# OpenMP

## Parallélisation multitâches pour machines à mémoire partagée

**Jalel Chergui**
**Pierre-François Lavallée**
<Prénom.Nom@idris.fr>

# OpenMP : plan

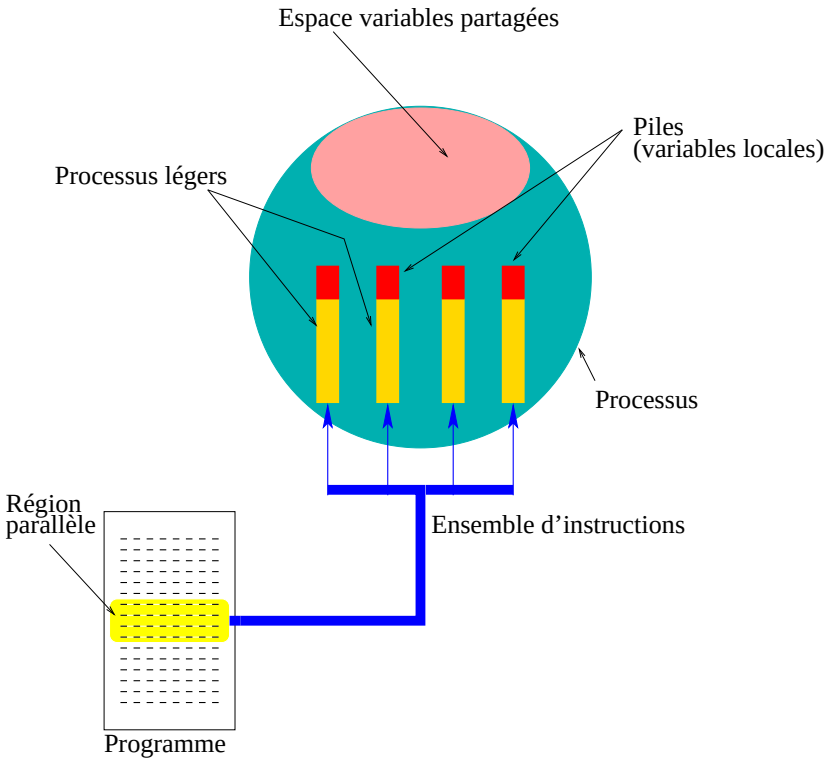
# 1 – Introduction

## 1.1 – Historique

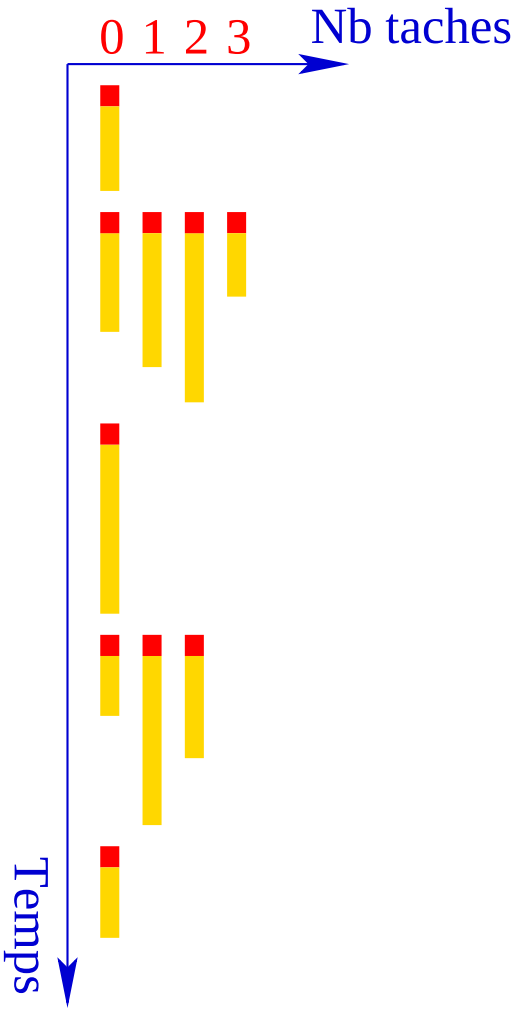
- La parallélisation multitâches existait depuis longtemps chez certains constructeurs (ex. CRAY, NEC, IBM, ...), mais chacun avait son propre jeu de directives.
- Le retour en force des machines multiprocesseurs à mémoire partagée a poussé à définir un standard.
- La tentative de standardisation de PCF (*Parallel Computing Forum*) n'a jamais été adoptée par les instances officielles de normalisation.
- Le 28 octobre 1997, une majorité importante d'industriels et de constructeurs ont adopté `OpenMP` (*Open Multi Processing*) comme un standard dit « industriel ».
- Les spécifications d'`OpenMP` appartiennent aujourd'hui à l'ARB (*Architecture Review Board*), seul organisme chargé de son évolution.
- Une version `OpenMP-2` a été finalisée en novembre 2000. Elle apporte surtout des extensions relatives à la parallélisation de certaines constructions `Fortran` 95.

## 1.2 – Concepts généraux

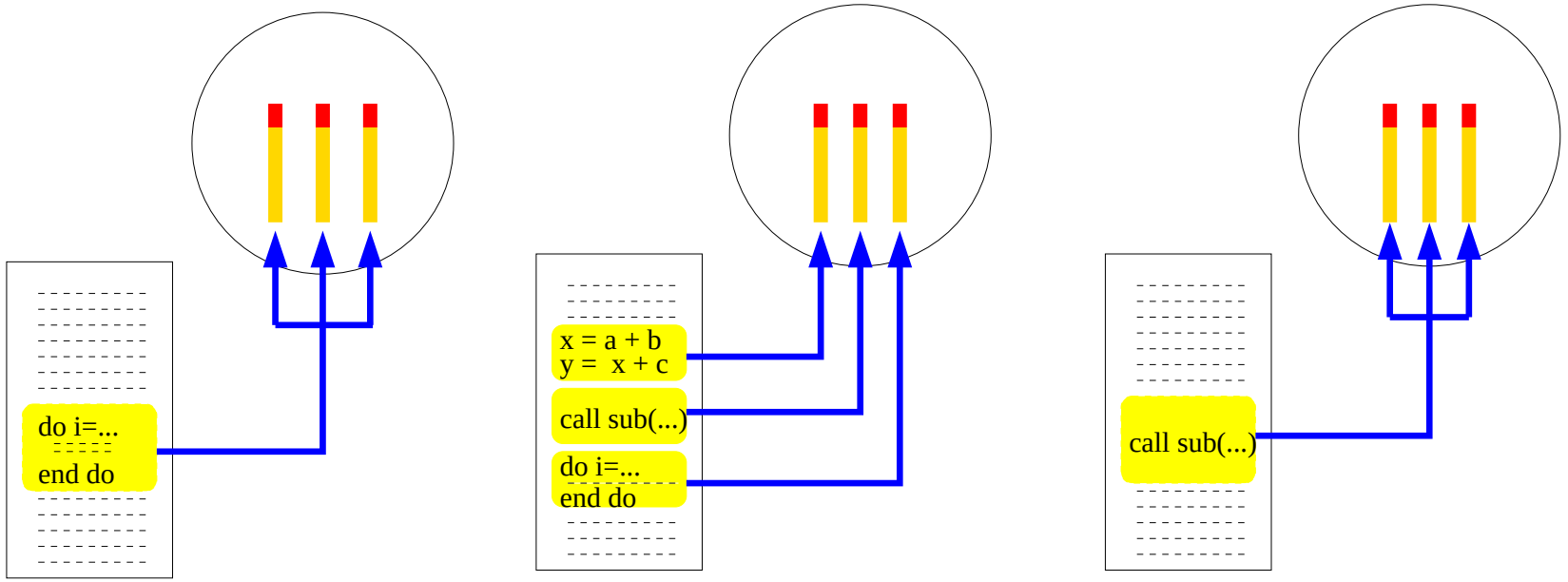
- Un programme `OpenMP` est exécuté par un processus unique.
- Ce processus active des processus légers (*threads*) à l'entrée d'une région parallèle.
- Chaque processus léger exécute une tâche composée d'un ensemble d'instructions.
- Pendant l'exécution d'une tâche, une variable peut être lue et/ou modifiée en mémoire.
  - Elle peut être définie dans la pile (*stack*) (espace mémoire local) d'un processus léger ; on parle alors de variable privée.
  - Elle peut être définie dans un espace mémoire partagé par tous les processus légers ; on parle alors de variable partagée.



- 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.



- 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 (*orphaning*).

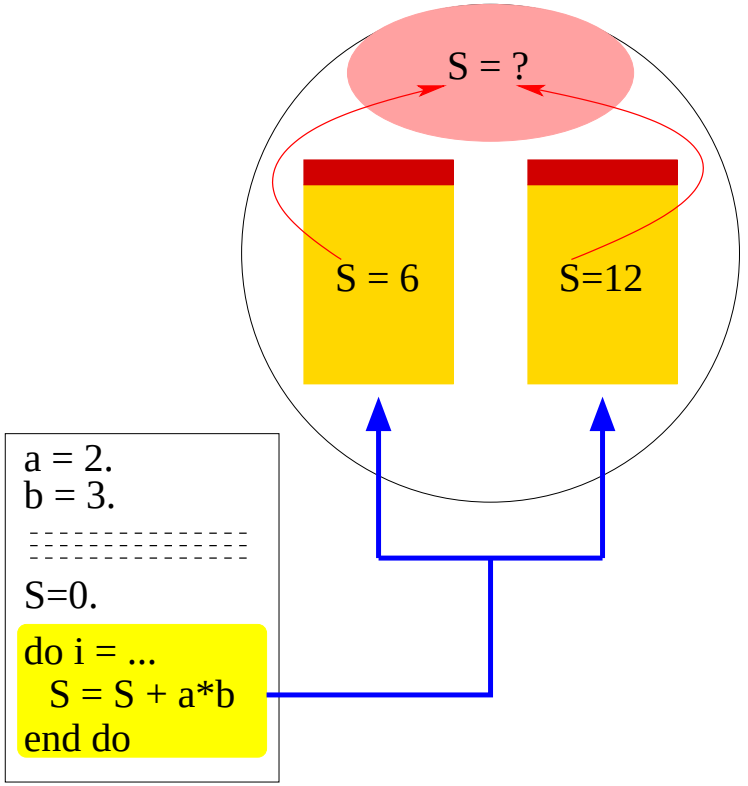



Boucle parallèle (Looplevel parallelism)

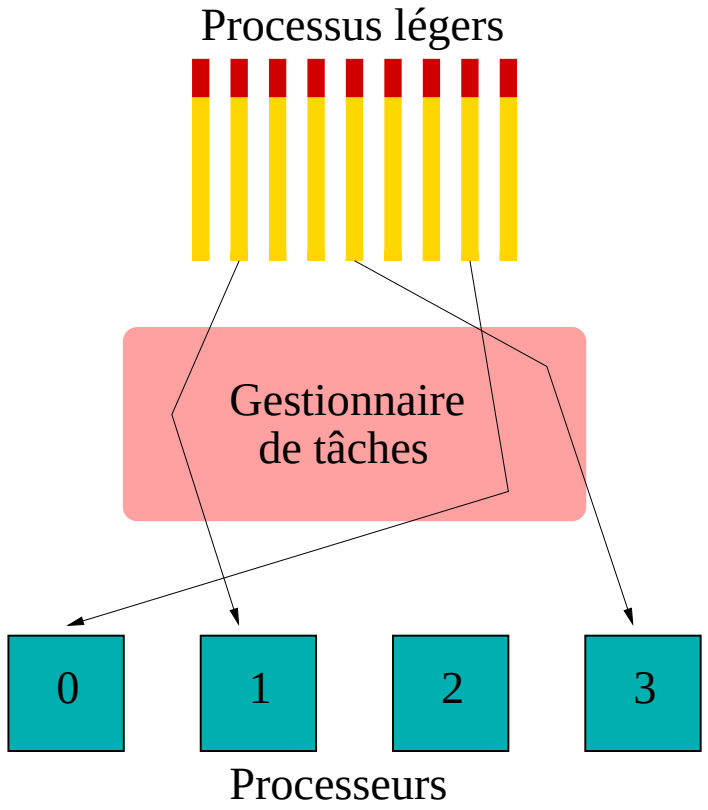
Sections parallèles

Procédure parallèle (orphaning)

☞ 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 (cas des opérations de réduction).

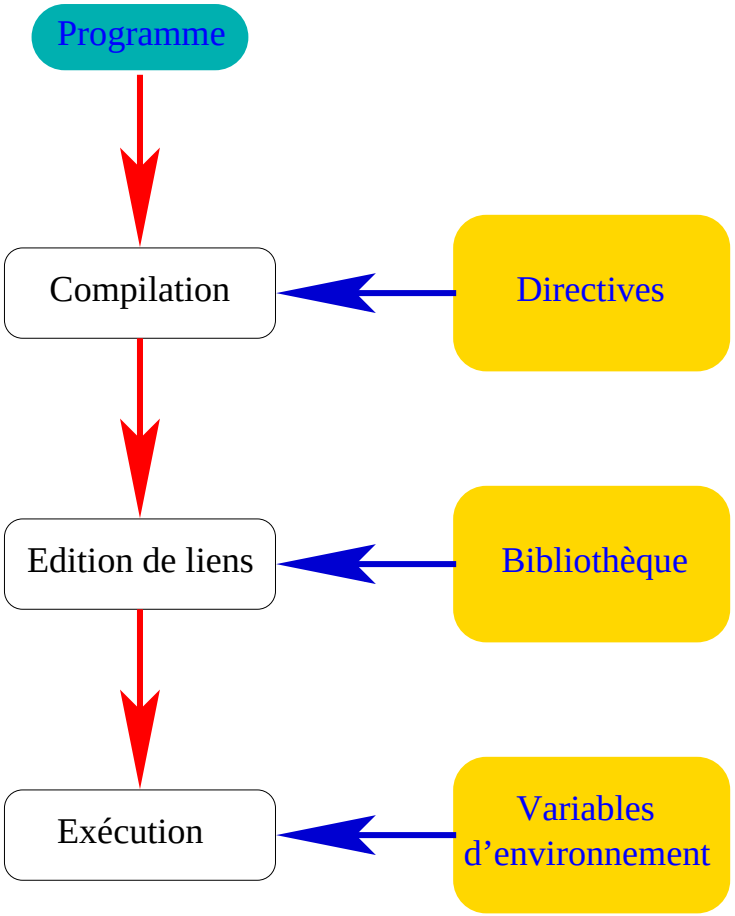


- Les tâches sont affectées aux processeurs par le système d'exploitation. Différents cas peuvent se produire :
  - au mieux, à chaque instant, il existe une tâche par processeur avec autant de tâches que de processeurs dédiés pendant toute la durée du travail ;
  - au pire, toutes les tâches sont traitées séquentiellement par un et un seul processeur ;
  - en réalité, pour des raisons essentiellement d'exploitation sur une machine dont les processeurs ne sont pas dédiés, la situation est en général intermédiaire.



## 1.3 – Structure d'OpenMP

1. Directives et clauses de compilation :
   - ☞ elles servent à définir le partage du travail, la synchronisation et le statut privé ou partagé des données ;
   - ☞ elles sont considérées par le compilateur comme des lignes de commentaires à moins de spécifier une option adéquate de compilation pour qu'elles soient interprétées.
2. Fonctions et sous-programmes : ils font partie d'une bibliothèque chargée à l'édition de liens du programme.
3. Variables d'environnement : une fois positionnées, leurs valeurs sont prises en compte à l'exécution.



Voici les options de compilation pour forcer l'interprétation des directives OpenMP par certains compilateurs Fortran :

- sur IBM SP : -q smp=omp

```
xlf_r -qsuffix=f=f90 -qnosave -qsmp=omp prog.f90  # Compilation et chargement
export OMP_NUM_THREADS=4                          # Nombre de tâches souhaité
a.out                                             # Exécution
```

- sur NEC SX-5 : -Popenmp

```
f90 -Popenmp prog.f90                             # Compilation et chargement
export OMP_NUM_THREADS=4                          # Nombre de tâches souhaité
a.out                                             # Exécution
```

- sur SGI O2000 : -mp

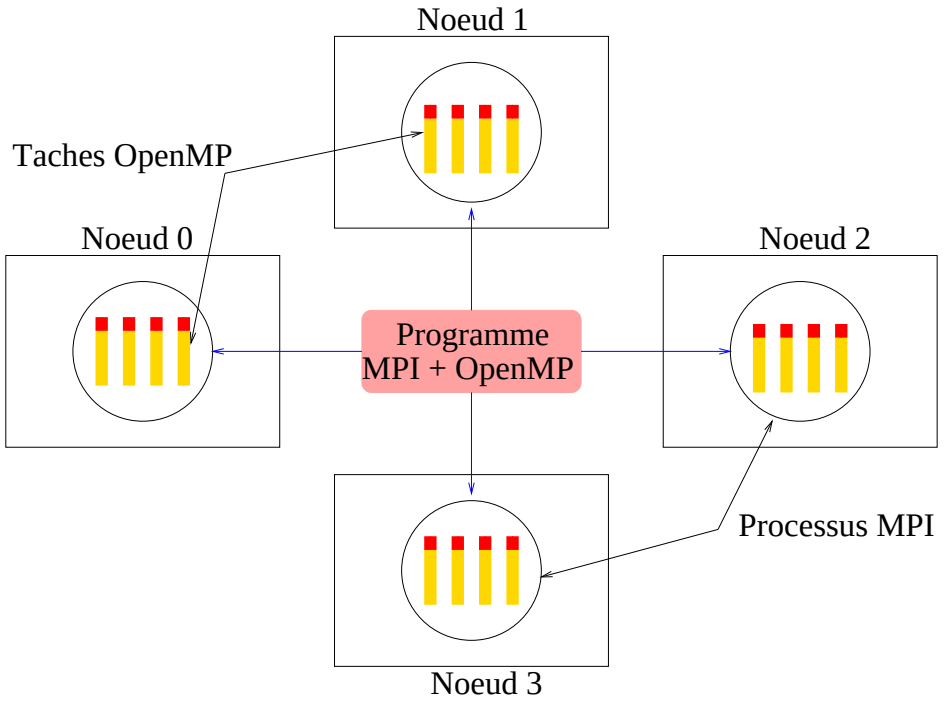
```
f90 -mp prog.f90                                  # Compilation et chargement
export OMP_NUM_THREADS=4                          # Nombre de tâches souhaité
a.out                                             # Exécution
```

## 1.4 – OpenMP versus MPI

Ce sont deux modèles complémentaires de parallélisation.

- `OpenMP`, comme `MPI`, possède une interface `Fortran`, `C` et `C++`.
- `MPI` est un modèle multiprocessus dont le mode de communication entre les processus est explicite (la gestion des communications est à la charge de l'utilisateur).
- `OpenMP` est un modèle multitâches dont le mode de communication entre les tâches est implicite (la gestion des communications est à la charge du compilateur).

- MPI est utilisé en général sur des machines multiprocesseurs à mémoire distribuée.
- OpenMP est utilisé sur des machines multiprocesseurs à mémoire partagée.
- Sur une grappe de machines indépendantes (nœuds) multiprocesseurs à mémoire partagée, la mise en œuvre d'une parallélisation à deux niveaux (MPI et OpenMP) dans un même programme peut être un atout majeur pour les performances parallèles du code.



## 1.5 – Bibliographie

- Premier livre sur `OpenMP` : R. Chandra & *al.*, *Parallel Programming in OpenMP*, éd. Morgan Kaufmann Publishers, oct. 2000.
- Spécifications validées du standard `OpenMP`, http://www.openmp.org/specs/
- E. Gondet, support de présentation `OpenMP`, http://www.idris.fr/data/publications/openmp.pdf

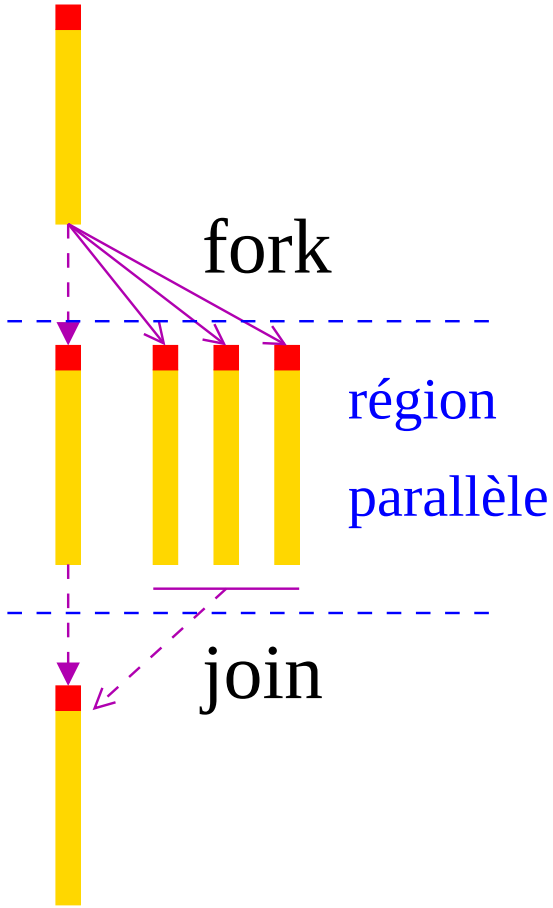
Comme les exemples et les travaux dirigés du cours sont en `Fortran`, voici quelques références utiles en la matière :

- P. Lignelet, Manuel complet du langage `Fortran` 90 et `Fortran` 95, calcul intensif et génie logiciel, éd. Masson, 1996.
- P. Corde & H. Delouis, Support de cours `Fortran` 95, http://www.idris.fr/data/cours/lang/f90/choix_doc.html, CNRS/IDRIS.

# Position dans le cours

## 2 – Principes

- Il est à la charge du développeur d'introduire des directives `OpenMP` dans son code (du moins en l'absence d'outils de parallélisation automatique).
- À l'exécution du programme, le système d'exploitation construit une région parallèle sur le modèle « *fork and join* ».
- À 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.



## 2.1 – Syntaxe générale d'une directive

☞ Une directive `OpenMP` possède la forme générale suivante :

```
sentinelle directive [clause[ clause]...]
```

☞ C'est une ligne qui doit être ignorée par le compilateur si l'option permettant l'interprétation des directives `OpenMP` n'est pas spécifiée.

☞ La sentinelle est une chaîne de caractères dont la valeur dépend du langage utilisé.

☞ Il existe un module Fortran 95 `OMP_LIB` et un fichier d'inclusion C/C++ `omp.h` qui définissent le prototype de toutes les fonctions OpenMP. Il est indispensable de les inclure dans toute unité de programme OpenMP utilisant ces fonctions.

☞ Pour Fortran, en format libre :

```
  use OMP_LIB
  ...
  !$OMP PARALLEL PRIVATE(a,b) &
        !$OMP FIRSTPRIVATE(c,d,e)
       ...
  !$OMP END PARALLEL ! C'est un commentaire
```

☞ Pour Fortran, en format fixe :

```
      use OMP_LIB
      ...
C$OMP PARALLEL PRIVATE(a,b)
C$OMP1 FIRSTPRIVATE(c,d,e)
       ...
C$OMP END PARALLEL
```

☞ Pour C et C++ :

```
#include <omp.h>
...
#pragma omp parallel private(a,b) firstprivate(c,d,e)
{ ... }
```

## 2.2 – Construction d'une région parallèle

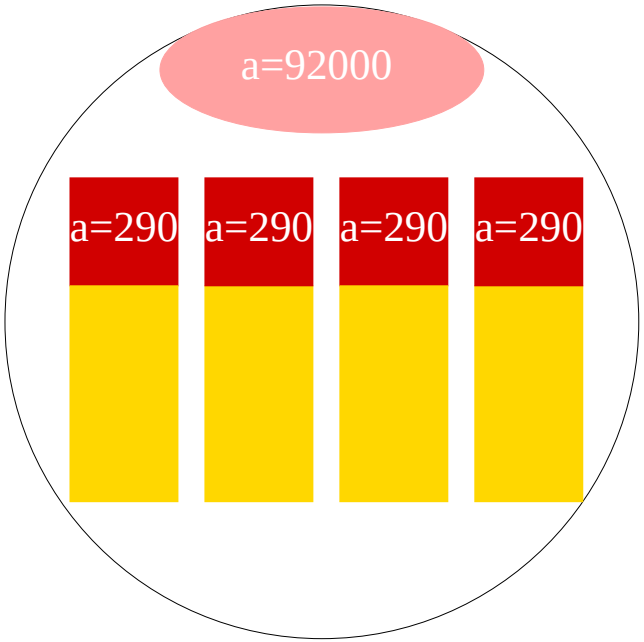
- Dans une région parallèle, par défaut, le statut des variables est partagé.
- Au sein d'une même région parallèle, toutes les tâches concurrentes exécutent le même code.
- Il existe une barrière implicite de synchronisation en fin de région parallèle.
- Il est interdit d'effectuer des « branchements » (ex. `GOTO`, `CYCLE`, etc.) vers l'intérieur ou vers l'extérieur d'une région parallèle ou de toute autre construction `OpenMP`.

```
program parallel
 use OMP_LIB
 implicit none
 real    :: a
 logical :: p

 a = 92290. ; p=.false.
 !$OMP PARALLEL
  !$ p = OMP_IN_PARALLEL()
  print *,"A vaut : ",a, &
          "; p vaut : ",p
 !$OMP END PARALLEL
end program parallel
```

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4
> a.out
```

```
A vaut : 92290. ; p vaut : T
A vaut : 92290. ; p vaut : T
A vaut : 92290. ; p vaut : T
A vaut : 92290. ; p vaut : T
```

- Il est possible, grâce à la clause DEFAULT, de changer le statut par défaut des variables dans une région parallèle.
- Si une variable possède un statut privé (PRIVATE), elle se trouve dans la pile de chaque tâche. Sa valeur est alors indéfinie à l'entrée d'une région parallèle.

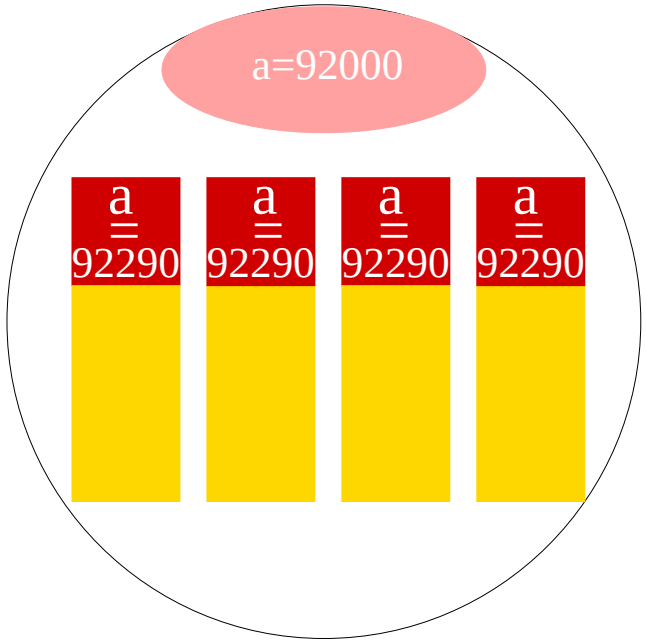



```
program parallel
 implicit none
 real :: a

 a = 92000.
 !$OMP PARALLEL DEFAULT(PRIVATE)
   a = a + 290.
   print *,"A vaut : ",a
 !$OMP END PARALLEL
end program parallel
```

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4
> a.out
```

```
 A vaut :  290.
 A vaut :  290.
 A vaut :  290.
 A vaut :  290.
```

☞ Cependant, grâce à la clause FIRSTPRIVATE, il est possible de forcer l'initialisation de cette variable privée à la dernière valeur qu'elle avait avant l'entrée dans la région parallèle.




```
program parallel
 implicit none
 real :: a

 a = 92000.
!$OMP PARALLEL DEFAULT(NONE) &
        !$OMP FIRSTPRIVATE(a)
   a = a + 290.
   print *,"A vaut : ",a
!$OMP END PARALLEL
print*,"Hors region, A vaut :",a
end program parallel
```

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4
> a.out
```

```
 A vaut :  92290.
 A vaut :  92290.
 A vaut :  92290.
 A vaut :  92290.
 Hors region, A vaut : 92000.
```

## 2.3 – Étendue d'une région parallèle

- L'étendue d'une construction OpenMP représente le champ d'influence de celle-ci dans le programme.
- L'influence (ou la portée) d'une région parallèle s'étend aussi bien au code contenu lexicalement dans cette région (étendue statique), qu'au code des sous-programmes appelés. L'union des deux représente « l'étendue dynamique ».

```
program parallel
 implicit none
 !$OMP PARALLEL
    call sub()
 !$OMP END PARALLEL
end program parallel
subroutine sub()
 use OMP_LIB
 implicit none
 logical :: p
 !$ p = OMP_IN_PARALLEL()
 !$ print *,"Parallele ?:", p
end subroutine sub
```

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4;a.out
```

```
 Parallele ? : T
 Parallele ? : T
 Parallele ? : T
 Parallele ? : T
```

☞ Dans un sous-programme appelé dans une région parallèle, les variables locales et automatiques sont implicitement privées à chacune des tâches (elles sont définies dans la pile de chaque tâche).

```
program parallel
 implicit none
 !$OMP PARALLEL DEFAULT(SHARED)
    call sub()
 !$OMP END PARALLEL
end program parallel
subroutine sub()
 use OMP_LIB
 implicit none
 integer :: a
 a = 92290
 a = a + OMP_GET_THREAD_NUM()
 print *,"A vaut : ",a
end subroutine sub
```

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4;a.out
```

```
 A vaut :  92290
 A vaut :  92291
 A vaut :  92292
 A vaut :  92293
```

## 2.4 – Cas de la transmission par argument

☞ Dans une procédure, toutes les variables transmises par argument (*dummy parameters*) héritent du statut défini dans l'étendue lexicale (statique) de la région.

```
program parallel
 implicit none
 integer :: a, b

 a = 92000
 !$OMP PARALLEL SHARED(a) PRIVATE(b)
    call sub(a, b)
    print *,"B vaut : ",b
 !$OMP END PARALLEL
end program parallel

subroutine sub(x, y)
 use OMP_LIB
 implicit none
 integer :: x, y

 y = x + OMP_GET_THREAD_NUM()
end subroutine sub
```

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4
> a.out
```

```
 B vaut : 92002
 B vaut : 92003
 B vaut : 92001
 B vaut : 92000
```

## 2.5 – Cas des variables statiques

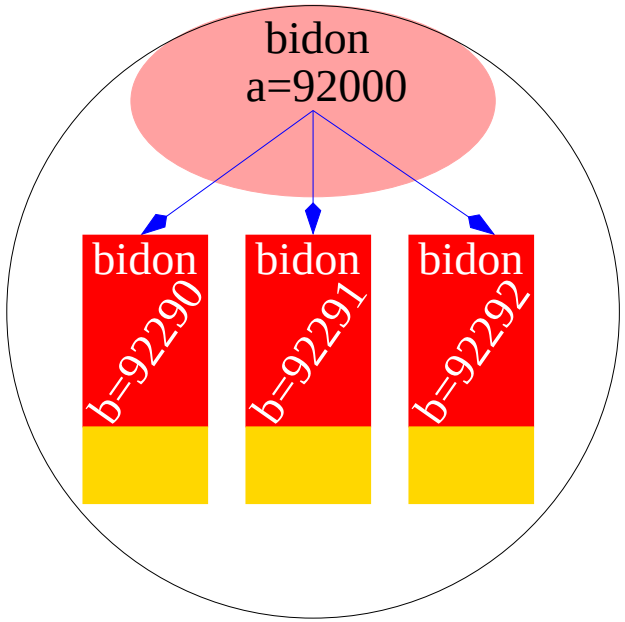
- Une variable est statique si son emplacement en mémoire est défini à la déclaration par le compilateur.
- C'est le cas des variables apparaissant en `COMMON` ou contenues dans un `MODULE` ou déclarées `SAVE` ou initialisées à la déclaration (ex. `PARAMETER, DATA`, etc.).
- Par défaut, une variable statique est une variable partagée.

```
program parallel
 implicit none
 real :: a
 common/bidon/a
 a = 92000.
 !$OMP PARALLEL
   call sub()
 !$OMP END PARALLEL
end program parallel
subroutine sub()
 implicit none
 real :: a, b
 common/bidon/a
 b = a + 290.
 print *,"B vaut : ",b
end subroutine sub
```

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=2;a.out
```

```
B vaut : 92290.
B vaut : 92290.
```

- L'utilisation de la directive THREADPRIVATE permet de privatiser une instance statique et faire que celle-ci soit persistante d'une région parallèle à une autre.
- Si, en outre, la clause COPYIN est spécifiée alors la valeur des instances statiques est transmise à toutes les tâches.




```
program parallel
 use OMP_LIB
 implicit none
 integer :: a
 common/bidon/a
 !$OMP THREADPRIVATE(/bidon/)
 a = 92000
 !$OMP PARALLEL COPYIN(/bidon/)
   a = a + OMP_GET_THREAD_NUM()
   call sub()
 !$OMP END PARALLEL
 print*,"Hors region, A vaut:",a
end program parallel
subroutine sub()
 implicit none
 integer :: a, b
 common/bidon/a
 !$OMP THREADPRIVATE(/bidon/)
 b = a + 290
 print *,"B vaut : ",b
end subroutine sub
```

```
B vaut : 92290
B vaut : 92291
B vaut : 92292
B vaut : 92293
Hors region, A vaut : 92000
```

## 2.6 – Cas de l'allocation dynamique

- L'opération d'allocation/désallocation dynamique de mémoire peut être effectuée à l'intérieur d'une région parallèle.
- Si l'opération porte sur une variable privée, celle-ci sera locale à chaque tâche.
- Si l'opération porte sur une variable partagée, il est alors plus prudent que seule une tâche (ex. la tâche maître) se charge de cette opération.

```
program parallel
  use OMP_LIB
  implicit none
  integer                              :: n,debut,fin,rang,nb_taches,i
  real, allocatable, dimension(:) :: a

  n=1024 ; nb_taches=4
  allocate(a(n*nb_taches))
  !$OMP PARALLEL DEFAULT(NONE) PRIVATE(debut,fin,rang,i) &
        !$OMP SHARED(a,n) IF(n .gt. 512)
     rang=OMP_GET_THREAD_NUM()
     debut=rang*n+1
     fin=(rang+1)*n
     do i = debut, fin
        a(i) = 92290. + real(i)
     end do
     print *,"Rang : ",rang,"; A(",debut,"),...,A(",fin,") : ",a(debut),",...,",A(fin)
  !$OMP END PARALLEL
  deallocate(a)
end program parallel
```

```
Rang : 3 ; A( 3073 ), ... , A( 4096 ) : 95363., ... , 96386.
Rang : 2 ; A( 2049 ), ... , A( 3072 ) : 94339., ... , 95362.
Rang : 1 ; A( 1025 ), ... , A( 2048 ) : 93315., ... , 94338.
Rang : 0 ; A(    1 ), ... , A( 1024 ) : 92291., ... , 93314.
```

## 2.7 – Cas de l'équivalence

- Ne mettre en équivalence que des variables de même statut.
- Dans le cas contraire, le résultat est indéterminé.
- Ces remarques restent vraies dans le cas d'une association par `POINTER`.

```
program parallel
  implicit none
  real :: a, b
  equivalence(a,b)

  a = 92290.
  !$OMP PARALLEL PRIVATE(b) &
        !$OMP SHARED(a)
     print *,"B vaut : ",b
  !$OMP END PARALLEL
end program parallel
```

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4;a.out
```

```
B vaut : -0.3811332074E+30
B vaut : 0.0000000000E+00
B vaut : -0.3811332074E+30
B vaut : 0.0000000000E+00
```

## 2.8 – Compléments

- La construction d'une région parallèle admet deux autres clauses :
  - ⇨ `REDUCTION` : pour les opérations de réduction avec synchronisation implicite entre les tâches ;
  - ⇨ `NUM_THREADS` : Elle permet de spécifier le nombre de tâches souhaité à l'entrée d'une région parallèle de la même manière que le ferait le sous-programme `OMP_SET_NUM_THREADS`.
- D'une région parallèle à l'autre, le nombre de tâches concurrentes peut être variable si on le souhaite. Pour cela, il suffit d'utiliser le sous-programme `OMP_SET_DYNAMIC` ou de positionner la variable d'environnememt `OMP_DYNAMIC` à `true`.

```
program parallel
  implicit none

  !$OMP PARALLEL NUM_THREADS(2)
    print *,"Bonjour !"
  !$OMP END PARALLEL

  !$OMP PARALLEL NUM_THREADS(3)
    print *,"Coucou !"
  !$OMP END PARALLEL
end program parallel
```

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4;
> export OMP_DYNAMIC=true; a.out
```

```
Bonjour !
Bonjour !
Coucou !
Coucou !
Coucou !
```

☞ Il est possible d'imbriquer (*nesting*) des régions parallèles, mais cela n'a d'effet que si ce mode a été activé à l'appel du sous-programme OMP_SET_NESTED ou en positionnant la variable d'environnement OMP_NESTED à la valeur `true`.

```
> xlf_r ... -qsmp=nested_par prog.f90
> export OMP_DYNAMIC=true
> export OMP_NESTED=true; a.out
```

```
program parallel
  use OMP_LIB
  implicit none
  integer :: rang

  !$OMP PARALLEL NUM_THREADS(3) &
        !$OMP PRIVATE(rang)
    rang=OMP_GET_THREAD_NUM()
    print *,"Mon rang dans region 1 :",rang
    !$OMP PARALLEL NUM_THREADS(2) &
          !$OMP PRIVATE(rang)
      rang=OMP_GET_THREAD_NUM()
      print *," Mon rang dans region 2 :",rang
    !$OMP END PARALLEL
  !$OMP END PARALLEL
end program parallel
```

```
 Mon rang dans region 1 : 0
   Mon rang dans region 2 : 1
   Mon rang dans region 2 : 0
 Mon rang dans region 1 : 2
   Mon rang dans region 2 : 1
   Mon rang dans region 2 : 0
 Mon rang dans region 1 : 1
   Mon rang dans region 2 : 0
   Mon rang dans region 2 : 1
```

# Position dans le cours

# 3 – Partage du travail

- En principe, la construction d'une région parallèle et l'utilisation de quelques fonctions `OpenMP` suffisent à eux seuls pour paralléliser une portion de code.
- Mais il est, dans ce cas, à la charge du programmeur de répartir aussi bien le travail que les données et d'assurer la synchronisation des tâches.
- Heureusement, `OpenMP` propose trois directives (`DO`, `SECTIONS` et `WORKSHARE`) qui permettent aisément de contrôler assez finement la répartition du travail et des données en même temps que la synchronisation au sein d'une région parallèle.
- Par ailleurs, il existe d'autres constructions `OpenMP` qui permettent l'exclusion de toutes les tâches à l'exception d'une seule pour exécuter une portion de code située dans une région parallèle.

## 3.1 – Boucle parallèle

- C'est un parallélisme par répartition des itérations d'une boucle.
- La boucle parallélisée est celle qui suit immédiatement la directive `DO`.
- Les boucles « infinies » et `do while` ne sont pas parallélisables avec `OpenMP`.
- Le mode de répartition des itérations peut être spécifié dans la clause `SCHEDULE`.
- Le choix du mode de répartition permet de mieux contrôler l'équilibrage de la charge de travail entre les tâches.
- Les indices de boucles sont des variables entières privées.
- Par défaut, une synchronisation globale est effectuée en fin de construction `END DO` à moins d'avoir spécifié la clause `NOWAIT`.
- Il est possible d'introduire autant de constructions `DO` (les unes après les autres) qu'il est souhaité dans une région parallèle.

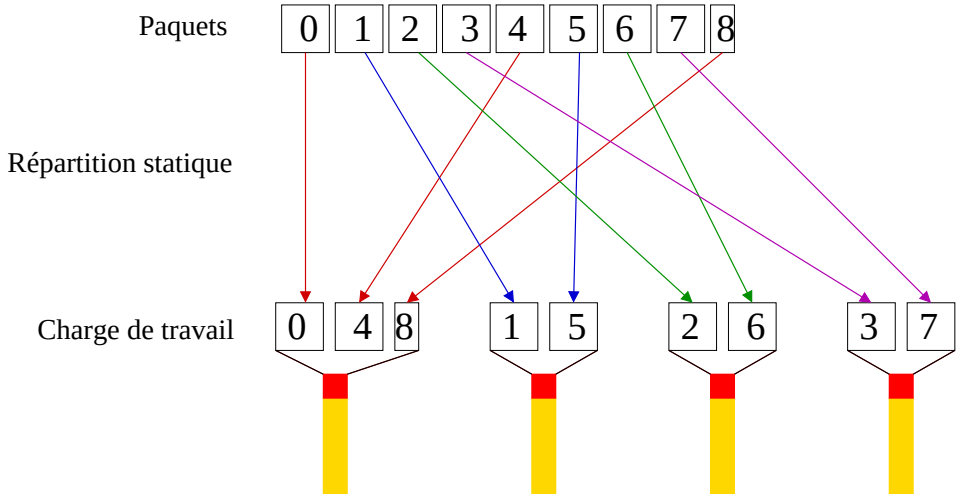
## 3.1.1 – Clause SCHEDULE

```
program parallel
 use OMP_LIB
 implicit none
 integer, parameter :: n=4096
 real, dimension(n) :: a
 integer            :: i, i_min, i_max, rang, nb_taches
 !$OMP PARALLEL PRIVATE(rang,nb_taches,i_min,i_max)
   rang=OMP_GET_THREAD_NUM() ; nb_taches=OMP_GET_NUM_THREADS() ; i_min=n ; i_max=0
   !$OMP DO SCHEDULE(STATIC,n/nb_taches)
     do i = 1, n
       a(i) = 92290. + real(i) ; i_min=min(i_min,i) ; i_max=max(i_max,i)
     end do
   !$OMP END DO NOWAIT
   print *,"Rang : ",rang,"; i_min :",i_min,"; i_max :",i_max
 !$OMP END PARALLEL
end program parallel
```

```
> xlf_r ... -qsmp=omp prog.f90 ; export OMP_NUM_THREADS=4 ; a.out
```

```
Rang : 1 ; i_min : 1025 ; i_max : 2048
Rang : 3 ; i_min : 3073 ; i_max : 4096
Rang : 0 ; i_min :    1 ; i_max : 1024
Rang : 2 ; i_min : 2049 ; i_max : 3072
```

☞ La répartition STATIC consiste à diviser les itérations en paquets d'une taille donnée (sauf peut-être pour le dernier). Il est ensuite attribué, d'une façon cyclique à chacune des tâches, un ensemble de paquets suivant l'ordre des tâches jusqu'à concurrence du nombre total de paquets.



- Nous aurions pu différer à l'exécution le choix du mode de répartition des itérations à l'aide de la variable d'environnement OMP_SCHEDULE.
- Le choix du mode de répartition des itérations d'une boucle peut être un atout majeur pour l'équilibrage de la charge de travail sur une machine dont les processeurs ne sont pas dédiés.
- Attention, pour des raisons de performances vectorielles ou scalaires, éviter de paralléliser les boucles faisant référence à la première dimension d'un tableau multi-dimensionnel.

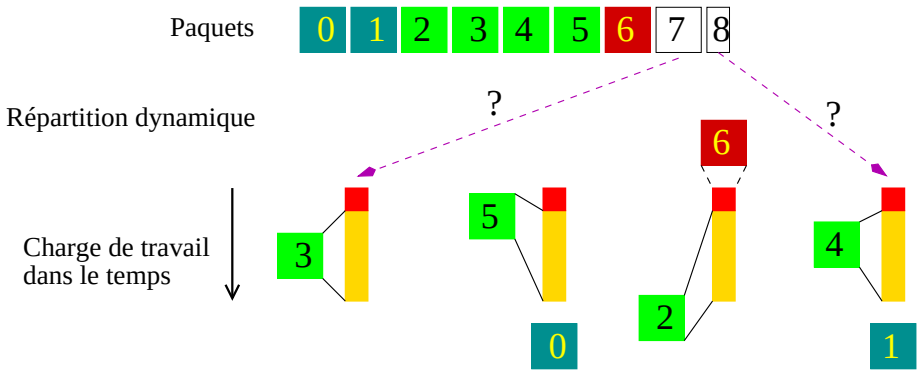
```
program parallel
 use OMP_LIB
 implicit none
 integer, parameter :: n=4096
 real, dimension(n) :: a
 integer :: i, i_min, i_max
 !$OMP PARALLEL DEFAULT(PRIVATE) SHARED(a)
   i_min=n ; i_max=0
   !$OMP DO SCHEDULE(RUNTIME)
      do i = 1, n
        a(i) = 92290. + real(i)
        i_min=min(i_min,i)
        i_max=max(i_max,i)
      end do
   !$OMP END DO
   print*,"Rang:",OMP_GET_THREAD_NUM(), &
          ";i_min:",i_min,";i_max:",i_max
 !$OMP END PARALLEL
end program parallel
```

```
> export OMP_NUM_THREADS=2
> export OMP_SCHEDULE="STATIC,2048"
> a.out
```

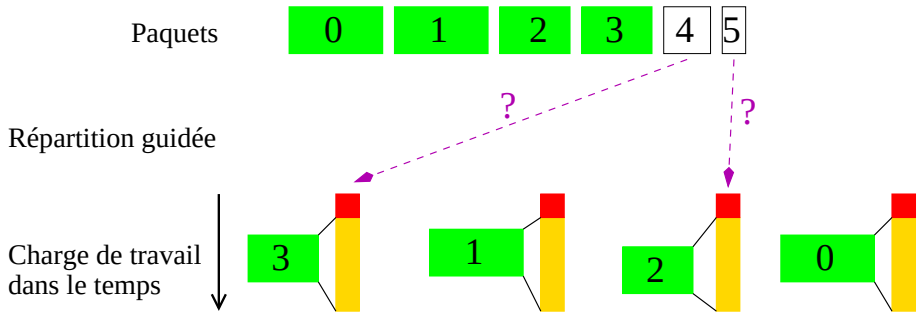
```
Rang: 0 ; i_min:    1 ; i_max: 2048
Rang: 1 ; i_min: 2049 ; i_max: 4096
```

```
> export OMP_SCHEDULE="DYNAMIC,480"
> export OMP_NUM_THREADS=4 ; a.out
```

- En plus du mode **STATIC**, il existe deux autres façons de répartir les itérations d'une boucle :
  - **DYNAMIC** : les itérations sont divisées en paquets de taille donnée. Sitôt qu'une tâche épuise ses itérations, un autre paquet lui est attribué ;
  - **GUIDED** : les itérations sont divisées en paquets dont la taille décroît exponentiellement. Tous les paquets ont une taille inférieure ou égale à une valeur donnée à l'exception du dernier dont la taille peut être inférieure. Sitôt qu'une tâche finit ses itérations, un autre paquet d'itérations lui est attribué.




```
> export OMP_SCHEDULE="GUIDED,256"
> export OMP_NUM_THREADS=4 ; a.out
```



## 3.1.2 – Cas d'une exécution ordonnée

- Il est parfois utile (cas de débogage) d'exécuter une boucle d'une façon ordonnée.
- L'ordre des itérations sera alors identique à celui correspondant à une exécution séquentielle.

```
program parallel
 use OMP_LIB
 implicit none
 integer, parameter :: n=9
 integer            :: i,rang
 !$OMP PARALLEL DEFAULT(PRIVATE)
   rang = OMP_GET_THREAD_NUM()
   !$OMP DO SCHEDULE(RUNTIME) ORDERED
     do i = 1, n
      !$OMP ORDERED
        print *,"Rang :",rang,"; iteration :",i
      !$OMP END ORDERED
     end do
   !$OMP END DO NOWAIT
 !$OMP END PARALLEL
end program parallel
```

```
> export OMP_SCHEDULE="STATIC,2"
> export OMP_NUM_THREADS=4 ; a.out
```

```
 Rang :  0  ; iteration :  1
 Rang :  0  ; iteration :  2
 Rang :  1  ; iteration :  3
 Rang :  1  ; iteration :  4
 Rang :  2  ; iteration :  5
 Rang :  2  ; iteration :  6
 Rang :  3  ; iteration :  7
 Rang :  3  ; iteration :  8
 Rang :  0  ; iteration :  9
```

## 3.1.3 – Cas d'une réduction

- Une réduction est une opération associative appliquée à une variable partagée.
- L'opération peut être :
  - arithmétique : $+$, $-$, $\times$ ;
  - logique : `.AND.`, `.OR.`, `.EQV.`, `.NEQV.` ;
  - une fonction intrinsèque : `MAX`, `MIN`, `IAND`, `IOR`, `IEOR`.
- Chaque tâche calcule un résultat partiel indépendemment des autres. Elles se synchronisent ensuite pour mettre à jour le résultat final.

```
program parallel
 implicit none
 integer, parameter :: n=5
 integer            :: i, s=0, p=1, r=1
 !$OMP PARALLEL
 !$OMP DO REDUCTION(+:s) REDUCTION(*:p,r)
    do i = 1, n
      s = s + 1
      p = p * 2
      r = r * 3
    end do
 !$OMP END PARALLEL
 print *,"s =",s, "; p =",p, "; r =",r
end program parallel
```

```
> export OMP_NUM_THREADS=4
> a.out
```

```
s = 5 ; p = 32 ; r = 243
```

## 3.1.4 – Compléments

☞ Les autres clauses admises dans la directive `DO` sont :

- ➪ `PRIVATE` : pour attribuer à une variable un statut privé ;
- ➪ `FIRSTPRIVATE` : privatise une variable partagée dans l'étendue de la construction `DO` et lui assigne la dernière valeur affectée avant l'entrée dans cette région ;
- ➪ `LASTPRIVATE` : privatise une variable partagée dans l'étendue de la construction `DO` et permet de conserver, à la sortie de cette construction, la valeur calculée par la tâche exécutant la dernière itération d'une boucle.

```
program parallel
 use OMP_LIB
 implicit none
 integer, parameter :: n=9
 integer            :: i, rang
 real               :: temp

 !$OMP PARALLEL PRIVATE(rang)
   !$OMP DO LASTPRIVATE(temp)
      do i = 1, n
        temp = real(i)
      end do
   !$OMP END DO
   rang=OMP_GET_THREAD_NUM()
   print *,"Rang:",rang,";temp=",temp
 !$OMP END PARALLEL
end program parallel
```

```
> export OMP_NUM_THREADS=4 ; a.out
```

```
Rang : 2 ; temp= 9.0
Rang : 3 ; temp= 9.0
Rang : 1 ; temp= 9.0
Rang : 0 ; temp= 9.0
```

- La directive `PARALLEL DO` est une fusion des directives `PARALLEL` et `DO` munie de l'union de leurs clauses respectives.
- La directive de terminaison `END PARALLEL DO` inclut une barrière globale de synchronisation et ne peut admettre la clause `NOWAIT`.

```
program parallel
 implicit none
 integer, parameter :: n=9
 integer            :: i
 real               :: temp

 !$OMP PARALLEL DO LASTPRIVATE(temp)
   do i = 1, n
     temp = real(i)
   end do
 !$OMP END PARALLEL DO
end program parallel
```

## 3.2 – Sections parallèles

- 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.
- La clause `NOWAIT` est admise en fin de construction `END SECTIONS` pour lever la barrière de synchronisation implicite.

## 3.2.1 – Construction SECTIONS

```
program parallel
 implicit none
 integer, parameter    :: n=513, m=4097
 real, dimension(m,n)  :: a, b
 real, dimension(m)    :: coord_x
 real, dimension(n)    :: coord_y
 real                  :: pas_x, pas_y
 integer               :: i

 !$OMP PARALLEL
  !$OMP SECTIONS
   !$OMP SECTION
   call lecture_champ_initial_x(a)
   !$OMP SECTION
   call lecture_champ_initial_y(b)
   !$OMP SECTION
   pas_x      = 1./real(m-1)
   pas_y      = 2./real(n-1)
   coord_x(:) = (/ (real(i-1)*pas_x,i=1,m) /)
   coord_y(:) = (/ (real(i-1)*pas_y,i=1,n) /)
  !$OMP END SECTIONS NOWAIT
 !$OMP END PARALLEL
end program parallel
```

```
subroutine lecture_champ_initial_x(x)
 implicit none
 integer, parameter    :: n=513, m=4097
 real, dimension(m,n) :: x

 call random_number(x)
end subroutine lecture_champ_initial_x


subroutine lecture_champ_initial_y(y)
 implicit none
 integer, parameter    :: n=513, m=4097
 real, dimension(m,n) :: y

 call random_number(y)
end subroutine lecture_champ_initial_y
```

## 3.2.2 – Compléments

- Toutes les directives `SECTION` doivent apparaître dans l'étendue lexicale de la construction `SECTIONS`.
- Les clauses admises dans la directive `SECTIONS` sont celles que nous connaissons déja :
  - `PRIVATE` ;
  - `FIRSTPRIVATE` ;
  - `LASTPRIVATE` ;
  - `REDUCTION`.
- La directive `PARALLEL SECTIONS` est une fusion des directives `PARALLEL` et `SECTIONS` munie de l'union de leurs clauses respectives.
- La directive de terminaison `END PARALLEL SECTIONS` inclut une barrière globale de synchronisation et ne peut admettre la clause `NOWAIT`.

## 3.3 – Construction WORKSHARE

- Elle ne peut être spécifée qu'au sein d'une région parallèle.
- Elle est utile pour répartir le travail essentiellement lié à certaines constructions Fortran 95 telles que les :
  - instructions sur des variables scalaires ;
  - instructions sur des variables tableaux ;
  - instructions ou blocs FORALL et WHERE ;
  - fonctions dites « ELEMENTAL » définies par l'utilisateur.
- Elle n'admet aucune autre clause que NOWAIT aussi bien en début (WORKSHARE) qu'en fin de construction (END WORKSHARE).

- Seules les instructions ou blocs Fortran 95 spécifiés dans l'étendue lexicale verront leur travail réparti entre les tâches.
- L'unité de travail est l'élément d'un tableau. Il n'existe aucun moyen de changer ce comportement par défaut.
- Les surcoûts liés à une telle répartition du travail peuvent être importants.

```
program parallel
 implicit none
 integer, parameter   :: m=4097, n=513
 integer              :: i, j
 real, dimension(m,n) :: a, b

 call random_number(b)
 a(:,:) = 1.
 !$OMP PARALLEL
   !$OMP DO
     do j=1,n
       do i=1,m
          b(i,j) = b(i,j) - 0.5
       end do
     end do
   !$OMP END DO
   !$OMP WORKSHARE NOWAIT
     WHERE(b(:,:) >= 0.) a(:,:)=sqrt(b(:,:))
   !$OMP END WORKSHARE NOWAIT
 !$OMP END PARALLEL
end program parallel
```

☞ La construction PARALLEL WORKSHARE est une fusion des constructions PARALLEL et WORKSHARE munie de l'union de leurs clauses et de leurs contraintes respectives à l'exception de la clause NOWAIT en fin de construction.

```
program parallel
 implicit none
 integer, parameter   :: m=4097, n=513
 real, dimension(m,n) :: a, b

 call random_number(b)
 !$OMP PARALLEL WORKSHARE
   a(:,:) = 1.
   b(:,:) = b(:,:) - 0.5
   WHERE(b(:,:) >= 0.) a(:,:)=sqrt(b(:,:))
 !$OMP END PARALLEL WORKSHARE
end program parallel
```

## 3.4 – Exécution exclusive

- 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`.
- Bien que le but recherché soit le même, le comportement induit par ces deux constructions reste fondamentalement différent.

## 3.4.1 – Construction 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 END SINGLE, la terminaison de celle qui en a la charge, à moins d'avoir spécifié la clause NOWAIT.

```
program parallel
 use OMP_LIB
 implicit none
 integer :: rang
 real    :: a

 !$OMP PARALLEL DEFAULT(PRIVATE)
  a = 92290.

  !$OMP SINGLE
    a = -92290.
  !$OMP END SINGLE

  rang = OMP_GET_THREAD_NUM()
  print *,"Rang :",rang,"; A vaut :",a
 !$OMP END PARALLEL
end program parallel
```

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4 ; a.out
```

```
Rang : 1 ; A vaut :  92290.
Rang : 2 ; A vaut :  92290.
Rang : 0 ; A vaut :  92290.
Rang : 3 ; A vaut : -92290.
```

- Une clause supplémentaire admise uniquement par la directive de terminaison END SINGLE est la clause COPYPRIVATE.
- Elle permet à la tâche chargée d'exécuter la région SINGLE, de diffuser aux autres tâches la valeur d'une liste de variables privées avant de sortir de cette région.
- Les autres clauses admises par la directive SINGLE sont PRIVATE et FIRSTPRIVATE.

```
program parallel
 use OMP_LIB
 implicit none
 integer :: rang
 real    :: a

 !$OMP PARALLEL DEFAULT(PRIVATE)
  a = 92290.

 !$OMP SINGLE
    a = -92290.
 !$OMP END SINGLE COPYPRIVATE(a)

  rang = OMP_GET_THREAD_NUM()
  print *,"Rang :",rang,"; A vaut :",a
 !$OMP END PARALLEL
end program parallel
```

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4 ; a.out
```

```
Rang : 1 ; A vaut : -92290.
Rang : 2 ; A vaut : -92290.
Rang : 0 ; A vaut : -92290.
Rang : 3 ; A vaut : -92290.
```

## 3.4.2 – Construction 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 (END MASTER).

```
program parallel
 use OMP_LIB
 implicit none
 integer :: rang
 real    :: a

 !$OMP PARALLEL DEFAULT(PRIVATE)
  a = 92290.

  !$OMP MASTER
    a = -92290.
  !$OMP END MASTER

  rang = OMP_GET_THREAD_NUM()
  print *,"Rang :",rang,"; A vaut :",a
 !$OMP END PARALLEL
end program parallel
```

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4 ; a.out
```

```
Rang : 0 ; A vaut : -92290.
Rang : 3 ; A vaut :  92290.
Rang : 2 ; A vaut :  92290.
Rang : 1 ; A vaut :  92290.
```

## 3.5 – Procédures orphelines

- Une procédure (fonction ou sous-programme) appelée dans une région parallèle est exécutée par toutes les tâches.
- En général, il n'y a aucun intérêt à cela si le travail de la procédure n'est pas distribué.
- Cela nécessite l'introduction de directives OpenMP (DO, SECTIONS, etc.) dans le corps de la procédure si celle-ci est appelée dans une région parallèle.
- Ces directives sont dites « orphelines » et, par abus de langage, on parle alors de procédures orphelines (*orphaning*).
- Une bibliothèque scientifique multi-tâches avec OpenMP sera constituée d'un ensemble de procédures orphelines.

```
> ls
> mat_vect.f90 prod_mat_vect.f90
```

```
program mat_vect
 implicit none
 integer,parameter   :: n=1025
 real,dimension(n,n) :: a
 real,dimension(n)   :: x, y
 call random_number(a)
 call random_number(x) ; y(:)=0.
 !$OMP PARALLEL IF(n.gt.256)
 call prod_mat_vect(a,x,y,n)
 !$OMP END PARALLEL
end program mat_vect
```

```
subroutine prod_mat_vect(a,x,y,n)
 implicit none
 integer,intent(in) :: n
 real,intent(in),dimension(n,n) :: a
 real,intent(in),dimension(n)   :: x
 real,intent(out),dimension(n)  :: y
 integer :: i
 !$OMP DO
 do i = 1, n
   y(i) = SUM(a(i,:) * x(:))
 end do
 !$OMP END DO
end subroutine prod_mat_vect
```

- Attention, car il existe trois contextes d'exécution selon le mode de compilation des unités de programme appelantes et appelées :
  - la directive `PARALLEL` de l'unité appelante est interprétée (l'exécution peut être Parallèle) à la compilation ainsi que les directives de l'unité appelée (le travail peut être Distribué) ;
  - la directive `PARALLEL` de l'unité appelante est interprétée à la compilation (l'exécution peut être Parallèle) mais pas les directives contenues dans l'unité appelée (le travail peut être Répliqué) ;
  - la directive `PARALLEL` de l'unité appelante n'est pas interprétée à la compilation. L'exécution est partout Séquentielle même si les directives contenues dans l'unité appelée ont été interprétées à la compilation.

| unité appelante compilée \ unité appelée compilée | avec `-qsmp=omp` | sans `-qsmp=omp` |
|---|---|---|
| avec `-qsmp=omp` | P + D | P + R |
| sans `-qsmp=omp` | S | S |

TAB. 1 – Contexte d'exécution selon le mode de compilation (exemple avec des options du compilateur `Fortran` d'`IBM`)

## 3.6 – Récapitulatif

| | default | shared | private | firstprivate | lastprivate | copyprivate | if | reduction | schedule | ordered | copyin | nowait |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| parallel | ✓ | ✓ | ✓ | ✓ | | | ✓ | ✓ | | | ✓ | |
| do | | | ✓ | ✓ | ✓ | | | ✓ | ✓ | ✓ | | ✓ |
| sections | | | ✓ | ✓ | ✓ | | | ✓ | | | | ✓ |
| workshare | | | | | | | | | | | | ✓ |
| single | | | ✓ | ✓ | | ✓ | | | | | | ✓ |
| master | | | | | | | | | | | | |
| threadprivate | | | | | | | | | | | | |

# Position dans le cours

# 4 – Synchronisations

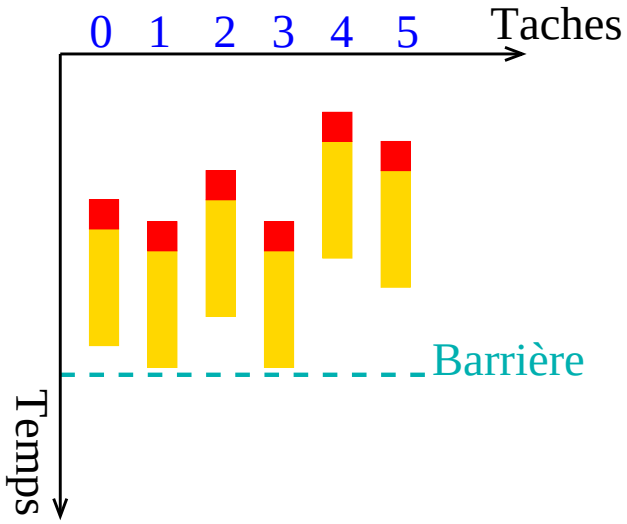
La synchronisation devient nécessaire dans les situations suivantes :

1. pour s'assurer que toutes les tâches concurrentes aient atteint un même niveau d'instruction dans le programme (barrière globale) ;
2. pour ordonner l'exécution de toutes les tâches concurrentes quand celles-ci doivent exécuter une même portion de code affectant une ou plusieurs variables partagées dont la cohérence (en lecture ou en écriture) en mémoire doit être garantie (exclusion mutuelle).
3. pour synchroniser au moins deux tâches concurrentes parmi l'ensemble (mécanisme de verrous).

- Comme nous l'avons déja indiqué, l'absence de clause `NOWAIT` signifie qu'une barrière globale de synchronisation est implicitement appliquée en fin de construction `OpenMP`. Mais il est possible d'imposer explicitement une barrière globale de synchronisation grâce à la directive `BARRIER`.
- Le mécanisme d'exclusion mutuelle (une tâche à la fois) se trouve, par exemple, dans les opérations de réduction (clause `REDUCTION`) ou dans l'exécution ordonnée d'une boucle (directive `DO ORDERED`). Dans le même but, ce mécanisme est aussi mis en place dans les directives `ATOMIC` et `CRITICAL`.
- Des synchronisations plus fines peuvent être réalisées soit par la mise en place des mécanismes de verrous (cela nécessite l'appel à des sous-programmes de la bibliothèque `OpenMP`), soit par l'utilisation de la directive `FLUSH`.

## 4.1 – Barrière

- 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.




```
program parallel
 implicit none
 real,allocatable,dimension(:) :: a, b
 integer                       :: n, i
 n = 5
 !$OMP PARALLEL
  !$OMP SINGLE
    allocate(a(n),b(n))
  !$OMP END SINGLE
  !$OMP MASTER
    read(9) a(1:n)
  !$OMP END MASTER
  !$OMP BARRIER
  !$OMP DO SCHEDULE(STATIC)
    do i = 1, n
      b(i) = 2.*a(i)
    end do
  !$OMP SINGLE
    deallocate(a)
  !$OMP END SINGLE NOWAIT
 !$OMP END PARALLEL
 print *, "B vaut : ", b(1:n)
end program parallel
```

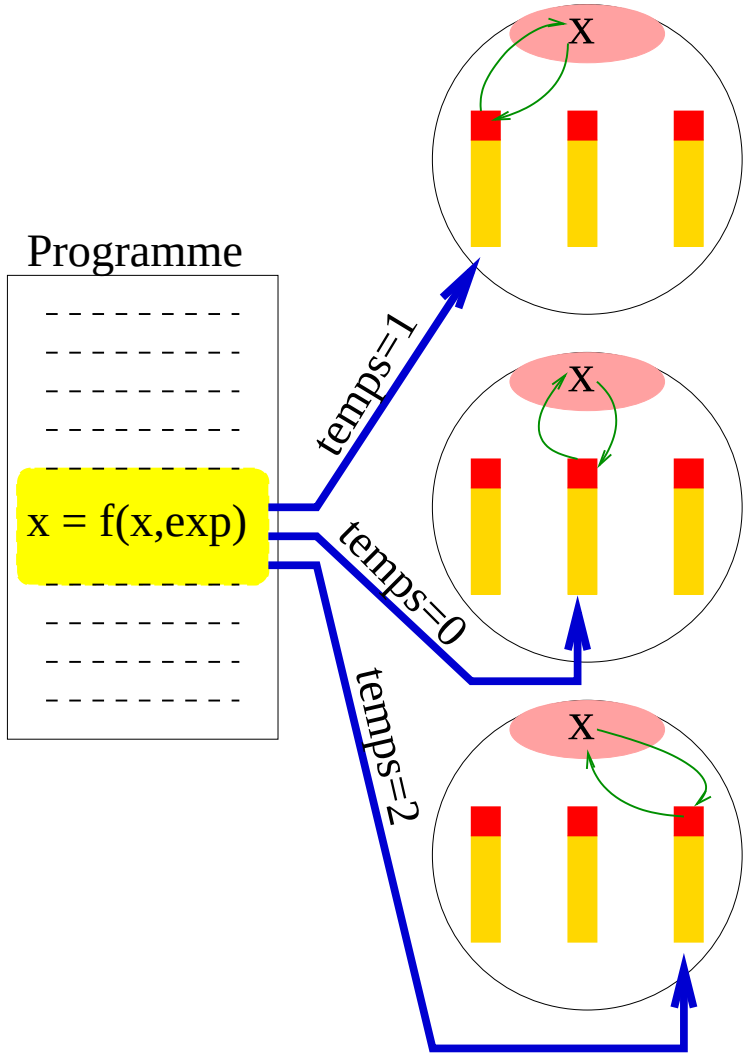
## 4.2 – Mise à jour atomique

- 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.

```
program parallel
 use OMP_LIB
 implicit none
 integer :: compteur, rang
 compteur = 92290
 !$OMP PARALLEL PRIVATE(rang)
  rang = OMP_GET_THREAD_NUM()
  !$OMP ATOMIC
   compteur = compteur + 1

  print *,"Rang :",rang, &
     "; compteur vaut :",compteur
 !$OMP END PARALLEL
 print *,"Au total, compteur vaut :", &
          compteur
end program parallel
```

```
Rang : 0 ; compteur vaut : 92291
Rang : 1 ; compteur vaut : 92292
Rang : 2 ; compteur vaut : 92293
Rang : 3 ; compteur vaut : 92294
Au total, compteur vaut : 92294
```

- L'instruction en question doit avoir l'une des formes suivantes :
  - `x=x (op) exp;`
  - `x=exp (op) x;`
  - `x=f(x,exp);`
  - `x=f(exp,x).`
- `(op)` représente l'une des opérations suivantes : `+`, `--`, $\times$, `/`, `.AND.`, `.OR.`, `.EQV.`, `.NEQV.`.
- `f` représente une des fonctions intrinsèques suivantes : `MAX`, `MIN`, `IAND`, `IOR`, `IEOR`.
- `exp` est une expression arithmétique quelconque indépendante de `x`.



## 4.3 – Régions critiques

- Une région critique peut être vue comme une généralisation de la directive ATOMIC bien que les mécanismes sous-jacents soient distincts.
- Les tâches exécutent cette région dans un ordre non-déterministe mais une à la fois.
- Une région critique est définie grâce à la directive CRITICAL et s'applique à une portion de code terminée par END CRITICAL.
- Son étendue est dynamique.
- Pour des raisons de performances, il est déconseillé d'émuler une instruction atomique par une région critique.

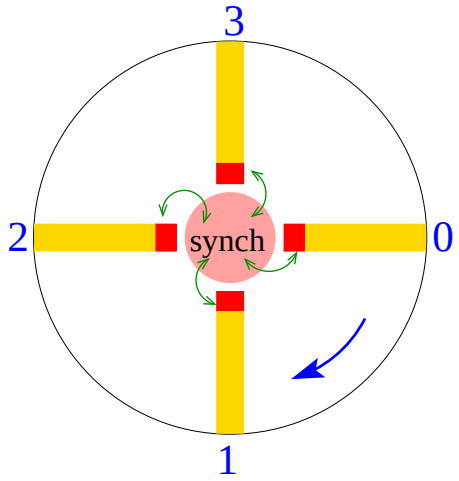
```
program parallel
 implicit none
 integer :: s, p

 s=0 ; p=1
 !$OMP PARALLEL
   !$OMP CRITICAL
     s = s + 1
     p = p * 2
   !$OMP END CRITICAL
 !$OMP END PARALLEL
 print *, "s=", s, " ; p=", p
end program parallel
```

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4 ; a.out
```

```
s= 4 ; p= 16
```

## 4.4 – Directive FLUSH

- Elle est utile dans une région parallèle pour rafraîchir la valeur d'une variable partagée en mémoire globale.
- Elle est d'autant plus utile que la mémoire d'une machine est hiérarchisée.
- Elle peut servir à mettre en place un mécanisme de point de synchronisation entre les tâches.




```
program anneau
 use OMP_LIB
 implicit none
 integer :: rang,nb_taches,synch=0
 !$OMP PARALLEL PRIVATE(rang,nb_taches)
   rang=OMP_GET_THREAD_NUM()
   nb_taches=OMP_GET_NUM_THREADS()
   if (rang == 0) then ; do
        !$OMP FLUSH(synch)
        if(synch == nb_taches-1) exit
     end do
   else ; do
        !$OMP FLUSH(synch)
        if(synch == rang-1) exit
     end do
   end if
   print *,"Rang:",rang,";synch:",synch
   synch=rang
   !$OMP FLUSH(synch)
 !$OMP END PARALLEL
end program anneau
```

```
Rang :   1   ; synch :   0
Rang :   2   ; synch :   1
Rang :   3   ; synch :   2
Rang :   0   ; synch :   3
```

## 4.5 – Récapitulatif

| | default | shared | private | firstprivate | lastprivate | copyprivate | if | reduction | schedule | ordered | copyin | nowait |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| parallel | ✓ | ✓ | ✓ | ✓ | | | ✓ | ✓ | | | ✓ | |
| do | | | ✓ | ✓ | ✓ | | | ✓ | ✓ | ✓ | | ✓ |
| sections | | | ✓ | ✓ | ✓ | | | ✓ | | | | ✓ |
| workshare | | | | | | | | | | | | ✓ |
| single | | | ✓ | ✓ | | ✓ | | | | | | ✓ |
| master | | | | | | | | | | | | |
| threadprivate | | | | | | | | | | | | |
| atomic | | | | | | | | | | | | |
| critical | | | | | | | | | | | | |
| flush | | | | | | | | | | | | |

# Position dans le cours

## 5 – Quelques pièges

☞ Dans le première exemple ci-contre, le statut de la variable « s » est erroné ce qui produit un résultat indéterminé. En effet, le statut de « s » doit être SHARED dans l'étendue lexicale de la région parallèle si la clause LASTPRIVATE est spécifiée dans la directive DO (ce n'est pas la seule clause dans ce cas là). Ici, les deux implémentations, IBM et NEC, fournissent deux résultats différents. Pourtant, ni l'une ni l'autre n'est en contradiction avec la norme alors qu'un seul des résultats est correct.

```
program faux_1
 ...
 real                   :: s
 real, dimension(9) :: a
 a(:) = 92290.
 !$OMP PARALLEL DEFAULT(PRIVATE) &
       !$OMP SHARED(a)
   !$OMP DO LASTPRIVATE(s)
      do i = 1, n
        s = a(i)
      end do
   !$OMP END DO
   print *, "s=",s,"; a(9)=",a(n)
 !$OMP END PARALLEL
end program faux_1
```

```
IBM SP-3> export OMP_NUM_THREADS=3;a.out
s=92290.  ; a( 9 )=92290.
s=0.      ; a( 9 )=92290.
s=0.      ; a( 9 )=92290.
```

```
NEC SX-5> export OMP_NUM_THREADS=3;a.out
s=92290.  ; a( 9 )=92290.
s=92290.  ; a( 9 )=92290.
s=92290.  ; a( 9 )=92290.
```

- Dans le second exemple ci-contre, il se produit un effet de course entre les tâches qui fait que l'instruction « `print` » n'imprime pas le résultat escompté de la variable « s » dont le statut est `SHARED`. Il se trouve ici que `NEC` et `IBM` fournissent des résultats identiques mais il est possible et légitime d'obtenir un résultat différent sur d'autres plateformes. La solution est de glisser, par exemple, une directive `BARRIER` juste après l'instruction « `print` ».

```
program faux_2
 implicit none
 real    :: s
 !$OMP PARALLEL DEFAULT(NONE) &
       !$OMP SHARED(s)
  !$OMP SINGLE
   s=1.
  !$OMP END SINGLE
  print *,"s = ",s
  s=2.
 !$OMP END PARALLEL
end program faux_2
```

```
IBM SP-3> export OMP_NUM_THREADS=3;a.out
s =  1.0
s =  2.0
s =  2.0
```

```
NEC SX-5> export OMP_NUM_THREADS=3;a.out
s =  1.0
s =  2.0
s =  2.0
```

# Position dans le cours

# 6 – Performances

- En général, les performances dépendent de l'architecture (processeurs, liens d'interconnexion et mémoire) de la machine et de l'implémentation `OpenMP` utilisée.
- Il existe, néanmoins, quelques règles de « bonnes performances » indépendantes de l'architecture.
- En phase d'optimisation avec `OpenMP`, l'objectif sera de réduire le temps de restitution du code et d'estimer son accélération par rapport à une exécution séquentielle.

## 6.1 – Règles de bonnes performances

- Minimiser le nombre de régions parallèles dans le code.
- Adapter le nombre de tâches demandé à la taille du problème à traiter afin de minimiser les surcoûts de gestion des tâches par le système.
- Dans la mesure du possible, paralléliser la boucle la plus externe.
- Utiliser la clause `SCHEDULE(RUNTIME)` pour pouvoir changer dynamiquement l'ordonnancement et la taille des paquets d'itérations dans une boucle.
- La directive `SINGLE` et la clause `NOWAIT` peuvent permettre de baisser le temps de restitution au prix, le plus souvent, d'une synchronisation explicite.
- La directive `ATOMIC` et la clause `REDUCTION` sont plus restrictives mais plus performantes que la directive `CRITICAL`.

- Utiliser la clause **IF** pour mettre en place une parallélisation conditionnelle (ex. sur une architecture vectorielle, ne paralléliser une boucle que si sa longueur est suffisament grande).
- Éviter de paralléliser la boucle faisant référence à la première dimension des tableaux (en `Fortran`) car c'est celle qui fait référence à des éléments contigus en mémoire.

```
program parallel
 implicit none
 integer, parameter    :: n=1025
 real, dimension(n,n) :: a, b
 integer               :: i, j

 call random_number(a)

!$OMP PARALLEL DO SCHEDULE(RUNTIME)&
        !$OMP IF(n.gt.514)
   do j = 2, n-1
     do i = 1, n
        b(i,j) = a(i,j+1) - a(i,j-1)
     end do
   end do
!$OMP END PARALLEL DO
end program parallel
```

- Les conflits inter-tâches (de banc mémoire sur une machine vectorielle ou de défauts de cache sur une machine scalaire), peuvent dégrader sensiblement les performances.
- Sur une machine multi-nœuds à mémoire virtuellement partagée (ex. `SGI-O2000`), les accès mémoire (type `NUMA` : *Non Uniform Memory Access*) sont moins rapides que des accès intra-nœuds.
- Indépendamment de l'architecture des machines, la qualité de l'implémentation `OpenMP` peut affecter assez sensiblement l'extensibilité des boucles parallèles.

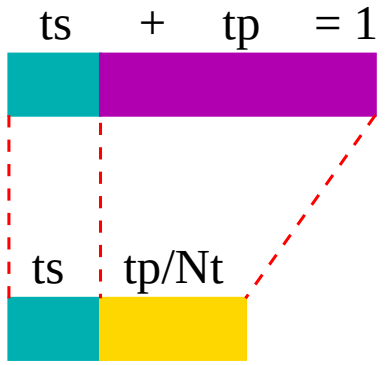
## 6.2 – Mesures du temps

- OpenMP offre deux fonctions :
  - OMP_GET_WTIME pour mesurer le temps de restitution en secondes ;
  - OMP_GET_WTICK pour connaître la précision des mesures en secondes.
- Ce que l'on mesure est le temps écoulé depuis un point de référence arbitraire du code.
- Cette mesure peut varier d'une exécution à l'autre selon la charge de la machine et la répartition des tâches sur les processeurs.

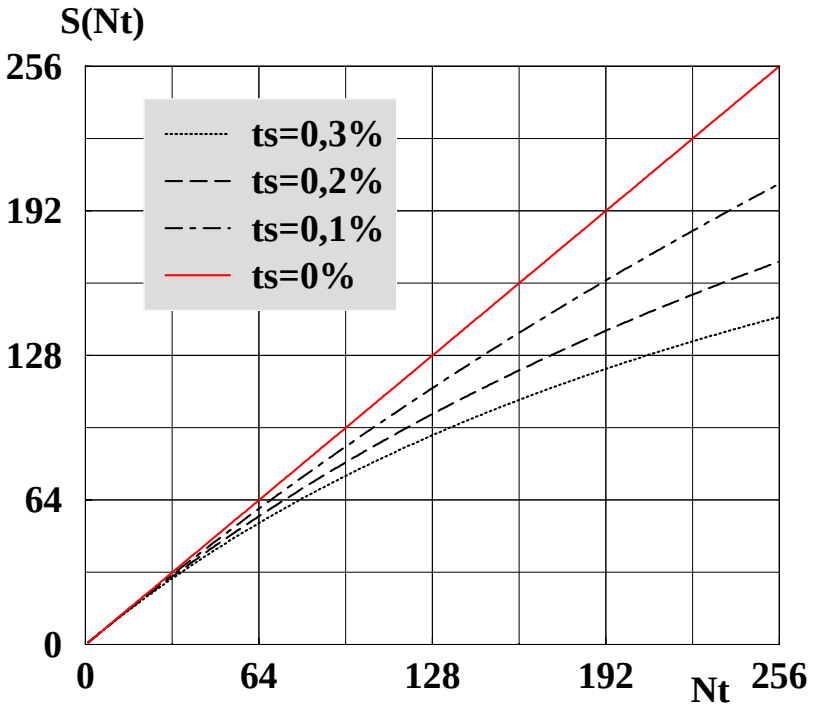
```
program mat_vect
 use OMP_LIB
 implicit none
 integer,parameter    :: n=1025
 real,dimension(n,n) :: a
 real,dimension(n)    :: x, y
 real(kind=8) :: t_ref, t_final
 integer :: rang
 call random_number(a)
 call random_number(x) ; y(:)=0.
 !$OMP PARALLEL &
  !$OMP PRIVATE(rang,t_ref,t_final)
  rang = OMP_GET_THREAD_NUM()
  t_ref=OMP_GET_WTIME()
  call prod_mat_vect(a,x,y,n)
  t_final=OMP_GET_WTIME()
  print *,"Rang :",rang, &
          "; Temps :",t_final-t_ref
 !$OMP END PARALLEL
end program mat_vect
```

## 6.3 – Accélération

- Le gain en performance d'un code parallèle est estimé par rapport à une exécution séquentielle.
- Le rapport entre le temps séquentiel $T_s$ et le temps parallèle $T_p$ sur une machine dédiée est déjà un bon indicateur sur le gain en performance. Celui-ci définit l'accélération $S(N_t)$ du code qui dépend du nombre de tâches $N_t$.
- Si l'on considère $T_s = t_s + t_p = 1$ ($t_s$ représente le temps relatif à la partie séquentielle et $t_p$ celui relatif à la partie parallélisable du code), la loi dite de « AMDHAL » $S(N_t) = \frac{1}{t_s + \frac{t_p}{N_t}}$ indique que l'accélération $S(N_t)$ est majorée par la fraction séquentielle $\frac{1}{t_s}$ du programme.




$$S(Nt) = \frac{1}{ts + \frac{tp}{Nt}}$$



# Position dans le cours

# 7 – Conclusion

- Nécessite une machine multi-processeurs à mémoire partagée.
- Mise en œuvre relativement facile, même dans un programme à l'origine séquentiel.
- Permet la parallélisation progressive d'un programme séquentiel.
- Tout le potentiel des performances parallèles se trouve dans les régions parallèles.
- Au sein de ces régions parallèles, le travail peut être partagé grâce aux boucles et aux sections parallèles. Mais on peut aussi singulariser une tâche pour un travail particulier.
- Les directives orphelines permettent de développer des procédures parallèles.
- Des synchronisations explicites globales ou point à point sont parfois nécessaires dans les régions parallèles.
- Un soin tout particulier doit être apporté à la définition du statut des variables utilisées dans une construction.
- L'accélération mesure l'extensibilité d'un code. Elle est majorée par la fraction séquentielle du programme et est ralentie par les surcoûts liés à la gestion des tâches.

# 8 – Annexes

Ce que nous n'avons pas (ou que peu) abordé dans ce cours :

- les procédures verrous pour la synchronisation point à point ;
- d'autres sous-programmes de service ;
- la parallélisation mixte MPI & OpenMP.

# Index des directives et des clauses OpenMP