>
#include <omp.h>
#include "idl_export.h"
#include "prime_number.h"


int primes(IDL_LONG number)
{
        // On fixe le nb de thread a utiliser
        omp_set_num_threads(48);

        double start = omp_get_wtime( ); //start the timer

        IDL_LONG result = prime_number(number); // Calcul

        double stop = omp_get_wtime( ); //stop the timer

        double dif = stop - start; // stores the difference in dif
        printf("(Time : %.2lf s)", dif);

        return result;
}
```

Appeler du C (OMP) dans IDL

- Principe :

1. Récupérer le fichier idl_export.h
2. Ecrire un fichier C « ayant conscience d'IDL »
3. Compiler
4. Appeler la fonction C compilée dans IDL

Appeler du C (OMP) dans IDL

Compiler les sources

- build_so:idl_c_omp.c
- $(GCC) -fPIC -shared -lgomp -o idlcopenmp.so idl_c_omp.c prime_number.c

- => Créer un fichier idlcopenmp.so

Appeler du C (OMP) dans IDL

- Principe :

1. Récupérer le fichier idl_export.h
2. Ecrire un fichier C « ayant conscience d'IDL »
3. Compiler
4. Appeler la fonction C compilée dans IDL

Appeler du C (OMP) dans IDL

Appeler la fonction C compilée dans IDL

number = **CALL_EXTERNAL**(pathSO+"idlcopenmp.so", "primes", 1000, /AUTO_GLUE,/ALL_VALUE,/L64_VALUE)

/AUTO_GLUE : Créer automatiquement une function d'interface

/ALL_VALUE : Passage de tous les arguments par Valeur

/L64_VALUE : type d'argument en retour (entier sur 8 octets)

Appeler du C (OMP) dans IDL

- **Exécution**

grolleau@biruni2:~/dev/omp$ **idl example.pro**

IDL Version 8.2.3 (linux x86_64 m64). (c) 2013, Exelis Visual Information Solutions, Inc.

Installation number: 10367.

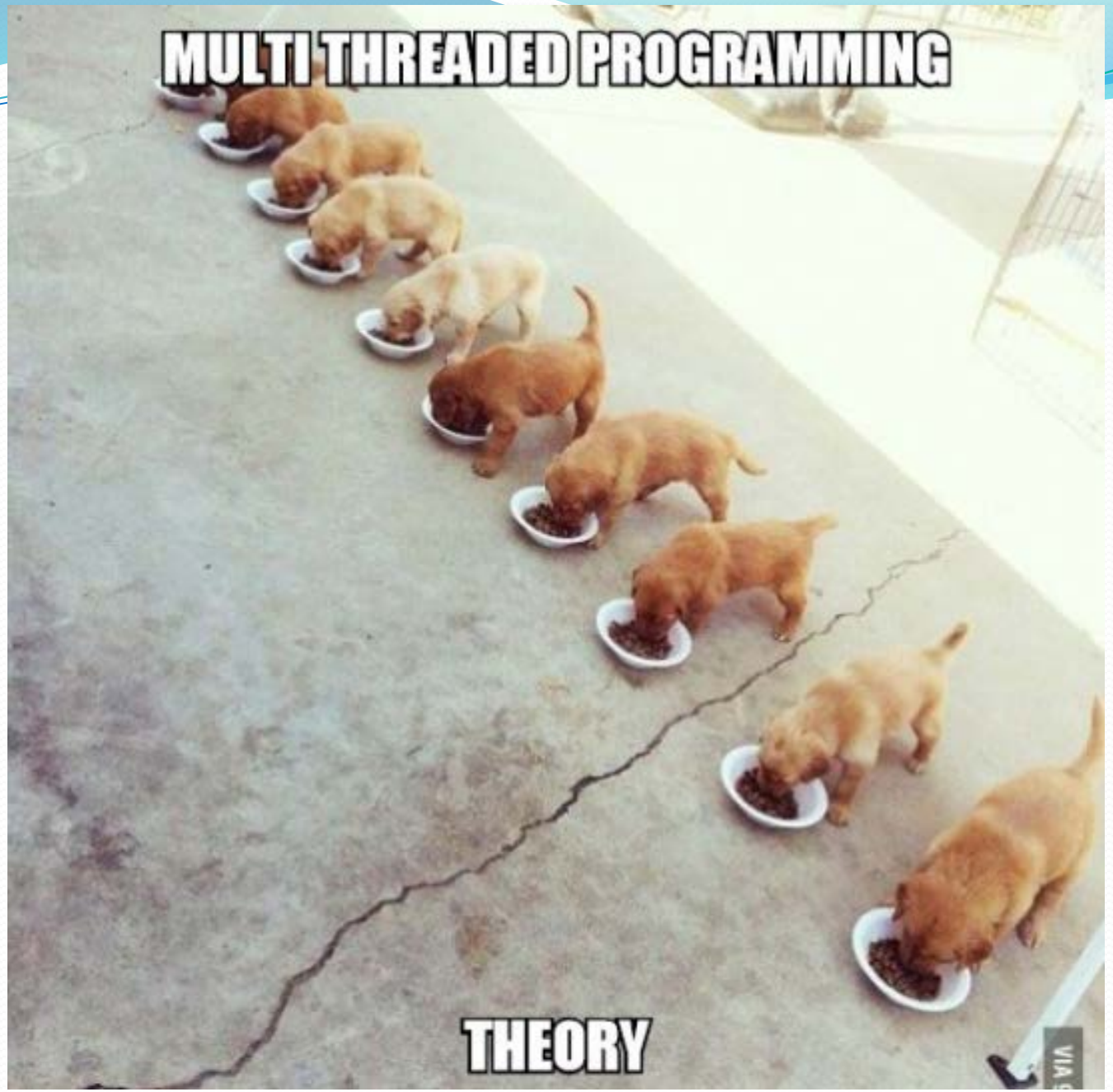
Licensed for use by: LESIA

Nombre de nombre premier de 1 a 1 attendu = 0, obtenu = 0

Nombre de nombre premier de 1 a 10 attendu = 4, obtenu = 4

Nombre de nombre premier de 1 a 100 attendu = 25, obtenu = 25

Nombre de nombre premier de 1 a 1000 attendu = 168, obtenu = 168


MULTI THREADED PROGRAMMING
THEORY


PRACTICE