OpenMP

Parallélisation multitâches pour machines à mémoire partagée

Intervenants :
Rémi Lacroix
Thibaut Véry
<prenom.nom@idris.fr>

Auteurs :
Jalel Chergui
Pierre-François Lavallée

Copyright © 2001-2020 CNRS/IDRIS

1 – Introduction 8
1.1 – Historique 9
1.2 – Spécifications `OpenMP` 10
1.3 – Terminologie et définitions 11
1.4 – Concepts généraux 12
1.4.1 – Modèle d'exécution 12
1.4.2 – Processus légers (*threads*) 13
1.5 – Fonctionnalités 16
1.6 – OpenMP versus MPI 17
1.7 – Bibliographie 19
2 – Principes 20
2.1 – Interface de programmation 20
2.1.1 – Syntaxe générale d'une directive 21
2.1.2 – Compilation 23
2.2 – Construction d'une région parallèle 24
2.3 – Statut des variables 26
2.3.1 – Variables privées 26
2.3.2 – La clause DEFAULT 28

2.3.3 – Allocation dynamique 29
2.3.4 – Équivalence entre variables Fortran 31
2.4 – Étendue d'une région parallèle 32
2.5 – Transmission par arguments 34
2.6 – Variables statiques 35
2.7 – Compléments 37
3 – Partage du travail 39
3.1 – Introduction 39
3.2 – Boucle parallèle 40
3.2.1 – Clause SCHEDULE 41
3.2.2 – Cas d'une exécution ordonnée 45
3.2.3 – Cas d'une réduction 46
3.2.4 – Cas de la fusion d'un nid de boucles 47
3.2.5 – Compléments 49
3.3 – Construction WORKSHARE 51
3.4 – Sections parallèles 54
3.4.1 – Construction SECTIONS 55
3.4.2 – Compléments 56

3.5 – Exécution exclusive 57
3.5.1 – Construction SINGLE 58
3.5.2 – Construction MASTER 60
3.6 – Procédures orphelines 61
3.7 – Récapitulatif 63
4 – Synchronisations 64
4.1 – Barrière 66
4.2 – Mise à jour atomique 67
4.3 – Régions critiques 69
4.4 – Directive FLUSH 71
4.4.1 – Exemple avec un piège facile 72
4.4.2 – Exemple avec un piège difficile 73
4.4.3 – Commentaires sur les codes précédents 74
4.4.4 – Code corrigé 75
4.4.5 – Nid de boucles avec double dépendance 76
4.5 – Récapitulatif 82
5 – Vectorisation SIMD 83
5.1 – Introduction 83

5.2 – Vectorisation SIMD d'une boucle . . . 84
5.3 – Parallélisation et vectorisation SIMD d'une boucle . . . 85
5.4 – Vectorisation SIMD de fonctions scalaires . . . 86
6 – Les tâches OpenMP . . . 87
6.1 – Introduction . . . 87
6.2 – Les bases du concept . . . 88
6.3 – Le modèle d'exécution des tâches . . . 89
6.4 – Quelques exemples . . . 91
6.5 – Dépendance entre tâches . . . 95
6.6 – Statut des variables dans les tâches . . . 97
6.7 – Exemple de MAJ des éléments d'une liste chaînée . . . 98
6.8 – Exemple d'algorithme récursif . . . 99
6.9 – Clauses `FINAL` et `MERGEABLE` . . . 100
6.10 – Synchronisation de type `TASKGROUP` . . . 101
7 – Affinités . . . 103
7.1 – Affinité des threads . . . 103
7.1.1 – Commande *cpuinfo* . . . 104
7.1.2 – Utilisation de la variable d'environnement *KMP_AFFINITY* . . . 106

7.1.3 – Affinité des threads avec OpenMP 4.0 108
7.2 – Affinité mémoire 110
7.3 – Stratégie « *First Touch* » 113
7.4 – Exemples d'impact sur les performances 114
8 – Performances 118
8.1 – Règles de bonnes performances 119
8.2 – Mesures du temps 122
8.3 – Accélération 123
9 – Conclusion 124
10 – Annexes 125
10.1 – Parties non abordées ici 125
10.2 – Quelques pièges 126

1 – Introduction

- OpenMP est un modèle de programmation parallèle qui initalement ciblait uniquement les architectures à mémoire partagée. Aujourd'hui, il cible aussi les accélérateurs, les systèmes embarqués et les systèmes temps réel.
- Les tâches de calcul peuvent accéder à un espace mémoire commun, ce qui limite la redondance des données et simplifie les échanges d'information entre les tâches.
- En pratique, la parallélisation repose sur l'utilisation de processus système légers (ou *threads*), on parle alors de programme *multithreadé*.

1.1 – Historique

- La parallélisation multithreadée 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.

1.2 – Spécifications OpenMP

- Une version OpenMP 2 a été finalisée en novembre 2000. Elle apporte surtout des extensions relatives à la parallélisation de certaines constructions Fortran 95.
- La version OpenMP 3 datant de mai 2008 introduit essentiellement le concept de tâche.
- Les versions OpenMP 4 de juillet 2013 puis OpenMP 4.5 de novembre 2015 apportent de nombreuses nouveautés, avec notamment le support des accélérateurs, des dépendances entre les tâches, la programmation SIMD (vectorisation) et l'optimisation du placement des *threads*.
- Les version OpenMP 5 de novembre 2018 puis OpenMP 5.1 de novembre 2020 se concontrent principalement sur l'amélioration du support des accélérateurs. Elles apportent également des améliorations sur les tâches, la gestion des mémoires non uniformes et le support des versions récentes des langages C (11), C++ (17) et Fortran (2008).

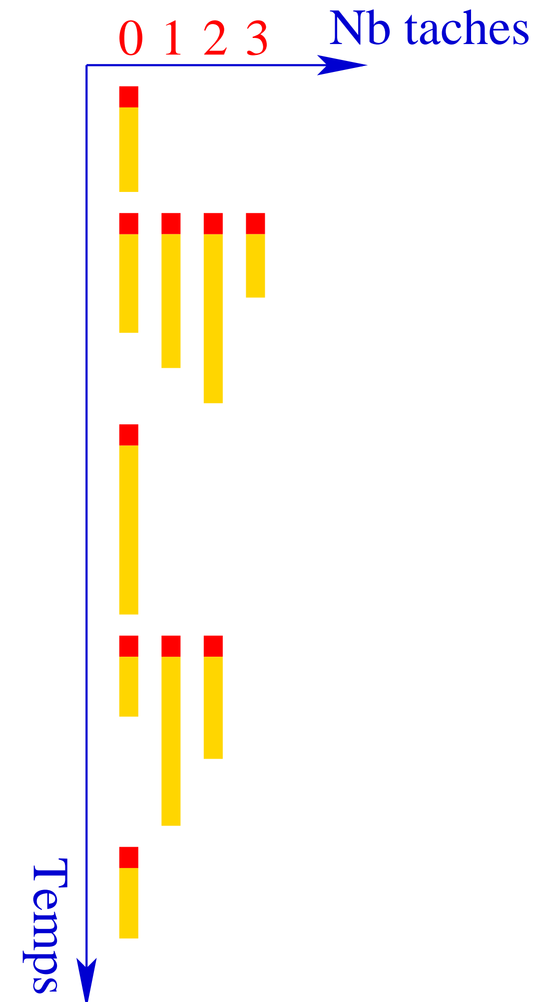
1.3 – Terminologie et définitions

- *Thread* : Entité d'exécution avec une mémoire locale (*stack*)
- *Team* : Un ensemble de un ou plusieurs *threads* qui participent à l'exécution d'une région parallèle.
- *Task/Tâche* : Une instance de code exécutable et ses données associées. Elles sont générées par les constructions `PARALLEL` ou `TASK`.
- *Variable partagée* : Une variable dont le nom permet d'accéder au même bloc de stockage au sein d'une région parallèle entre tâches.
- *Variable privée* : Une variable dont le nom permet d'accéder à différents blocs de stockage suivant les tâches, au sein d'une région parallèle.
- *Host device* : Partie matérielle (généralement noeud SMP) sur laquelle `OpenMP` commence son exécution.
- *Target device* : Partie matérielle (carte accélératrice de type GPU ou Xeon Phi) sur laquelle une partie de code ainsi que les données associées peuvent être transférées puis exécutées.

1.4 – Concepts généraux

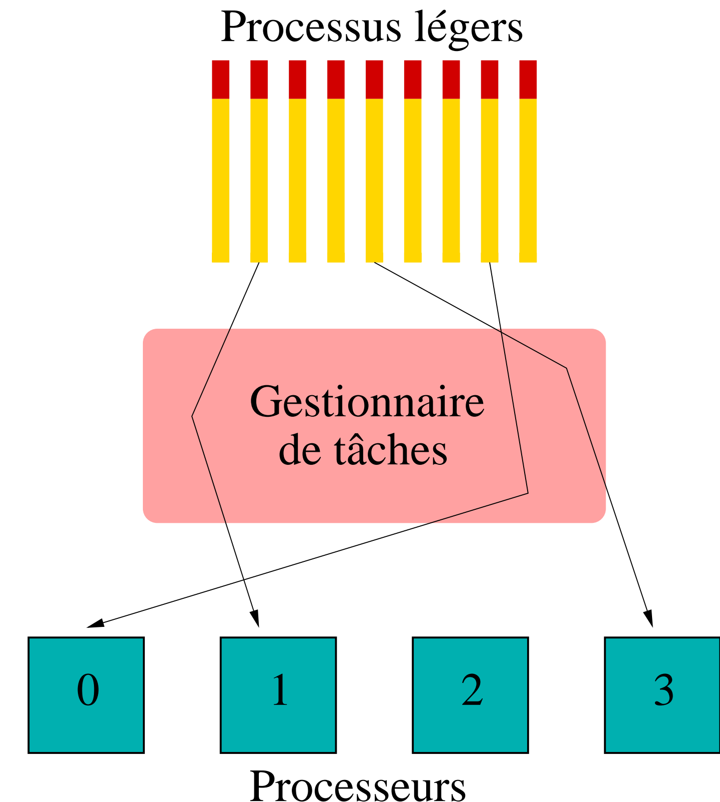
1.4.1 – Modèle d'exécution

- À son lancement, un programme OpenMP est séquentiel. Il est constitué d'un processus unique, le thread maître dont le rang vaut 0, qui execute la tâche implicite initiale.
- OpenMP permet de définir des **régions parallèles**, c'est à dire des portions de code destinées à être exécutées en parallèle.
- Au début d'une région parallèle, de nouveaux processus légers sont créés ainsi que de nouvelles tâches implicites, chaque *thread* exécutant sa tâche implicite, en vue de se partager le travail et de s'exécuter concurremment.
- Un programme OpenMP est une alternance de régions séquentielles et de régions parallèles.

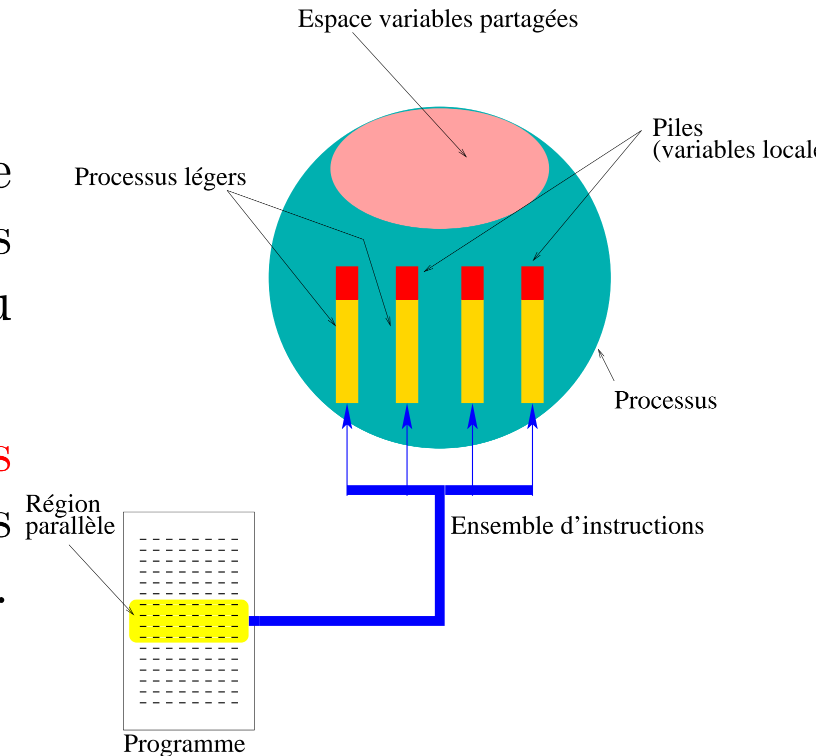


1.4.2 – Processus légers (*threads*)

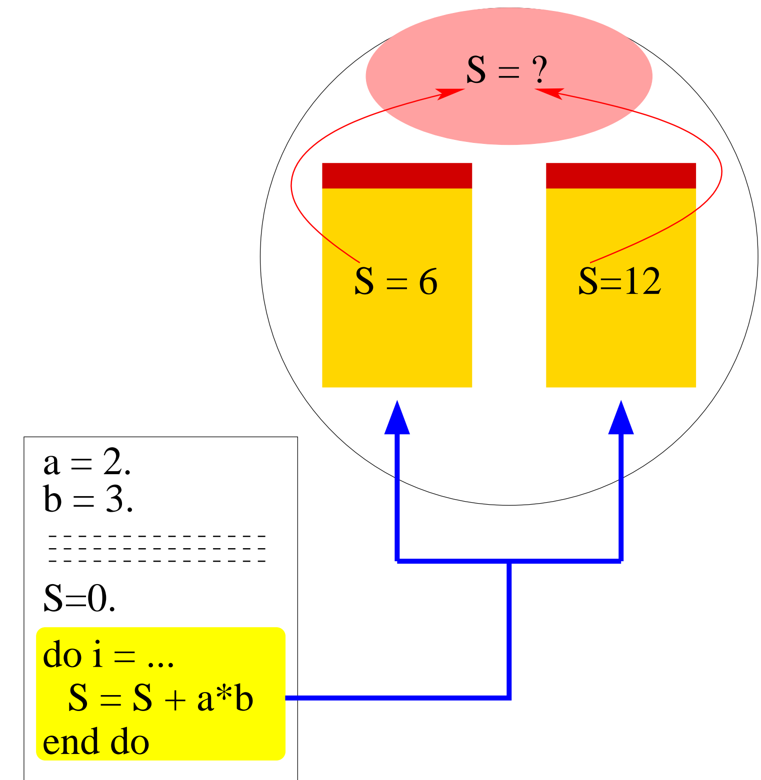
- Chaque processus léger exécute sa propre séquence d'instructions, qui correspond à sa tâche.
- C'est le système d'exploitation qui choisit l'ordre d'exécution des processus (légers ou non) : il les affecte aux unités de calcul disponibles (coeurs des processeurs).
- Il n'y a aucune garantie sur l'ordre global d'exécution des instructions d'un programme parallèle.



- Les tâches d'un même programme partagent l'espace mémoire de la tâche initiale (mémoire partagée) mais disposent aussi d'un espace mémoire local : la pile (ou *stack*).
- Il est ainsi possible de définir des variables partagées (stockées dans la mémoire partagée) ou des variables privées (stockées dans la pile de chacune des tâches).



- En mémoire partagée, il est parfois nécessaire d'introduire une synchronisation entre les tâches concurrentes.
- Une synchronisation permet par exemple d'éviter que deux threads ne modifient dans un ordre quelconque la valeur d'une même variable partagée (cas des opérations de réduction).



1.5 – Fonctionnalités

OpenMP facilite l'écriture d'algorithmes parallèles en mémoire partagée en proposant des mécanismes pour :

- ☞ partager le travail entre les tâches. Il est par exemple possible de répartir les itérations d'une boucle entre les tâches. Lorsque la boucle agit sur des tableaux, cela permet de distribuer simplement le traitement des données entre les processus légers.
- ☞ partager ou privatiser les variables.
- ☞ synchroniser les *threads*.

Depuis la version 3.0, OpenMP permet aussi d'exprimer le parallélisme sous la forme d'un ensemble de tâches explicites à réaliser. OpenMP 4.0 permet de décharger une partie du travail sur un accélérateur.

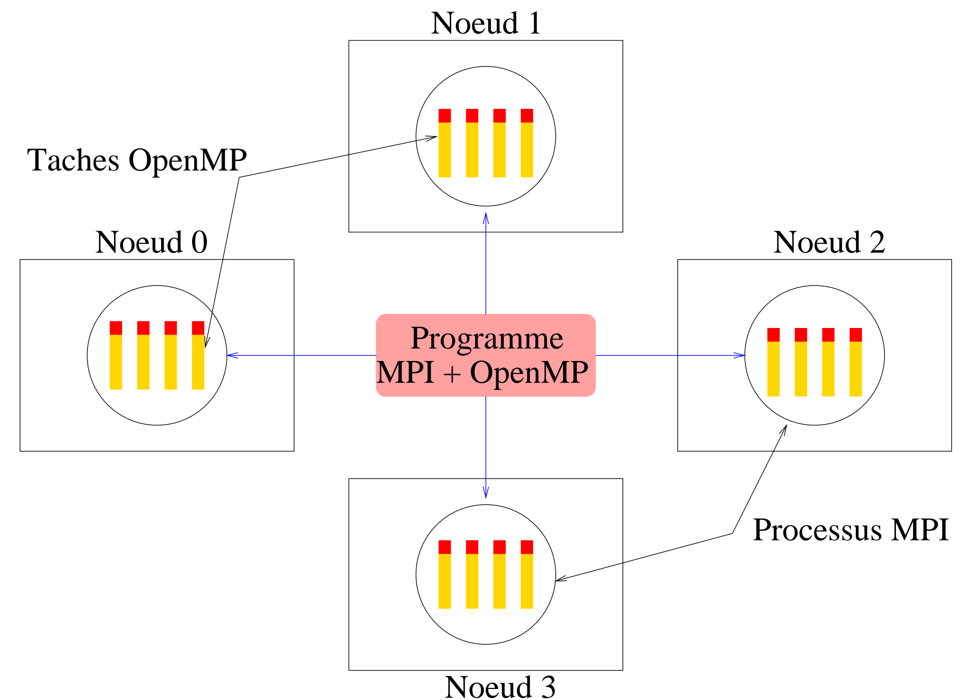
1.6 – OpenMP versus MPI

Ce sont des modèles de programmation adaptées à deux architectures parallèles différentes :

☞ MPI est un modèle de programmation à mémoire distribuée. La communication entre les processus est explicite et sa gestion est à la charge de l'utilisateur.

☞ OpenMP est un modèle de programmation à mémoire partagée. Chaque thread a une vision globale de la mémoire.

☞ 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 ou l'empreinte mémoire du code.



1.7 – Bibliographie

- Premier livre sur OpenMP : R. CHANDRA & *al.*, *Parallel Programming in OpenMP*, éd. Morgan Kaufmann Publishers, oct. 2000.
- Livre plus récent sur OpenMP : B. CHAPMAN & *al.*, *Using OpenMP*, MIT Press, 2008.
- Spécifications du standard OpenMP : https://www.openmp.org/

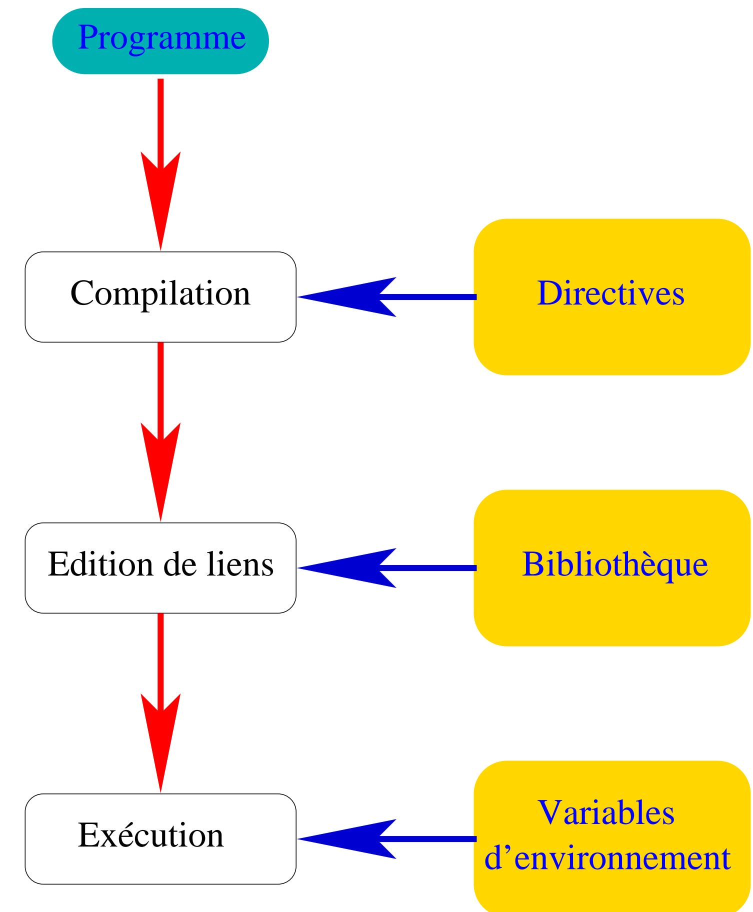
2 – Principes

2.1 – Interface de programmation

❶ Directives et clauses de compilation : elles servent à définir le partage du travail, la synchronisation et le statut privé ou partagé des données ;

❷ Fonctions et sous-programmes : ils font partie d'une bibliothèque chargée à l'édition de liens du programme.

❸ Variables d'environnement : une fois positionnées, leurs valeurs sont prises en compte à l'exécution.



2.1.1 – Syntaxe générale d'une directive

- Une directive OpenMP possède la forme générale suivante :

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

- Les directives OpenMP sont considérées par le compilateur comme des lignes de commentaires à moins de spécifier une option adéquate de compilation pour qu'elles soient interprétées.
- 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++ :

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

2.1.2 – Compilation

Voici les options de compilation permettant d'activer l'interprétation des directives OpenMP par certains compilateurs :

- Compilateur GNU : -fopenmp

```
gfortran -fopenmp prog.f90                        # Compilateur Fortran
```

- Compilateur Intel : -fopenmp ou -qopenmp

```
ifort -fopenmp prog.f90                           # Compilateur Fortran
```

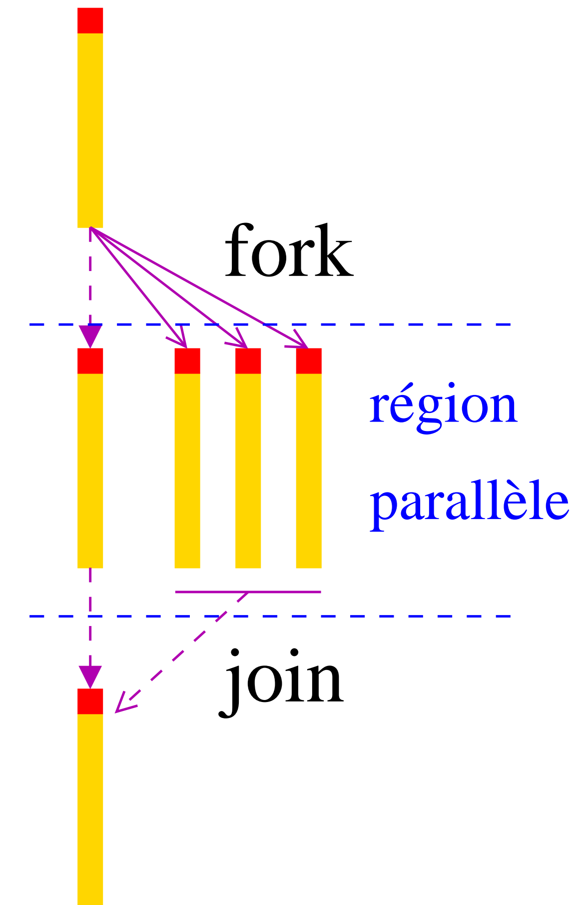
- Compilateur PGI/NVIDIA : -mp

```
pgfortran/nvfortran -mp prog.f90                  # Compilateur Fortran
```

Exemple d'exécution :

```
export OMP_NUM_THREADS=4                          # Nombre de threads souhaité
./a.out                                           # Exécution
```

2.2 – Construction d'une région parallèle



- Un programme `OpenMP` est une alternance de régions séquentielles et parallèles (modèle « *fork and join* »)
- À l'entrée d'une région parallèle, le *thread* maître (celui de rang 0) crée/active (*fork*) des processus « fils » (processus légers) et autant de tâches implicites. Ces processus fils exécutent leur tâche implicite puis disparaissent ou s'assoupissent en fin de région parallèle (*join*).
- En fin de région parallèle, l'exécution redevient séquentielle avec uniquement l'exécution du *thread* maître.

```
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
 !$OMP END PARALLEL
 print*,"Parallel ?:", p

end program parallel
```

```
> ifort -fopenmp prog.f90
> export OMP_NUM_THREADS=4
> a.out
```

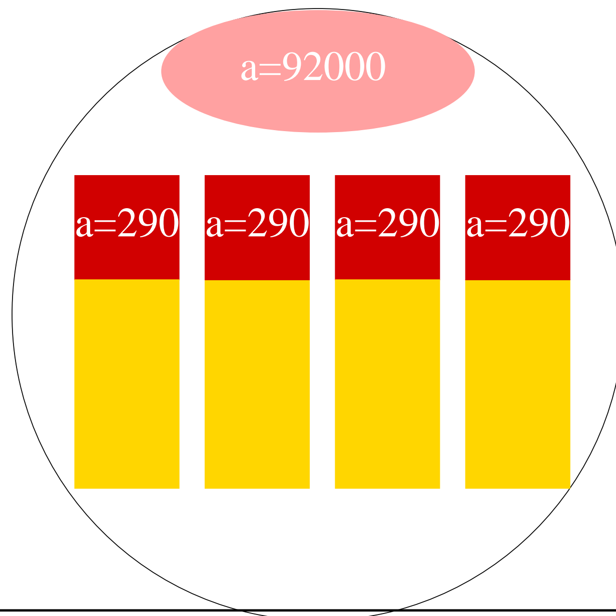
```
 A vaut : 92290
 A vaut : 92290
 A vaut : 92290
 A vaut : 92290
 Parallele ? : T
```

- Au sein d'une même région parallèle, tous les *threads* exécutent chacun une tâche implicite différente, mais composée du même code.
- Par défaut, les variables sont partagées.
- Il existe une barrière implicite de synchronisation en fin de région parallèle.

2.3 – Statut des variables

2.3.1 – Variables privées

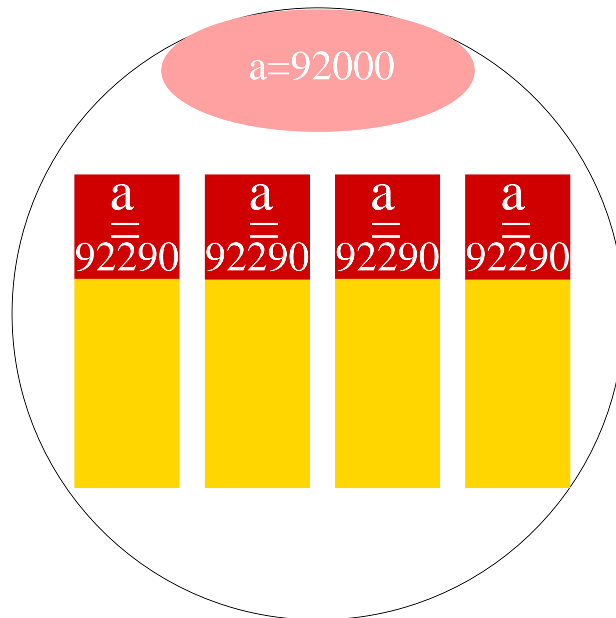
- La clause PRIVATE permet de changer le statut d'une variable.
- Si une variable possède un statut privé, elle est allouée dans la pile de chaque tâche.
- Les variables privées ne sont pas initialisées à l'entrée d'une région parallèle.




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

  a = 92000
!$OMP PARALLEL PRIVATE(rang,a)
    !$ rang = OMP_GET_THREAD_NUM()
    a = a + 290
    print *,"Rang : ",rang, &
            "; A vaut : ",a
!$OMP END PARALLEL
  print*,"Hors region, A vaut :",a

end program parallel
```

```
Rang : 1 ; A vaut : 290
Rang : 2 ; A vaut : 290
Rang : 0 ; A vaut : 290
Rang : 3 ; A vaut : 290
Hors region, A vaut : 92000
```

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



☞ En sortie de région parallèle, les variables privées sont perdues.

```
program parallel
 implicit none
 real :: a

 a = 92000.
!$OMP PARALLEL FIRSTPRIVATE(a)
   a = a + 290
   print *,"A vaut : ",a
!$OMP END PARALLEL
print*,"Hors region, A vaut :",a

end program parallel
```

```
> ifort -fopenmp 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
```

2.3.2 – La clause DEFAULT

- Par défaut, les variables sont partagées mais pour éviter les erreurs, il est recommandé de définir le statut de chaque variable explicitement.
- La clause DEFAULT(NONE) permet d'obliger le programmeur à expliciter le statut de chaque variable.
- En Fortran, il est aussi possible de changer le statut implicite des variables en utilisant par exemple la clause DEFAULT(PRIVATE).

```
program parallel
 !$ use OMP_LIB
 implicit none
 logical :: p

 p=.false.
 !$OMP PARALLEL DEFAULT(NONE) &
       !$OMP SHARED(p)
  !$ p = OMP_IN_PARALLEL()
 !$OMP END PARALLEL
 print*,"Parallel ?:", p
end program parallel
```

```
Parallele ? : T
```

2.3.3 – 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 seul un *thread* (p. ex. le *thread* maître) se charge de cette opération. Pour des raisons de localité des données, il est recommandé d'initialiser les variables à l'intérieur de la région parallèle (« first touch »).

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

  n=1024
  allocate(a(n))
  ! $OMP PARALLEL DEFAULT(NONE) PRIVATE(debut,fin,nb_taches,rang,i) &
        !$OMP SHARED(a,n) IF(n .gt. 512)
     nb_taches=OMP_GET_NUM_THREADS() ; rang=OMP_GET_THREAD_NUM()
     debut=1+(rang*n)/nb_taches
     fin=((rang+1)*n)/nb_taches
     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
```

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

```
Rang : 3 ; A( 769 ), ... , A( 1024 ) : 93059., ... , 93314.
Rang : 2 ; A( 513 ), ... , A(  768 ) : 92803., ... , 93058.
Rang : 1 ; A( 257 ), ... , A(  512 ) : 92547., ... , 92802.
Rang : 0 ; A(   1 ), ... , A(  256 ) : 92291., ... , 92546.
```

2.3.4 – Équivalence entre variables Fortran

- 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
```

```
> ifort -fopenmp 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
```

2.4 – 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
```

```
> ifort -fopenmp 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 tableaux automatiques sont implicitement privés à chaque tâche (ils sont définis dans la pile).
- ☞ En C/C++, les variables déclarées à l'intérieur d'une région parallèle sont privées.

```
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
```

```
> ifort -fopenmp prog.f90
> export OMP_NUM_THREADS=4;a.out
```

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

2.5 – Transmission par arguments

☞ 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
```

```
> ifort -fopenmp prog.f90
> export OMP_NUM_THREADS=4
> a.out
```

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

2.6 – Variables statiques

- Une variable statique est une variable conservée pendant toute la durée de vie d'un programme.
  - En Fortran, c'est le cas des variables apparaissant en COMMON ou dans un MODULE ou déclarées SAVE ou initialisées à la déclaration (instruction DATA ou symbole =).
  - En C/C++, ce sont les variables déclarées avec le mot clé static.
- Dans une région parallèle OpenMP, une variable statique est par défaut une **variable partagée**.

```
module var_stat
  real :: c
end module var_stat
```

```
program parallel
  use var_stat
  implicit none
  real :: a
  common /bidon/a
  !$OMP PARALLEL
    call sub()
  !$OMP END PARALLEL
end program parallel
subroutine sub()
  use var_stat
  use OMP_LIB
  implicit none
  real    :: a, b=10.
  integer :: rang
  common /bidon/a
  rang = OMP_GET_THREAD_NUM()
  a=rang;b=rang;c=rang
  !$OMP BARRIER
  print *,"valeurs de A, B et C : ",a,b,c
end subroutine sub
```

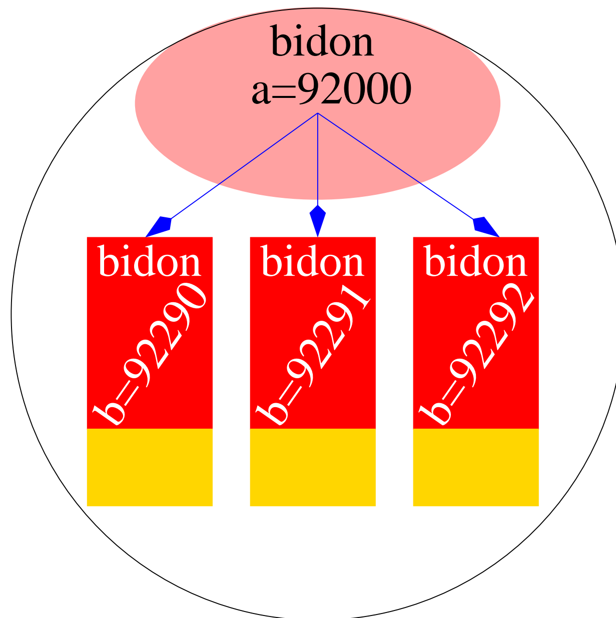
```
> ifort -fopenmp var_stat.f90 prog.f90
> export OMP_NUM_THREADS=2;a.out
```

Un résultat possible est :

```
valeurs de A, B et C : 0.0 1.0 1.0
valeurs de A, B et C : 0.0 1.0 1.0
```

☞ L'utilisation de la directive THREADPRIVATE permet de privatiser une instance statique (pour les *threads* et non les tâches...) 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 tous les *threads*.




```
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
```

2.7 – Compléments

- La construction d'une région parallèle admet deux autres clauses :
  - REDUCTION : pour les opérations de réduction avec synchronisation implicite entre les *threads* ;
  - NUM_THREADS : Elle permet de spécifier le nombre de *threads* 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 *threads* concurrents peut être variable si on le souhaite.

```
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
```

```
> ifort -fopenmp prog.f90
> export OMP_NUM_THREADS=4
> ./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`.

```
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
```

```
> ifort ... -fopenmp prog.f90
> export OMP_NESTED=true; ./a.out
```

```
 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
```

3 – Partage du travail

3.1 – Introduction

- En principe, la création d'une région parallèle et l'utilisation de quelques fonctions `OpenMP` suffisent à elles seules 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 au sein d'une région parallèle.
- Heureusement, des directives permettent de faciliter cette répartition (`DO`, `WORKSHARE`, `SECTIONS`)
- Par ailleurs, il est possible de faire exécuter des portions de code situées dans une région parallèle à un seul thread (`SINGLE`, `MASTER`).
- La synchronisation entre les threads sera abordée dans le chapitre suivant.

3.2 – Boucle parallèle

- Une boucle est parallèle si toutes ses itérations sont indépendantes les unes des autres.
- 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 cette directive, elles le sont via les tâches explicites.
- 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 threads.
- Les indices de boucles sont par défaut des variables entières privées, dont il n'est pas indispensable de spécifier le statut.
- 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.

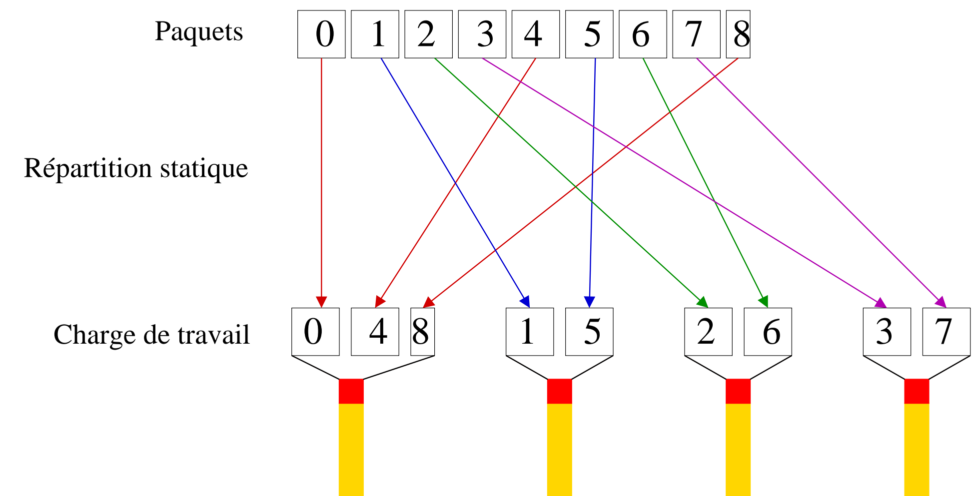
3.2.1 – Clause SCHEDULE

```
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
```

```
> ifort ... -fopenmp 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 chacun des threads, un ensemble de paquets suivant l'ordre des threads 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, ce qui peut parfois engendrer une dégradation des performances.
- 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.

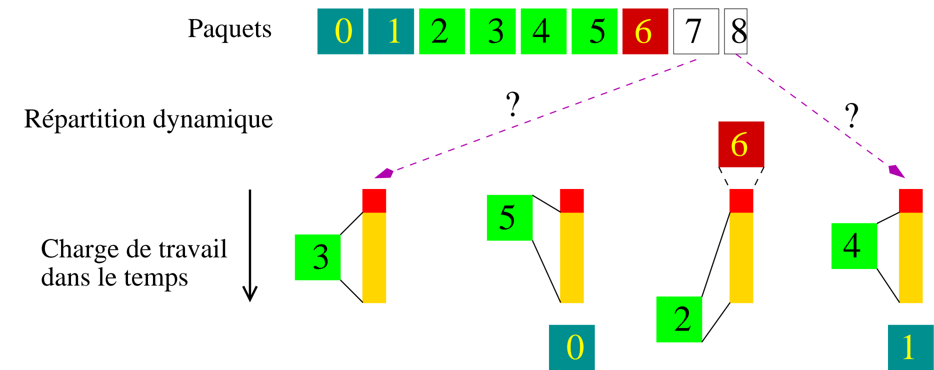
```
program parallel
 !$ use OMP_LIB
 implicit none
 integer, parameter :: n=4096
 real, dimension(n) :: a
 integer :: i, i_min, i_max
 !$OMP PARALLEL DEFAULT(PRIVATE) SHARED(a)
   i_min=n ; i_max=0
   !$OMP DO SCHEDULE(RUNTIME)
      do i = 1, n
        a(i) = 92290. + real(i)
        i_min=min(i_min,i)
        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,1024"
> a.out
```

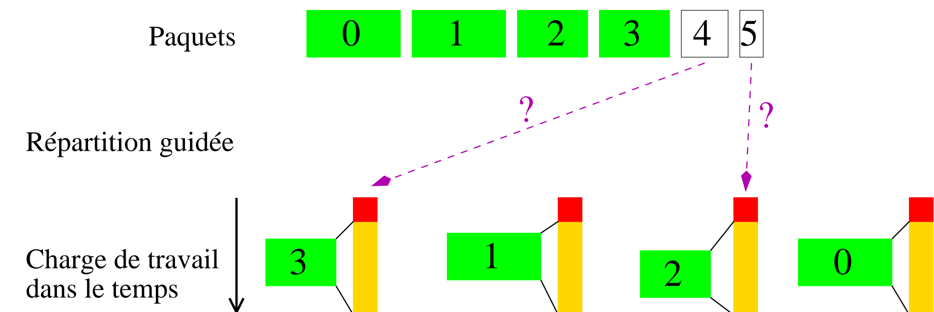
```
Rang: 0 ; i_min:    1 ; i_max: 3072
Rang: 1 ; i_min: 1025 ; i_max: 4096
```

- En plus du mode STATIC, il existe trois autres façons de répartir les itérations d'une boucle :
  - DYNAMIC : les itérations sont divisées en paquets de taille donnée. Sitôt qu'un thread épuise les itérations de son paquet, un autre paquet lui est attribué ;
  - GUIDED : les itérations sont divisées en paquets dont la taille décroît exponentiellement. Tous les paquets ont une taille supérieure ou égale à une valeur donnée à l'exception du dernier dont la taille peut être inférieure. Sitôt qu'un thread finit les itérations de son paquet, un autre paquet d'itérations lui est attribué.
  - AUTO : le choix de la répartition des itérations de la boucle est délégué au compilateur ou au système à l'exécution (i.e. « *runtime* »).

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




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



3.2.2 – 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.

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

```
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
```

```
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
```

3.2.3 – 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 thread calcule un résultat partiel indépendamment des autres. Ils 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
```

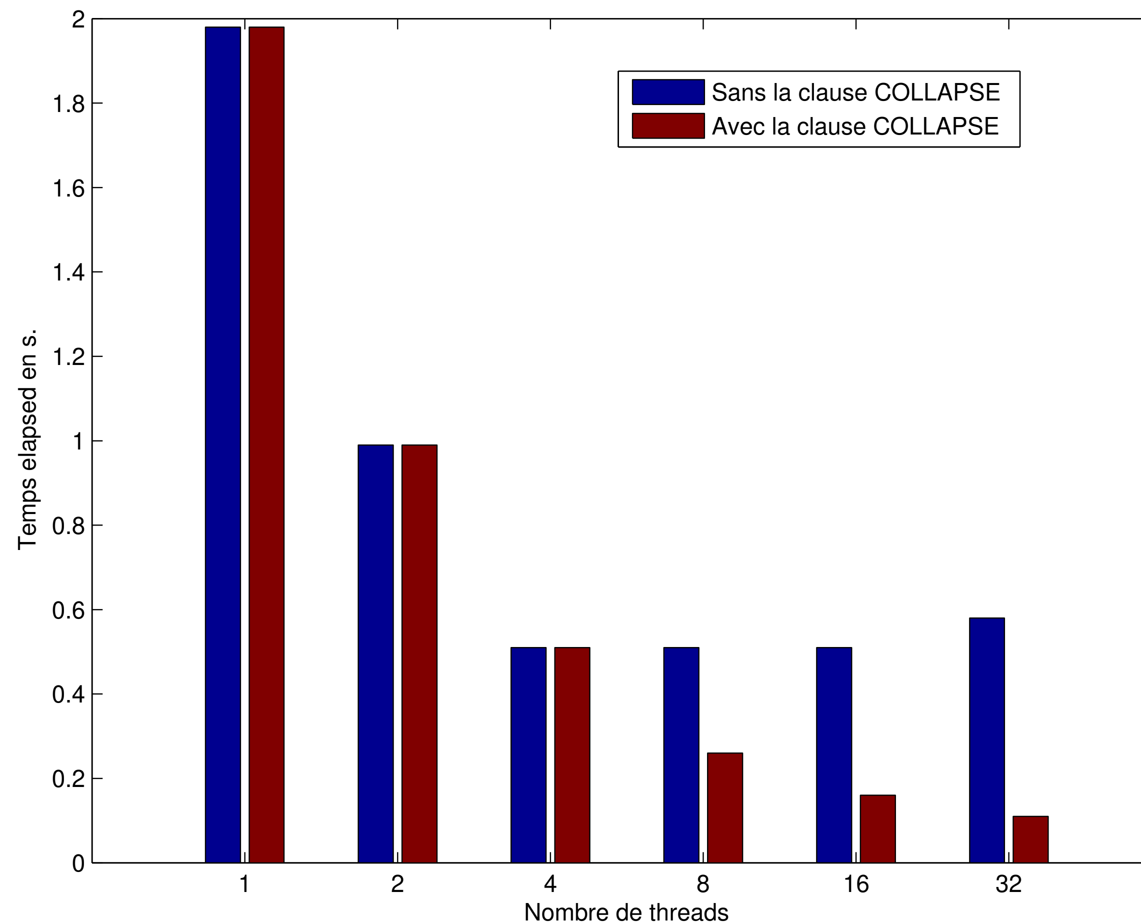
3.2.4 – Cas de la fusion d'un nid de boucles

- Dans le cas de boucles parfaitement imbriquées et sans dépendances, il peut être intéressant de les fusionner pour obtenir un espace d'itération plus grand.
- Ainsi, on augmente la granularité de travail de chacun des threads ce qui peut parfois améliorer significativement les performances.
- La clause `COLLAPSE(n)` permet de fusionner les `n` boucles imbriquées qui suivent immédiatement la directive. Le nouvel espace d'itération est alors partagé entre les threads suivant le mode de répartition choisi.

```
program boucle_collapse
implicit none
integer, parameter :: n1=4, n2=8, &
                      n3=1000000
real, dimension(:,:,:) :: A(n1,n2,n3)
integer :: i, j, k

...
!$OMP PARALLEL
!$OMP DO SCHEDULE(STATIC) COLLAPSE(2)
do i=1,n1
   do j=1,n2
      do k=2,n3
         A(i,j,k)=exp(sin(A(i,j,k-1))+ &
                  cos(A(i,j,k)))/2
      enddo
   enddo
enddo
!$OMP END DO
!$OMP END PARALLEL
end program boucle_collapse
```

☞ Exécution du programme précédent avec et sans la clause `COLLAPSE`.

☞ Évolution du temps elapsed d'exécution (en s.) en fonction du nombre de threads qui varie de 1 à 32.



3.2.5 – 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 le thread exécutant la dernière itération de la 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
```

3.3 – Construction WORKSHARE

- Elle ne peut être spécifiée qu'au sein d'une région parallèle.
- Elle est utile pour répartir le travail essentiellement lié à certaines constructions Fortran 95 telles que les :
  - affectations de type tableau Fortran 90 (i.e. notation A(:, :)) ;
  - fonctions intrinsèques portant sur des variables de type tableaux (MATMUL, DOT_PRODUCT, SUM, PRODUCT, MAXVAL, MINVAL, COUNT, ANY, ALL, SPREAD, PACK, UNPACK, RESHAPE, TRANSPOSE, EOSHIFT, CSHIFT, MINLOC et MAXLOC) ;
  - instructions ou blocs FORALL et WHERE ;
  - fonctions dites « ELEMENTAL » définies par l'utilisateur.
- Elle n'admet que la clause NOWAIT en fin de construction (END WORKSHARE).

- Seules les instructions ou blocs **Fortran 95** spécifiés dans l'étendue lexicale verront leur travail réparti entre les threads.
- 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 parfois 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
      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 – Sections parallèles

- Une section est une portion de code exécutée par un et un seul thread.
- 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 différents threads.
- La clause `NOWAIT` est admise en fin de construction `END SECTIONS` pour lever la barrière de synchronisation implicite.

3.4.1 – Construction SECTIONS

```fortran
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
```

```fortran
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
```

3.4.2 – 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.

3.5 – Exécution exclusive

- Il arrive que l'on souhaite exclure tous les threads à l'exception d'un seul pour exécuter certaines portions de code incluses dans une région parallèle.
- Pour ce 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.

3.5.1 – Construction SINGLE

- La construction SINGLE permet de faire exécuter une portion de code par un et un seul thread sans pouvoir indiquer lequel.
- En général, c'est le thread qui arrive le premier sur la construction SINGLE mais cela n'est pas spécifié dans la norme.
- Tous les threads n'exécutant pas la région SINGLE attendent, en fin de construction END SINGLE, la terminaison de celui qui en a la charge, à moins d'avoir spécifié la clause NOWAIT.

```fortran
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
```

```
> ifort ... -fopenmp 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.
```

- ☞ La clause supplémentaire COPYPRIVATE est admise par la directive de terminaison END SINGLE et elle seule.
- ☞ Elle permet au thread chargé d'exécuter la région SINGLE de diffuser aux autres threads la valeur d'une liste de variables privées avant de sortir de cette région.
- ☞ Les autres clauses admises par la directive SINGLE sont PRIVATE, FIRSTPRIVATE et NOWAIT.

```fortran
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
```

```
> ifort ... -fopenmp 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.
```

3.5.2 – Construction MASTER

- La construction MASTER permet de faire exécuter une portion de code par le seul thread maître.
- 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
```

```
> ifort ... -fopenmp 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.
```

3.6 – Procédures orphelines

- Une procédure (fonction ou sous-programme) appelée dans une région parallèle est exécutée par tous les threads.
- 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 multithreadée, parallélisée 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 OpenMP	sans OpenMP
avec OpenMP	P + D	P + R
sans OpenMP	S	S

Table 1 – Contexte d'exécution selon le mode de compilation

3.7 – Récapitulatif

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

4 – Synchronisations

La synchronisation devient nécessaire dans les situations suivantes :

❶ pour s'assurer que tous les threads concurrents aient atteint un même niveau d'instruction dans le programme (barrière globale) ;

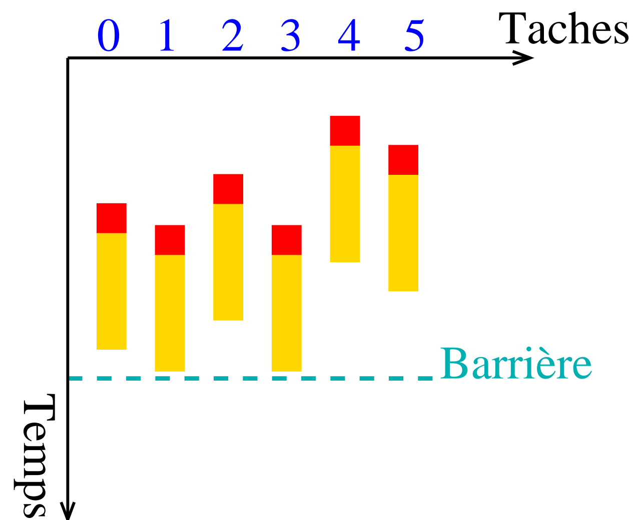
❷ pour ordonner l'exécution de tous les threads concurrents quand ceux-ci doivent exécuter une même portion de code affectant une ou plusieurs variables partagées dont la cohérence en mémoire (en lecture ou en écriture) doit être garantie (exclusion mutuelle).

❸ pour synchroniser au moins deux threads concurrents parmi tous les autres (mécanisme de verrou).

- 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 de mécanismes de verrou (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 threads concurrents dans une région parallèle.
- Chacun des threads attend que tous les autres soient arrivés 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
```

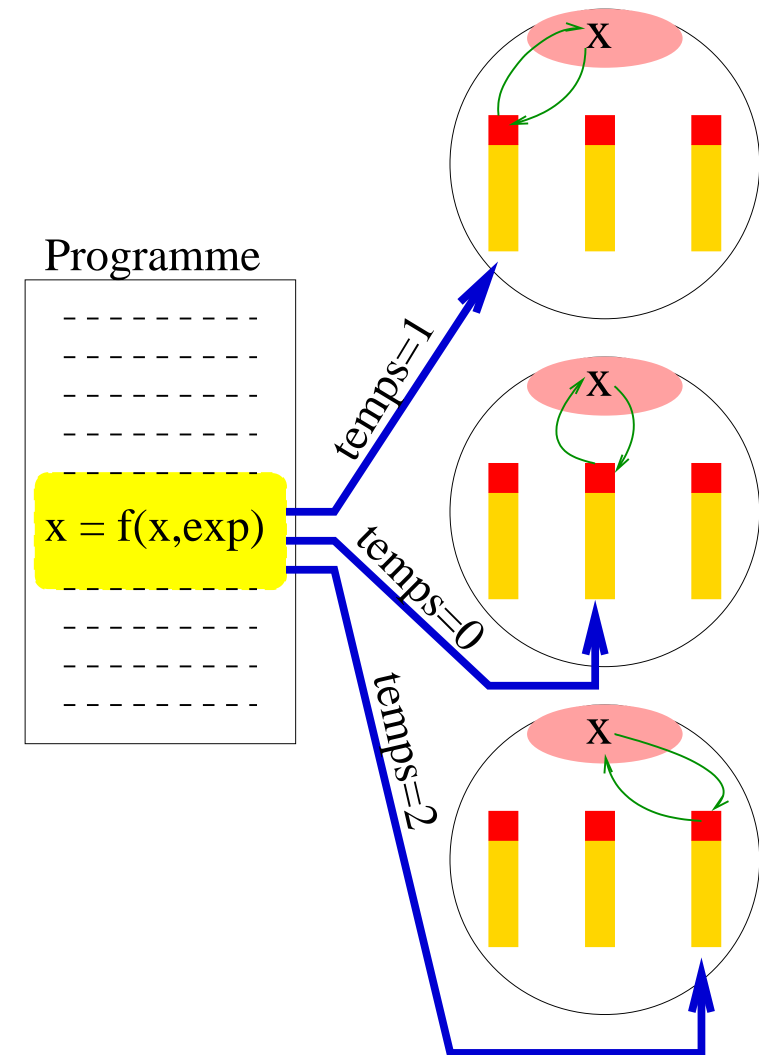
4.2 – Mise à jour atomique

- La directive ATOMIC assure qu'une variable partagée est lue et modifiée en mémoire par un seul thread 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 : 1 ; compteur vaut : 92291
Rang : 0 ; 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 : `+`, `-`, ×, `/`, `.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.
- ☞ Tous les threads exécutent cette région dans un ordre non-déterministe, mais un seul 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 performance, il est fortement déconseillé d'émuler une instruction atomique par une région critique.
- ☞ Un nom optionnel peut être utilisé pour nommer une région critique.
- ☞ Toutes les régions critiques non explicitement nommées sont considérées comme ayant le même nom non spécifié.
- ☞ Si plusieurs régions critiques ont le même nom, elles sont considérées pour le mécanisme d'exclusion mutuel comme une seule et unique région critique.

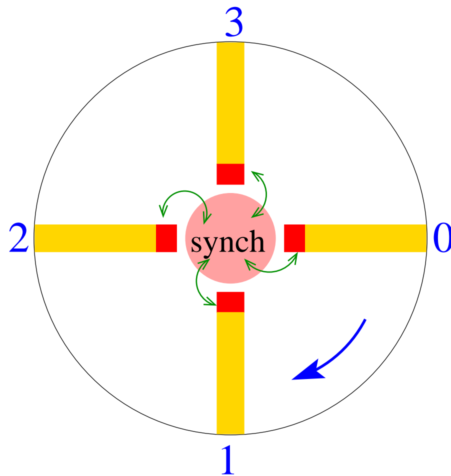
```fortran
program parallel
  implicit none
  integer :: s, p

  s=0
  p=1

  !$OMP PARALLEL
    !$OMP CRITICAL
      s = s + 1
    !$OMP END CRITICAL
    !$OMP CRITICAL (RC1)
      p = p * 2
    !$OMP END CRITICAL (RC1)
    !$OMP CRITICAL
      s = s + 1
    !$OMP END CRITICAL
  !$OMP END PARALLEL

  print *, "s= ",s, " ; p= ",p

end program parallel
```

```
> ifort ... -fopenmp prog.f90
> export OMP_NUM_THREADS=4 ; a.out
```

```
s= 8 ; 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 threads.




```
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
```

4.4.1 – Exemple avec un piège facile

```
program anneau2-faux
 !$ use OMP_LIB
 implicit none
 integer :: rang,nb_taches,synch=0,compteur=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
       !$OMP FLUSH(synch)
       if(synch == rang-1) exit
     end do
   end if
   compteur=compteur+1
   print *,"Rang:",rang,";synch:",synch
   synch=rang
   !$OMP FLUSH(synch)
 !$OMP END PARALLEL
 print *,"Compteur = ",compteur
end program anneau2-faux
```

4.4.2 – Exemple avec un piège difficile

```
program anneau3-faux
 !$ use OMP_LIB
 implicit none
 integer :: rang,nb_taches,synch=0,compteur=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
       !$OMP FLUSH(synch)
       if(synch == rang-1) exit
     end do
   end if
   print *,"Rang:",rang,";synch:",synch
   !$OMP FLUSH(compteur)
   compteur=compteur+1
   !$OMP FLUSH(compteur)
   synch=rang
   !$OMP FLUSH(synch)
 !$OMP END PARALLEL
 print *,"Compteur = ",compteur
end program anneau3-faux
```

4.4.3 – Commentaires sur les codes précédents

- Dans anneau2-faux, on n'a pas *flushé* la variable partagée *compteur* avant et après l'avoir incrémentée. Le résultat final peut potentiellement être faux.
- Dans anneau3-faux, le compilateur peut inverser les lignes :

```
compteur=compteur+1
!$OMP FLUSH(compteur)
```

et les lignes :

```
synch=rang
!$OMP FLUSH(synch)
```

libérant le *thread* qui suit avant que la variable *compteur* n'ait été incrémentée... Là encore, le résultat final pourrait potentiellement être faux.

- Pour résoudre ce problème, il faut *flusher* les deux variables *compteur* et *synch* juste après l'incrémentation de la variable *compteur*, ainsi on impose un ordre au compilateur.
- Le code corrigé se trouve à la diapositive suivante.

4.4.4 – Code corrigé

```
program anneau4
 !$ use OMP_LIB
 implicit none
 integer :: rang,nb_taches,synch=0,compteur=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
       !$OMP FLUSH(synch)
       if(synch == rang-1) exit
     end do
   end if
   print *,"Rang:",rang,";synch:",synch
   !$OMP FLUSH(compteur)
   compteur=compteur+1
   !$OMP FLUSH(compteur,synch)
   synch=rang
   !$OMP FLUSH(synch)
 !$OMP END PARALLEL
 print *,"Compteur = ",compteur
end program anneau4
```

4.4.5 – Nid de boucles avec double dépendance

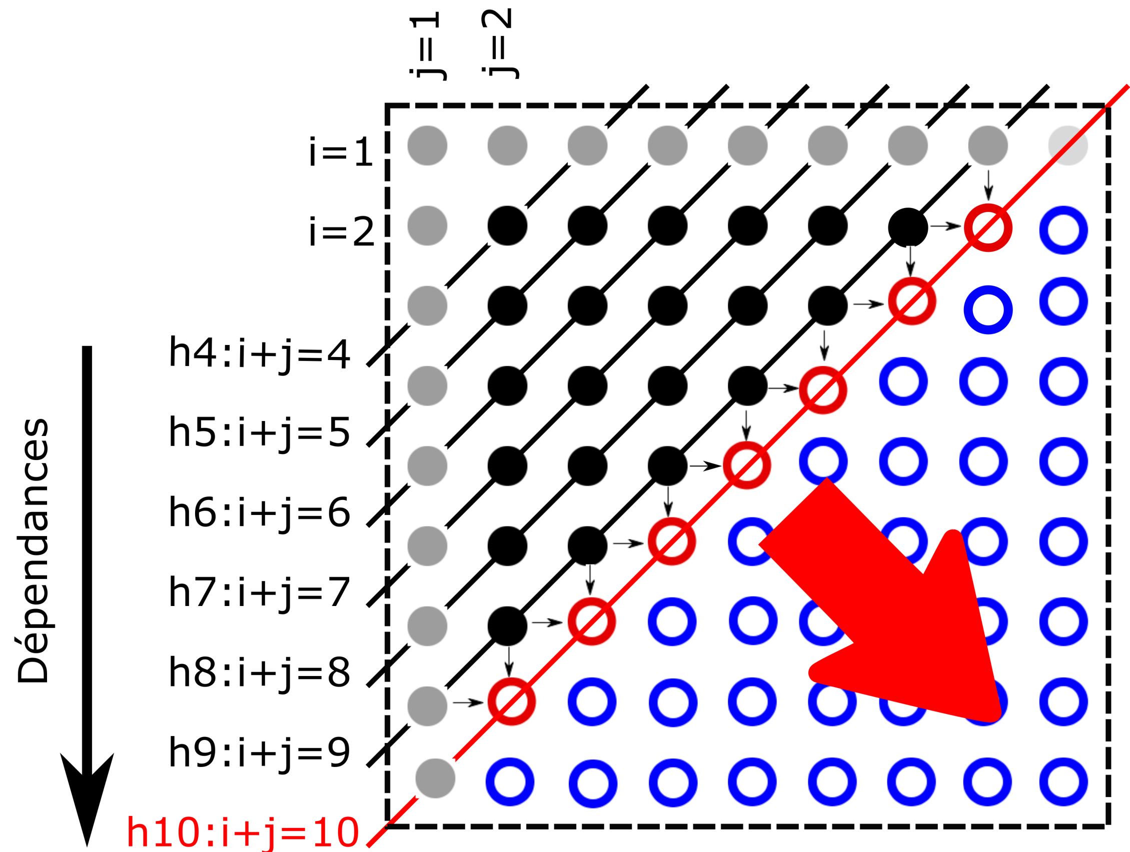
- Considérons le code suivant :

```
! Boucles avec double dependance
do j = 2, ny
   do i = 2, nx
      V(i,j) =(V(i,j) + V(i-1,j) + V(i,j-1))/3
   end do
end do
```

- C'est un problème classique en parallélisme (par exemple décomposition LU).
- Du fait de la dépendance arrière en i et en j, ni la boucle en i, ni la boucle en j ne sont parallèles (i.e. chaque itération en i ou j dépend de l'itération précédente).
- Paralléliser avec la directive OpenMP `PARALLEL DO` la boucle en i ou la boucle en j donnerait des résultats faux.
- Pourtant, il est quand même possible d'exhiber du parallélisme de ce nid de boucles en effectuant les calculs dans un ordre qui ne casse pas les dépendances.
- Il existe au moins deux méthodes pour paralléliser ce nid de boucles : l'algorithme de *l'hyperplan* et celui du *software pipelining*.

Algorithme de l'hyperplan

- Le principe est simple : nous allons travailler sur des hyperplans d'équation : $i + j = cste$ qui correspondent à des diagonales de la matrice.
- Sur un hyperplan donné, les mises à jour des éléments de cet hyperplan sont indépendantes les unes des autres, donc ces opérations peuvent être réalisées en parallèle.
- Par contre, il existe une relation de dépendance entre les hyperplans ; on ne peut pas mettre à jour d'éléments de l'hyperplan H_n tant que la mise à jour de ceux de l'hyperplan H_{n-1} n'est pas terminée.



Algorithme de l'hyperplan (2)

- Une réécriture du code est nécessaire, avec une boucle externe sur les hyperplans (non parallèle) et une boucle parallèle interne sur les éléments appartenant à l'hyperplan qui peuvent être mis à jour dans un ordre quelconque.
- Le code peut se réécrire sous la forme suivante :

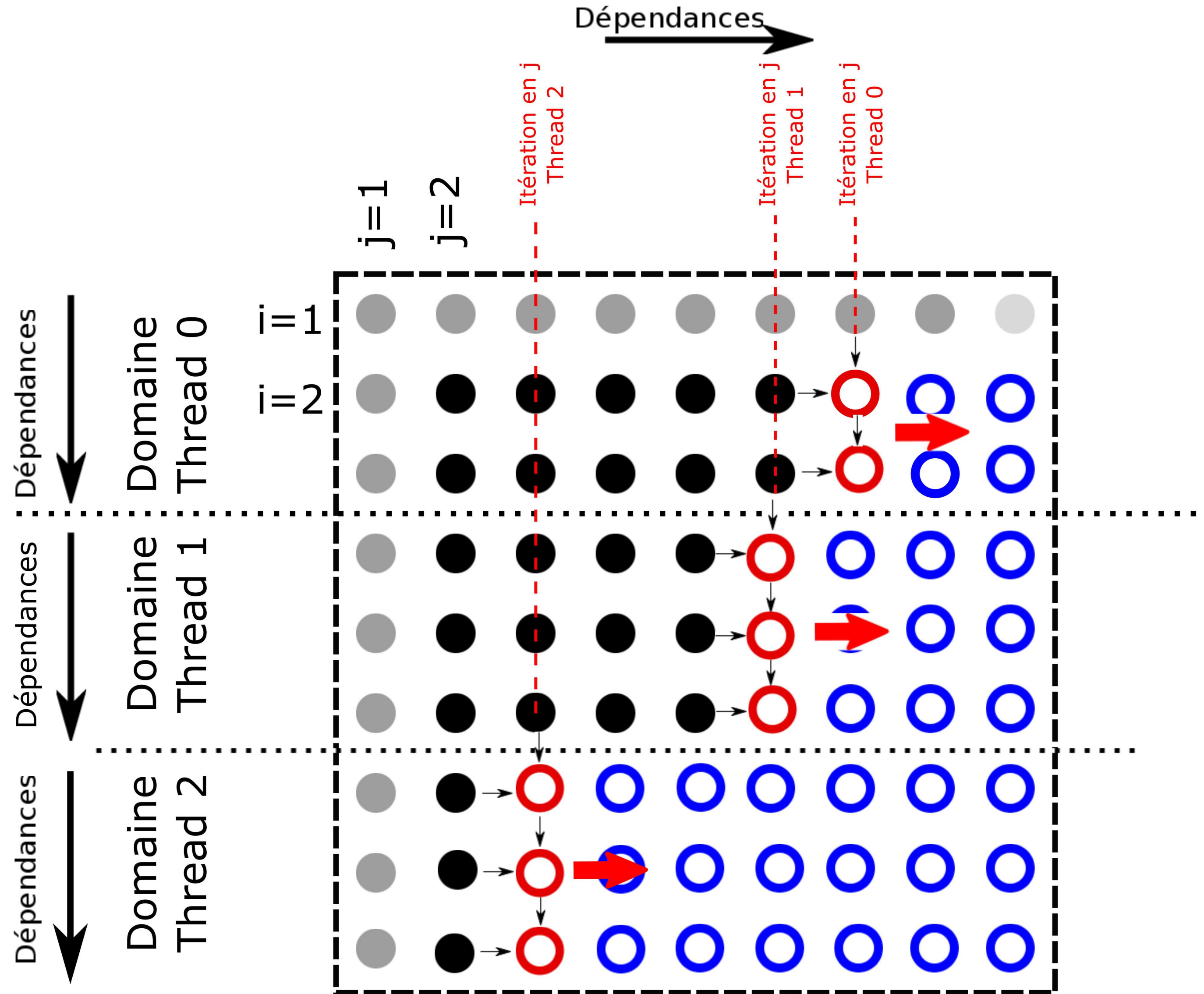
```
! boucle non //, dépendance entre les hyperplans
do h = 1,nbre_hyperplan
   ! calcul tab. d'indices i et j des éléments des hyperplans h
   call calcul(INDI,INDJ,h)
   ! boucle sur le nombre d'éléments de l'hyperplan h
   do e = 1,nbre_element_hyperplan
      i = INDI(e)
      j = INDJ(e)
      V(i,j) =(V(i,j) + V(i-1,j) + V(i,j-1))/3  ! MAJ de l'élément V(i,j)
   enddo
enddo
```

- Une fois le code réécrit, la parallélisation est très simple et ne nécessite pas d'avoir recours à des synchronisations fines.
- Les performances obtenues ne sont hélas pas optimales (médiocre utilisation des caches due aux accès en diagonale, donc non contigu en mémoire).

Algorithme *software pipelining*

- Le principe est simple : paralléliser par blocs la boucle la plus interne et jouer sur les itérations de la boucle externe pour ne pas casser de dépendance en synchronisant finement les *threads* entre eux.
- On découpe la matrice en tranches horizontales et on attribue chaque tranche à un *thread*.
- Les dépendances imposent alors que le *thread* 0 doit toujours traiter une itération de la boucle externe j qui doit être supérieure à celle du *thread* 1, qui elle-même doit être supérieure à celle du *thread* 2 et ainsi de suite...
- Lorsqu'un *thread* a terminé de traiter la j^{eme} colonne de son domaine, il doit vérifier avant de continuer que le *thread* qui le précède a lui-même terminé de traiter la colonne suivante (i.e. la $j+1^{eme}$). Si ce n'est pas le cas, il faut le faire attendre jusqu'à ce que cette condition soit remplie.
- Pour implémenter cet algorithme, il faut constamment synchroniser les *threads* deux à deux et ne libérer un *thread* que lorsque la condition énoncée précédemment est réalisée.

Algorithme *software pipelining* (2)



Algorithme *software pipelining* (3)

☞ Finalement, l'implémentation de cette méthode peut se faire de la façon suivante :

```
myOMPRank = ...
nbOMPThrds = ...

call calcul_borne(iDeb,iFin)

do j= 2,n
   ! On bloque le thread (sauf le 0) tant que le
   ! précedent n'a pas fini le traitement
   ! de l'itération j+1
   call sync(myOMPRank,j)

   ! Boucle // distribuée sur les threads
   do i = iDeb,iFin
      ! MAJ de l'élément V(i,j)
      V(i,j) =(V(i,j) + V(i-1,j) + V(i,j-1))/3
   enddo
enddo
```

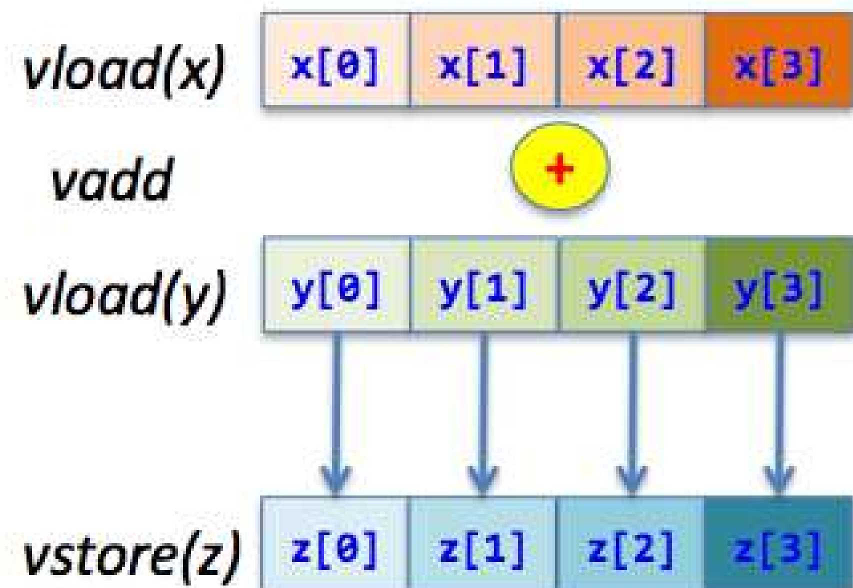
4.5 – Récapitulatif

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

5 – Vectorisation SIMD

5.1 – Introduction

- SIMD = Single Instruction Multiple Data
- Une seule instruction/opération agit en parallèle sur plusieurs éléments.
- Avant OpenMP 4.0, les développeurs devaient soit se reposer sur le savoir-faire du compilateur, soit avoir recours à des extensions propriétaires (directives ou fonctions intrinsèques).
- OpenMP 4.0 offre la possibilité de gérer la vectorisation SIMD de façon portable et performante en utilisant les instructions vectorielles disponibles sur l'architecture cible.



5.2 – Vectorisation SIMD d'une boucle

- La directive SIMD permet de découper la boucle qui la suit immédiatement en morceaux dont la taille correspond à celle des registres vectoriels disponibles sur l'architecture cible.
- La directive SIMD n'entraine pas la parallélisation de la boucle.
- La directive SIMD peut ainsi s'utiliser aussi bien à l'intérieur qu'à l'extérieur d'une région parallèle.

```
program boucle_simd
implicit none
integer(kind=8) :: i
integer(kind=8), parameter :: n=500000
real(kind=8), dimension(n) :: A, B
real(kind=8)               :: somme
...
somme=0
!$OMP SIMD REDUCTION(+:somme)
do i=1,n
   somme=somme+A(i)*B(i)
enddo
...
end program boucle_simd
```

5.3 – Parallélisation et vectorisation SIMD d'une boucle

- La construction DO SIMD est une fusion des directives DO et SIMD munie de l'union de leurs clauses respectives.
- Cette construction permet de partager le travail et de vectoriser le traitement des itérations de la boucle.
- Les paquets d'itérations sont distribués aux threads en fonction du mode de répartition choisi. Chacun vectorise le traitement de son paquet en le subdivisant en bloc d'itérations de la taille des registres vectoriels, blocs qui seront traités l'un après l'autre avec des instructions vectorielles.
- La directive PARALLEL DO SIMD permet en plus de créer la région parallèle.

```
program boucle_simd
implicit none
integer(kind=8) :: i
integer(kind=8), parameter :: n=500000
real(kind=8), dimension(n) :: A, B
real(kind=8)               :: somme
...
somme=0
!$OMP PARALLEL DO SIMD REDUCTION(+:somme)
do i=1,n
   somme=somme+A(i)*B(i)
enddo
...
end program boucle_simd
```

5.4 – Vectorisation SIMD de fonctions scalaires

- Le but est de créer automatiquement une version vectorielle de fonctions scalaires. Les fonctions ainsi générées pourront être appelées à l'intérieur de boucles vectorisées, sans casser la vectorisation.
- La version vectorielle de la fonction permettra de traiter les itérations par bloc et non plus l'une après l'autre...
- La directive DECLARE SIMD permet de générer une version vectorielle en plus de la version scalaire de la fonction dans laquelle elle est déclarée.

```
program boucle_fonction_simd
implicit none
integer, parameter :: n=1000
integer :: i
real, dimension(n) :: A, B
real :: dist_max
...
dist_max=0
!$OMP PARALLEL DO SIMD REDUCTION(max:dist_max)
do i=1,n
   dist_max=max(dist_max,dist(A(i),B(i)))
enddo
!$OMP END PARALLEL DO SIMD

print *,"Distance maximum = ",dist_max

contains

real function dist(x,y)
  !$OMP DECLARE SIMD (dist)
  real, intent(in) :: x, y
  dist=sqrt(x*x+y*y)
end function dist

end program boucle_fonction_simd
```

6 – Les tâches OpenMP

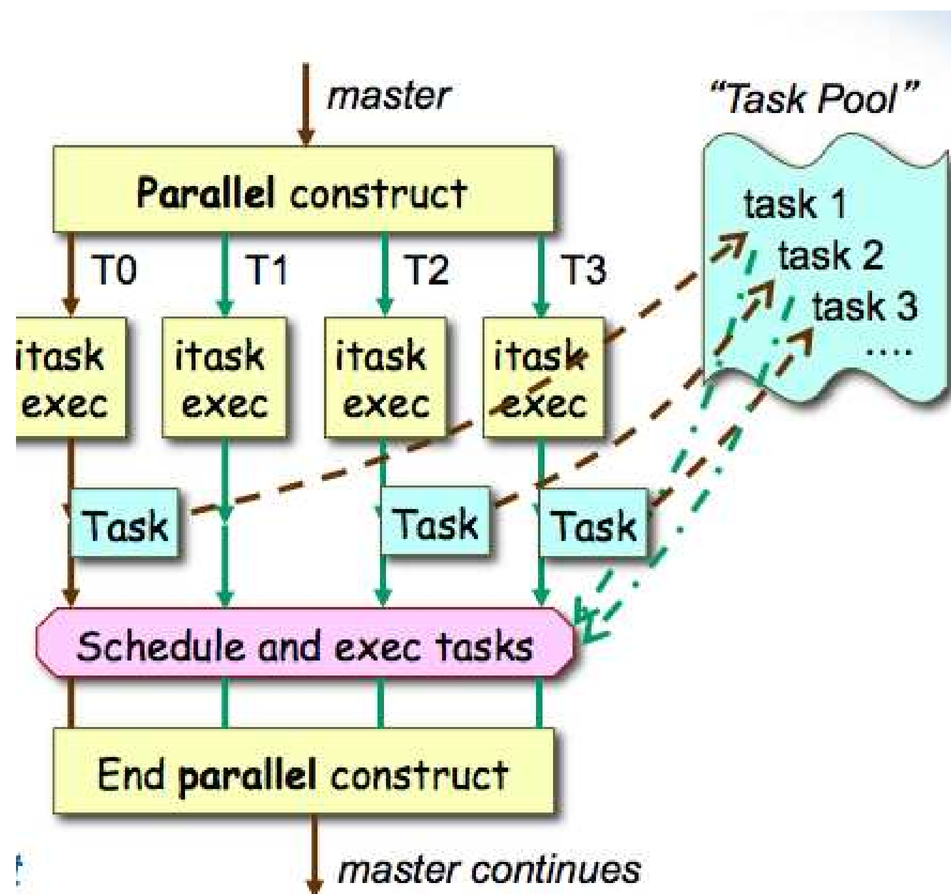
6.1 – Introduction

- Le modèle « *fork and join* » associé aux constructions de partage du travail est limitatif.
- En particulier, il n'est pas adapté aux problématiques dynamiques (boucles while, recherche en parallèle dans un arbre, etc.) ou aux algorithmes récursifs.
- Un mouveau modèle basé sur la notion de tâches a été introduit avec la version `OpenMP` 3.0. Il est complémentaire de celui uniquement axé sur les threads.
- Il permet l'expression du parallélisme pour les algorithmes récursifs ou à base de pointeurs, couramment utilisés en C/C++.
- La version `OpenMP` 4.0 permet de gérer des constructions de génération et de synchronisation de tâches explicites (avec ou sans dépendances).

6.2 – Les bases du concept

- Une tâche OpenMP est constituée d'une instance de code exécutable et de ses données associées. Elle est exécutée par un thread.
- Deux types de tâches existent :
  - Les tâches implicites générées par la directive PARALLEL
  - Les tâches explicites générées par la directive TASK
- Plusieurs types de synchronisation sont disponibles :
  - Pour une tâche donnée, la directive TASKWAIT permet d'attendre la terminaison de tous ses fils (de première génération).
  - La directive TASKGROUP/END TASKGROUP permet d'attendre la terminaison de tous les descendants d'un groupe de tâches.
  - Des barrières implicites ou explicites permettent d'attendre la terminaison de toutes les tâches explicites déjà créées.
- Les variables (et leur statut associé) sont relatives à une tâche, sauf pour la directive THREADPRIVATE qui est, elle, associée à la notion de thread.

6.3 – Le modèle d'exécution des tâches



- L'exécution commence avec le thread master seul.
- À la rencontre d'une région parallèle (PARALLEL) :
  - Création d'une équipe de threads.
  - Création des tâches implicites, une par thread, chaque thread exécutant sa tâche implicite.
- À la rencontre d'une construction de partage du travail :
  - Distribution du travail aux threads (ou aux tâches implicites)
- À la rencontre d'une construction TASK :
  - Création de tâches explicites.
  - L'exécution de ces tâches explicites peut ne pas être immédiate.
- Exécution des tâches explicites :
  - À des points du code appelés *task scheduling point* (TASK, TASKWAIT, BARRIER), les threads disponibles commencent l'exécution des tâches en attente.
  - Un thread peut passer de l'exécution d'une tâche à une autre.
- À la fin de la région parallèle :
  - Toutes les tâches terminent leur exécution.
  - Seul le thread master continue l'exécution de la partie séquentielle.

6.4 – Quelques exemples

```
program task_print
implicit none

print *,"Un "
print *,"grand "
print *,"homme "

end program task_print
```

```
> ifort ... -fopenmp task_print.f90
> export OMP_NUM_THREADS=2 ; a.out
```

```
Un
grand
homme
```

```
program task_print
implicit none

!$OMP PARALLEL
print *,"Un "
print *,"grand "
print *,"homme "
!$OMP END PARALLEL

end program task_print
```

```
> ifort ... -fopenmp task_print.f90
> export OMP_NUM_THREADS=2 ; a.out
```

```
Un
grand
Un
homme
grand
homme
```

```
program task_print
implicit none
!$OMP PARALLEL
!$OMP SINGLE
print *,"Un "
print *,"grand "
print *,"homme "
!$OMP END SINGLE
!$OMP END PARALLEL
end program task_print
```

```
> ifort ... -fopenmp task_print.f90
> export OMP_NUM_THREADS=2 ; a.out
```

```
Un
grand
homme
```

```
program task_print
implicit none
!$OMP PARALLEL
!$OMP SINGLE
print *,"Un "
!$OMP TASK
print *,"grand "
!$OMP END TASK
!$OMP TASK
print *,"homme "
!$OMP END TASK
!$OMP END SINGLE
!$OMP END PARALLEL
end program task_print
```

```
> ifort ... -fopenmp task_print.f90
> export OMP_NUM_THREADS=2 ; a.out; a.out
```

```
Un
grand
homme
```

```
Un
homme
grand
```

- Les tâches peuvent être exécutées dans n'importe quel ordre...
- Comment terminer la phrase par « a marche sur la lune » ?

☞ Si on rajoute un *print* juste avant la fin de la région SINGLE, ca ne marche pas !

☞ En effet, les tâches explicites ne sont exécutables qu'aux *task scheduling point* du code (TASK, TASKWAIT, BARRIER)...

```
program task_print
implicit none
!$OMP PARALLEL
!$OMP SINGLE
print *,"Un "
!$OMP TASK
print *,"grand "
!$OMP END TASK
!$OMP TASK
print *,"homme "
!$OMP END TASK
print *,"a marche sur la lune"
!$OMP END SINGLE
!$OMP END PARALLEL
end program task_print
```

```
> ifort ... -fopenmp task_print.f90
> export OMP_NUM_THREADS=2 ; a.out; a.out
```

```
Un
a marche sur la lune
homme
grand
```

```
Un
a marche sur la lune
grand
homme
```

☞ La solution consiste à introduire un *task scheduling point* avec la directive TASKWAIT pour exécuter les tâches explicites, puis attendre que ces dernières aient terminé avant de continuer.

☞ Si on veut imposer un ordre entre « *grand* » et « *homme* », il faut utiliser la clause DEPEND introduite dans OpenMP 4.0.

```
program task_print
implicit none
!$OMP PARALLEL
!$OMP SINGLE
print *,"Un "
!$OMP TASK
print *,"grand "
!$OMP END TASK
!$OMP TASK
print *,"homme "
!$OMP END TASK
!$OMP TASKWAIT
print *,"a marche sur la lune"
!$OMP END SINGLE
!$OMP END PARALLEL
end program task_print
```

```
> ifort ... -fopenmp task_print.f90
> export OMP_NUM_THREADS=2 ; a.out; a.out
```

```
Un
homme
grand
a marche sur la lune
```

```
Un
grand
homme
a marche sur la lune
```

6.5 – Dépendance entre tâches

- La clause `DEPEND(type_dependance:list)` permet de gérer des dépendances entre des tâches explicites ayant le même père (i.e. générées par la même tâche).
- Une tâche *T1* qui dépend de la tâche *T2* ne pourra commencer à s'exécuter que lorsque l'exécution de *T2* sera terminée.
- Il existe trois types de dépendance :
  - `IN` : la tâche générée sera une tâche dépendante de toutes les tâches précédemment générées par le même père, qui référencent au moins un élément en commun dans la liste de dépendance de type `OUT` ou `INOUT`.
  - `INOUT` et `OUT` : la tâche générée sera une tâche dépendante de toutes les tâches précédemment générées par le même père, qui référencent au moins un élément en commun dans la liste de dépendance de type `IN`, `OUT` ou `INOUT`.
- La liste de variables de la directive `DEPEND` correspond à une adresse mémoire et peut être un élément d'un tableau ou une section de tableau.

- Introduisons une dépendance entre les tâches explicites pour que la tâche *T1 : print *,"grand "* s'exécute avant la tâche *T2 : print *,"homme "*.
- On peut par exemple utiliser la clause DEPEND(OUT:T1) pour la tâche *T1* et DEPEND(IN:T1) pour la tâche *T2*.

```
program task_print
implicit none
integer :: T1
!$OMP PARALLEL
!$OMP SINGLE
print *,"Un "
!$OMP TASK DEPEND(OUT:T1)
print *,"grand "
!$OMP END TASK
!$OMP TASK DEPEND(IN:T1)
print *,"homme "
!$OMP END TASK
!$OMP TASKWAIT
print *,"a marche sur la lune"
!$OMP END SINGLE
!$OMP END PARALLEL
end program task_print
```

```
> ifort ... -fopenmp task_print.f90
> export OMP_NUM_THREADS=2 ; a.out
```

```
Un
grand
homme
a marche sur la lune
```

6.6 – Statut des variables dans les tâches

- Le statut par défaut des variables est :
  - `SHARED` pour les tâches implicites
  - Pour les tâches explicites :
    - Si la variable est `SHARED` dans la tâche père, alors elle hérite de son statut `SHARED`.
    - Dans les autres cas, le statut par défaut est `FIRSTPRIVATE`.
- Lors de la création de la tâche, on peut utiliser les clauses `SHARED(list)`, `PRIVATE(list)`, `FIRSTPRIVATE(list)` ou `DEFAULT(PRIVATE|FIRSTPRIVATE|SHARED|NONE)` (en C/C++ uniquement `DEFAULT(PRIVATE|NONE)`) pour spécifier explicitement le statut des variables qui apparaissent lexicalement dans la tâche.

6.7 – Exemple de MAJ des éléments d'une liste chaînée

☞ Étant donnée une liste chaînée, comment mettre à jour tous les éléments de cette liste en parallèle...

```
type element
integer :: valeur
type(element), pointer :: next
end type element

subroutine increment_lst_ch(debut)
type(element), pointer :: debut, p
p=>debut
do while (associated(p))
p%valeur=p%valeur+1
p=>p%next
end do
end subroutine increment_lst_ch
```

☞ Schéma de type producteur/consommateur (thread qui exécute la région `single`/les autres threads)

```
subroutine increment_lst_ch(debut)
type(element), pointer :: debut, p
!$OMP PARALLEL PRIVATE(p)
!$OMP SINGLE
p=>debut
do while (associated(p))
!$OMP TASK
p%valeur=p%valeur+1
!$OMP END TASK
p=>p%next
end do
!$OMP END SINGLE
!$OMP END PARALLEL
end subroutine increment_lst_ch
```

☞ Le statut de la variable *p* à l'intérieur de la tâche explicite est `FIRSTPRIVATE` par défaut, ce qui est le statut voulu.

6.8 – Exemple d'algorithme récursif

- La suite de Fibonacci est définie par : f(0)=0 ; f(1)=1 ; f(n)=f(n-1)+f(n-2)
- Le code construit un arbre binaire. La parallélisme provient du traitement des feuilles de cet arbre en parallèle.
- Un seul thread va générer les tâches, mais l'ensemble des threads vont participer à l'exécution.
- Attention au statut des variables dans cet exemple : le statut par défaut (i.e. `FIRSTPRIVATE`) donnerait des résultats faux. Il faut nécessairement que i et j soient partagées pour pouvoir récupérer le résultat dans la tâche père...
- Attention, la directive `TASKWAIT` est aussi obligatoire pour s'assurer que les calculs de i et j soient terminés avant de retourner le résultat.
- Cette version n'est pas performante...

```
program fib_rec
integer, parameter :: nn=10
integer :: res_fib
!$OMP PARALLEL
!$OMP SINGLE
res_fib=fib(nn)
!$OMP END SINGLE
!$OMP END PARALLEL
print *,"res_fib = ",res_fib
contains
recursive integer function fib(n) &
result(res)
integer, intent(in) :: n
integer :: i, j
if (n<2) then res = n
else
!$OMP TASK SHARED(i)
i=fib(n-1)
!$OMP END TASK
!$OMP TASK SHARED(j)
j=fib(n-2)
!$OMP END TASK
!$OMP TASKWAIT
res=i+j
endif
end function fib
end program fib_rec
```

6.9 – Clauses FINAL et MERGEABLE

- Dans le cas d'algorithmes récursifs de type « *Divide and Conquer* », le volume du travail de chaque tâche (i.e. la granularité) diminue au fil de l'exécution. C'est la principale raison pour laquelle le code précédent n'est pas performant.
- Les clauses `FINAL` et `MERGEABLE` sont alors très utiles : elles permettent au compilateur de pouvoir fusionner les nouvelles tâches créées.
- Malheureusement, ces fonctionnalités ne sont que très rarement implémentées de façon efficace, aussi vaut-il mieux mieux avoir recours à un « *cut off* » manuel dans le code...

6.10 – Synchronisation de type TASKGROUP

- La construction TASKGROUP permet de définir un groupe de tâches et d'attendre en fin de construction que toutes ces tâches, ainsi que leurs descendantes, aient terminé leur execution.
- Dans cet exemple, nous allons particulariser une tâche qui va effectuer un calcul en tâche de fond pendant que sont lancées en parallèle plusieurs itérations de la traversée d'un arbre binaire. A chacune des itérations, on synchronise les tâches ayant été générées pour la traversée de l'arbre et uniquement celles-ci.

```
module arbre_mod
type type_arbre
type(type_arbre), pointer :: fg, fd
end type
contains
subroutine traitement_feuille(feuille)
type(type_arbre), pointer :: feuille
! Traitement...
end subroutine traitement_feuille
recursive subroutine traverse_arbre(arbre)
type(type_arbre), pointer :: arbre
if (associated(arbre%fg)) then
!$OMP TASK
call traverse_arbre(arbre%fg)
!$OMP END TASK
endif
if (associated(arbre%fd)) then
!$OMP TASK
call traverse_arbre(arbre%fd)
!$OMP END TASK
endif
!$OMP TASK
call traitement_feuille(arbre)
!$OMP END TASK
end subroutine traverse_arbre
end module arbre_mod
```

```
program principal
use arbre_mod
type(type_arbre), pointer :: mon_arbre
integer, parameter :: niter=100
call init_arbre(mon_arbre)
!$OMP PARALLEL
!$OMP SINGLE
!$OMP TASK
call travail_tache_de_fond()
!$OMP END TASK
do i=1, niter
!$OMP TASKGROUP
!$OMP TASK
call traverse_arbre(mon_arbre)
!$OMP END TASK
!$OMP END TASKGROUP
enddo
!$OMP END SINGLE
!$OMP END PARALLEL
end program principal
```

7 – Affinités

7.1 – Affinité des threads

- Par défaut, le système d'exploitation choisit le cœur d'exécution d'un thread. Celui-ci peut changer en cours d'exécution, au prix d'une forte pénalité.
- Pour pallier ce problème, il est possible d'associer explicitement un thread à un cœur pendant toute la durée de l'exécution : c'est ce que l'on appelle le *binding*.
- Avec les compilateurs GNU, l'association thread/cœur d'exécution se fait avec la variable d'environnement *GOMP_CPU_AFFINITY*.
- Avec les compilateurs Intel, l'association thread/cœur d'exécution se fait avec la variable d'environnement *KMP_AFFINITY* (cf. Intel Thread Affinity Interface).
- Depuis `OpenMP`4.0, l'association thread/cœur d'exécution peut se faire de façon portable avec les variables d'environnement *OMP_PROC_BIND* et *OMP_PLACES*.

7.1.1 – Commande *cpuinfo*

☞ La commande *cpuinfo* permet d'obtenir de nombreuses informations sur la topologie du nœud d'exécution (nombre et numérotation des sockets, des cœurs physiques et logiques, activation ou non de l'hyperthreading, etc.).

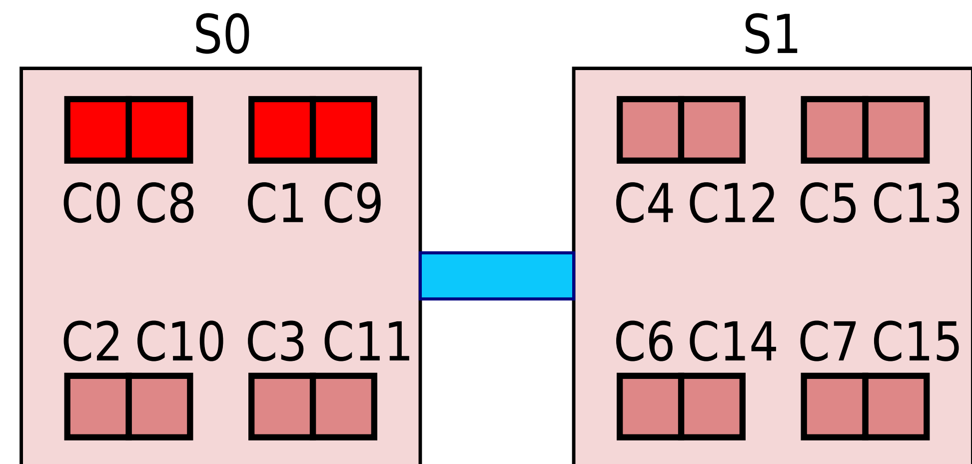
```
> cpuinfo     <= Exemple sur un noeud SMP sans l'hyperthreading activé
Intel(R) Processor information utility, Version 4.1.0 Build 20120831
Copyright (C) 2005-2012 Intel Corporation.  All rights reserved.

=====  Processor composition  =====
Processor name    : Intel(R) Xeon(R)  E5-4650 0
Packages(sockets) : 4      <= Nb de sockets du noeud
Cores             : 32     <= Nb de coeurs physiques du noeud
Processors(CPUs)  : 32     <= Nb de coeurs logiques du noeud
Cores per package : 8      <= Nb de coeurs physiques par socket
Threads per core  : 1      <= Nb de coeurs logiques par coeur physique,hyperthreading actif si valeur >1

=====  Processor identification  =====
Processor       Thread Id.      Core Id.        Package Id.
0               0               0               0
1               0               1               0
2               0               2               0
3               0               3               0
4               0               4               0
5               0               5               0
6               0               6               0
7               0               7               0
8               0               0               1
9               0               1               1
...             ...             ...             ...
30              0               6               3
31              0               7               3
=====  Placement on packages  =====
Package Id.     Core Id.        Processors
0               0,1,2,3,4,5,6,7         0,1,2,3,4,5,6,7
1               0,1,2,3,4,5,6,7         8,9,10,11,12,13,14,15
2               0,1,2,3,4,5,6,7         16,17,18,19,20,21,22,23
3               0,1,2,3,4,5,6,7         24,25,26,27,28,29,30,31

=====  Cache sharing  =====
Cache   Size            Processors
L1      32  KB          no sharing
L2      256 KB          no sharing
L3      20  MB          (0,1,2,3,4,5,6,7)(8,9,10,11,12,13,14,15)(16,17,18,19,20,21,22,23)(24,25,26,27,28,29,30,31)
```

```
> cpuinfo     <= Exemple sur un noeud SMP avec l'hyperthreading activé
Intel(R) Processor information utility, Version 4.1.0 Build 20120831
Copyright (C) 2005-2012 Intel Corporation.  All rights reserved.

=====  Processor composition  =====
Processor name     : Intel(R) Xeon(R)  E5-4650 0
Packages(sockets) : 4     <= Nb de sockets du noeud
Cores              : 32    <= Nb de coeurs physiques du noeud
Processors(CPUs)  : 64    <= Nb de coeurs logiques du noeud
Cores per package : 8     <= Nb de coeurs physiques par socket
Threads per core  : 2     <= Nb de coeurs logiques par coeur physique,hyperthreading actif si valeur >1

=====  Processor identification  =====
Processor        Thread Id.       Core Id.        Package Id.
0                0                0               0
1                0                1               0
2                0                2               0
3                0                3               0
4                0                4               0
5                0                5               0
6                0                6               0
7                0                7               0
8                0                0               1
9                0                1               1
10               0                2               1
...
54               1                6               2
55               1                7               2
56               1                0               3
57               1                1               3
58               1                2               3
59               1                3               3
60               1                4               3
61               1                5               3
62               1                6               3
63               1                7               3
=====  Placement on packages  =====
Package Id.      Core Id.         Processors
0                0,1,2,3,4,5,6,7          (0,32)(1,33)(2,34)(3,35)(4,36)(5,37)(6,38)(7,39)
1                0,1,2,3,4,5,6,7          (8,40)(9,41)(10,42)(11,43)(12,44)(13,45)(14,46)(15,47)
2                0,1,2,3,4,5,6,7          (16,48)(17,49)(18,50)(19,51)(20,52)(21,53)(22,54)(23,55)
3                0,1,2,3,4,5,6,7          (24,56)(25,57)(26,58)(27,59)(28,60)(29,61)(30,62)(31,63)

=====  Cache sharing  =====
Cache   Size            Processors
L1      32  KB          (0,32)(1,33)(2,34)(3,35)(4,36)(5,37)(6,38)(7,39)(8,40)(9,41)(10,42)(11,43)(12,44)(13,45)(14,46)(15,47)
(16,48)(17,49)(18,50)(19,51)(20,52)(21,53)(22,54)(23,55)(24,56)(25,57)(26,58)(27,59)(28,60)(29,61)(30,62)(31,63)
L2      256 KB          (0,32)(1,33)(2,34)(3,35)(4,36)(5,37)(6,38)(7,39)(8,40)(9,41)(10,42)(11,43)(12,44)(13,45)(14,46)(15,47)
(16,48)(17,49)(18,50)(19,51)(20,52)(21,53)(22,54)(23,55)(24,56)(25,57)(26,58)(27,59)(28,60)(29,61)(30,62)(31,63)
L3      20  MB          (0,1,2,3,4,5,6,7,32,33,34,35,36,37,38,39)(8,9,10,11,12,13,14,15,40,41,42,43,44,45,46,47)
(16,17,18,19,20,21,22,23,48,49,50,51,52,53,54,55)(24,25,26,27,28,29,30,31,56,57,58,59,60,61,62,63)
```

7.1.2 – Utilisation de la variable d'environnement *KMP_AFFINITY*

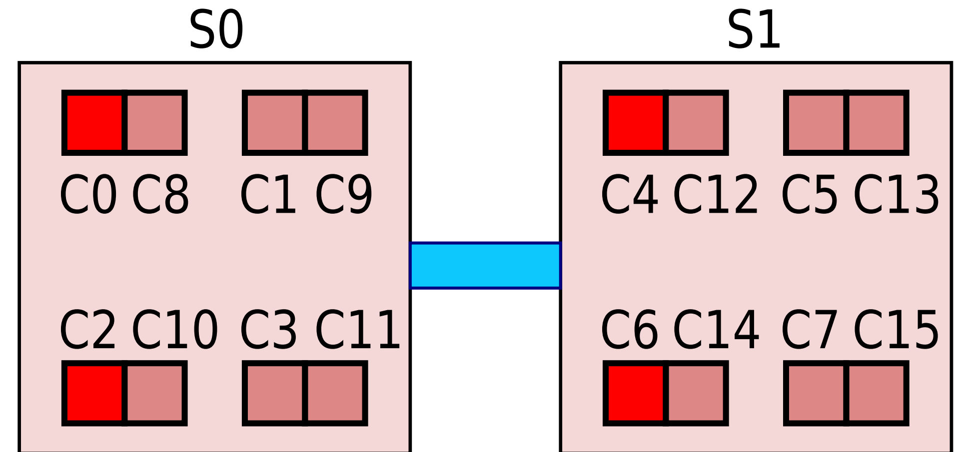
Les principaux modes d'association thread/cœur d'exécution sont les suivants :

☞ mode *compact* : les threads de numéros consécutifs sont bindés sur des cœurs logiques ou physiques (suivant que l'hyperthreading est activé ou non) qui sont les plus proches possibles les uns des autres. Cela permet de réduire les défauts de cache et de TLB (Translation lookaside buffer).

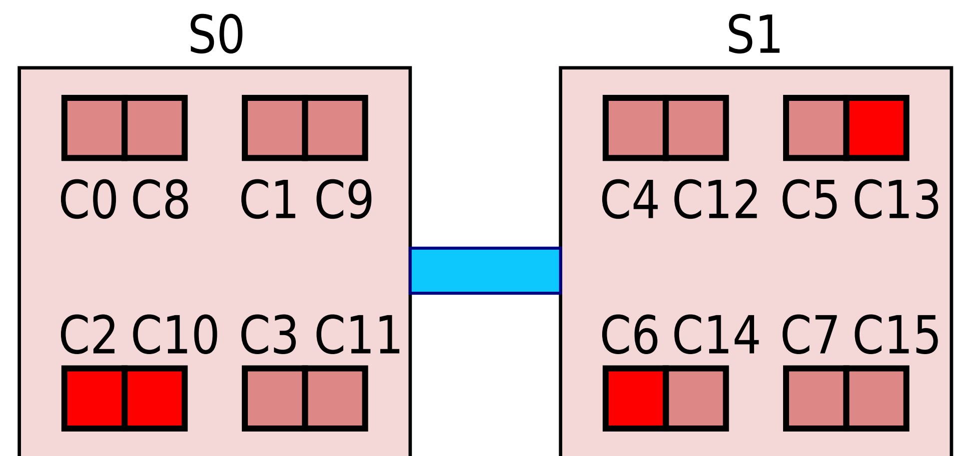


Exemple sur une architecture bi-sockets quadri-cœurs, avec l'hyperthreading activé.

☞ mode *scatter* : c'est le contraire du mode *compact*, les threads de numéros consécutifs sont bindés sur des cœurs logiques ou physiques (suivant que l'hyperthreading est activé ou non) qui sont les plus éloignés les uns des autres.



☞ mode *explicit* : on définit explicitement le binding des threads sur les cœurs logiques ou physiques.



7.1.3 – Affinité des threads avec OpenMP 4.0

- OpenMP 4.0 introduit la notion de *places* qui définissent des ensembles de coeurs logiques ou physiques qui seront associés à l'exécution d'un thread.
- Les *places* peuvent être définies explicitement par l'intermédiaire d'une liste, ou directement avec les mots clés suivants :
  - threads : chaque *place* correspond à un coeur logique de la machine,
  - cores : chaque *place* correspond à un coeur physique de la machine,
  - sockets : chaque *place* correspond à un socket de la machine.
- Exemples pour une architecture bi-sockets quadri-coeurs avec hyperthreading :
  - OMP_PLACES=threads : 16 *places* correspondant a un coeur logique
  - OMP_PLACES="threads(4)" : 4 *places* correspondant a un coeur logique
  - OMP_PLACES="{0,8,1,9},{6,14,7,15}" : 2 *places*, la premiere sur le premier socket, la seconde sur le deuxieme.

☞ La clause PROC_BIND de la construction PARALLEL ou la variable d'environnement OMP_PROC_BIND permettent de choisir l'affinité parmi les choix suivants :

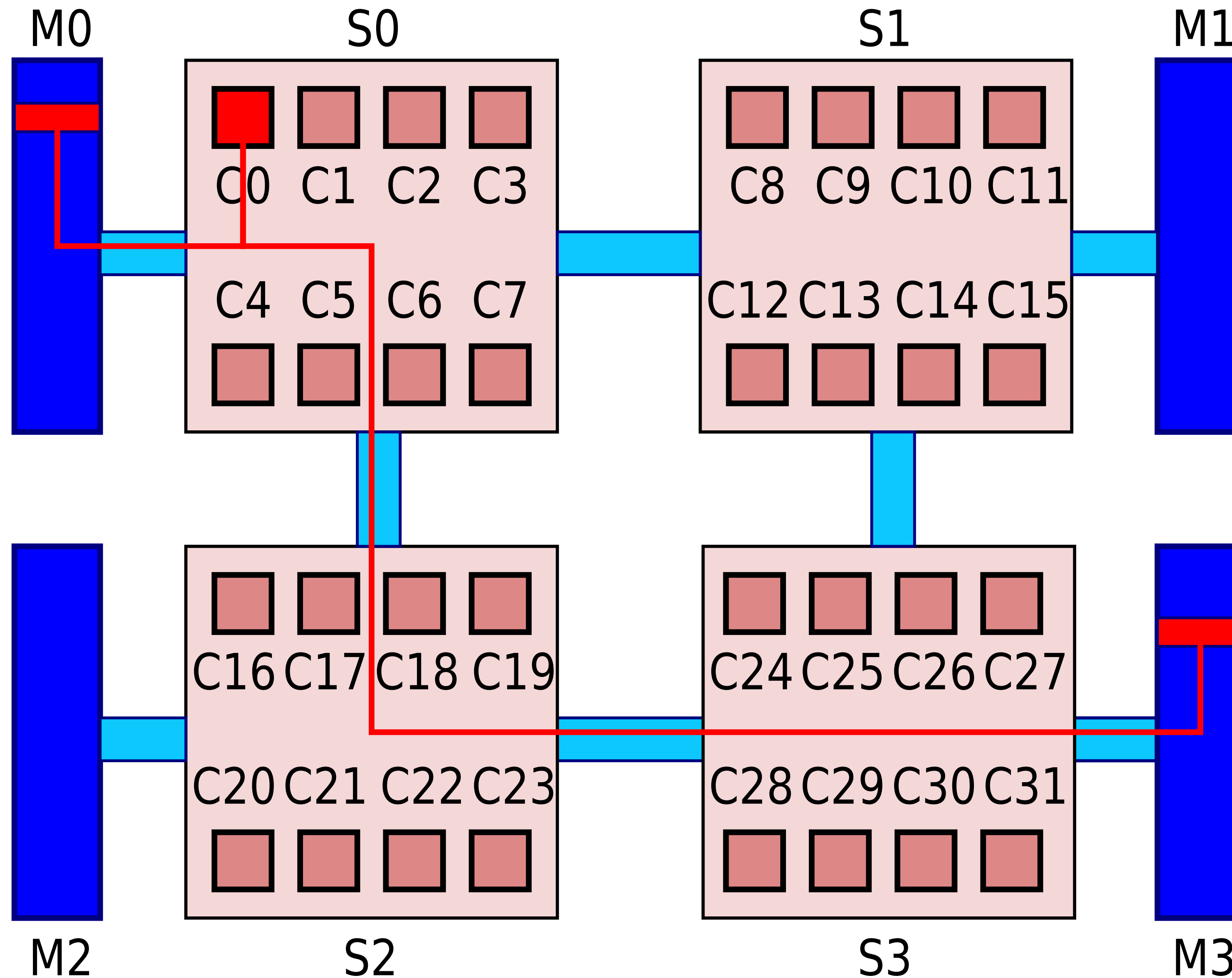
- MASTER (PRIMARY à partir d'OpenMP 5) : les threads s'exécutent sur la même *place* que celle du thread maître
- CLOSE : répartition des threads sur les *places* les plus proches de celle du thread maître
- SPREAD : répartition équitable des threads sur les différentes *places* définies.

```
export OMP_PLACES="{0,8,1,9},{2,10,3,11},{4,12,5,13},{6,14,7,15}"
Soit 4 places p0={0,8,1,9}, p1={2,10,3,11}, p2={4,12,5,13} et p3={6,14,7,15}
!$OMP PARALLEL PROC_BIND(SPREAD) NUM_THREADS(2)
   !$OMP PARALLEL PROC_BIND(CLOSE) NUM_THREADS(4)
....
Dans la premiere region parallele
Th0 s'executera sur p0 avec une partition de place =p0p1
Th1 s'executera sur p2 avec une partition de place =p2p3
Dans la seconde region parallele
Th00 et Th01 s'executeront sur p0
Th02 et Th03 s'executeront sur p1
Th10 et Th11 s'executeront sur p2
Th12 et Th13 s'executeront sur p3
```

7.2 – Affinité mémoire

- Les nœuds multi-socket modernes sont fortement `NUMA` (*Non Uniform Memory Access*), le temps d'accès à une donnée est variable suivant l'emplacement du banc mémoire où elle est stockée.
- La localité du stockage en mémoire des variables partagées (sur la mémoire locale du socket qui exécute le thread ou sur la mémoire distante d'un autre socket) va fortement influer sur les performances du code.
- Le système d'exploitation essaie d'optimiser ce processus d'allocation mémoire en privilégiant, lorsque cela est possible, l'allocation dans la mémoire locale du socket qui est en charge de l'exécution du thread. C'est ce que l'on appelle l'*affinité mémoire*.

Architecture simplifiée d'une machine fortement `NUMA` (quadri-sockets, octo-cœurs).



- Pour les tableaux, l'allocation réelle de la mémoire se fait à l'exécution, page par page, lors du premier accès à un élément de ce tableau.
- Suivant les caractéristiques des codes (memory bound, CPU bound, accès mémoire aléatoires, accès mémoire suivant une dimension privilégiée, etc.), il vaut mieux regrouper tous les threads au sein du même socket (répartition de type *compact*) ou au contraire les répartir sur les différents sockets disponibles (répartition de type *scatter*).
- En général, on essaiera de regrouper sur un même socket des threads qui travaillent sur les mêmes données partagées.

7.3 – Stratégie « *First Touch* »

- Pour optimiser l'affinité mémoire dans une application, il est très fortement recommandé d'implémenter une stratégie de type « *First Touch* » : chaque thread va initialiser la partie des données partagées sur lesquelles il va travailler ultérieurement.
- Si les threads sont bindés, on optimise ainsi les accès mémoire en privilégiant la localité des accès.
- Avantage : gains substantiels en terme de performance.
- Inconvénient :
  - aucun gain à escompter avec les scheduling `DYNAMIC` et `GUIDED` ou avec la directive `WORKSHARE`...
  - aucun gain à escompter si la parallélisation utilise le concept des tâches explicites.

7.4 – Exemples d'impact sur les performances

☞ Code « *Memory Bound* » s'exécutant avec 4 threads sur des données privées.

```
program SBP
...
!$OMP PARALLEL PRIVATE(A,B,C)
do i=1,n
   A(i) = A(i)*B(i)+C(i)
enddo
!$OMP END PARALLEL
...
end program SBP
```

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

```
Temps elapsed = 116 s.
```

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

```
Temps elapsed = 49 s.
```

☞ Pour optimiser l'utilisation des 4 bus mémoire, il est donc préférable de binder un thread par socket. Ici le mode *scatter* est 2.4 fois plus performant que le mode *compact* !

☞ Exemple sans « *First Touch* »

```
program NoFirstTouch
 implicit none
 integer, parameter   :: n = 30000
 integer              :: i, j
 real, dimension(n,n) :: TAB

 ! Initialisation de TAB
 TAB(1:n,1:n)=1.0

 !$OMP PARALLEL
 ! Calcul sur TAB
 !$OMP DO SCHEDULE(STATIC)
 do j=1,n
    do i=1,n
       TAB(i,j)=TAB(i,j)+i+j
    enddo
 enddo
 !$OMP END PARALLEL
end program NoFirstTouch
```

```
> export OMP_NUM_THREADS=32 ; a.out
```

```
Temps elapsed = 98.35 s.
```

☞ Exemple avec « *First Touch* »

```
program FirstTouch
 implicit none
 integer, parameter   :: n = 30000
 integer              :: i, j
 real, dimension(n,n) :: TAB
 !$OMP PARALLEL
 ! Initialisation de TAB
 !$OMP DO SCHEDULE(STATIC)
 do j=1,n
   TAB(1:n,j)=1.0
 enddo
 ! Calcul sur TAB
 !$OMP DO SCHEDULE(STATIC)
 do j=1,n
    do i=1,n
       TAB(i,j)=TAB(i,j)+i+j
    enddo
 enddo
 !$OMP END PARALLEL
end program FirstTouch
```

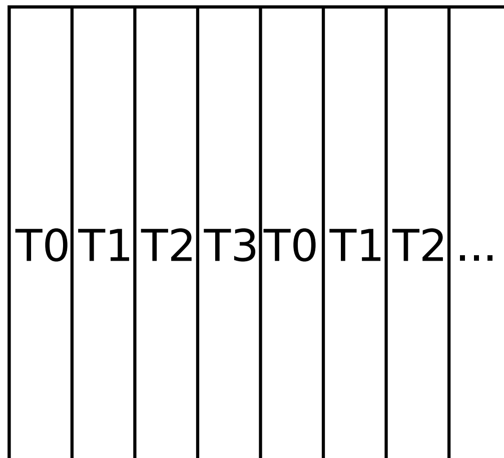
```
> export OMP_NUM_THREADS=32 ; a.out
```

```
Temps elapsed = 10.22 s.
```

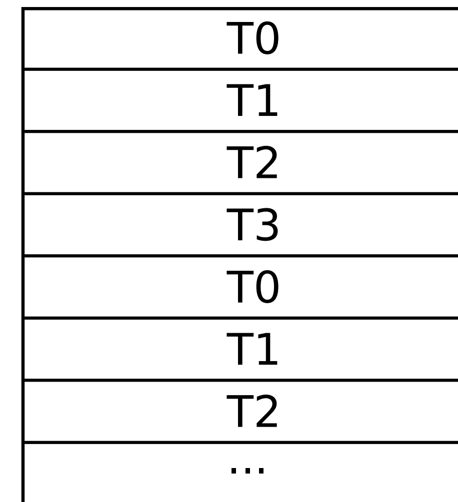
☞ L'utilisation de la stratégie de type « *First Touch* » permet un gain de l'ordre d'un facteur 10 sur cet exemple !

- Code de type « directions alternées » s'exécutant avec 4 threads sur un tableau 2D partagé, tenant dans le cache L3 d'un socket. C'est un exemple pour lequel il n'y a pas de localité *thread d'exécution/donnée*.
  - Aux itérations paires, chaque thread travaille sur des colonnes du tableau partagé.
  - Aux itérations impaires, chaque thread travaille sur des lignes du tableau partagé.

Itération paire

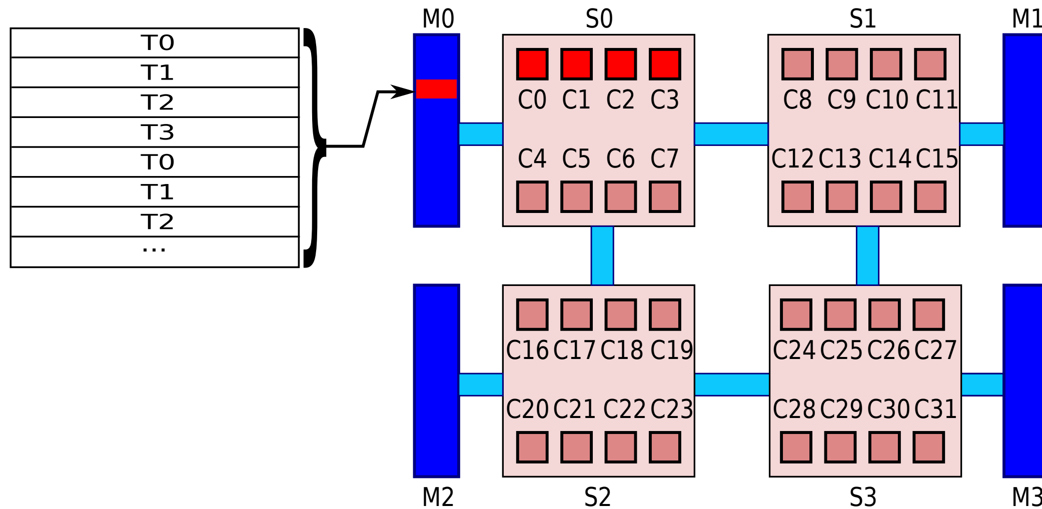


Itération impaire



- La stratégie « *First Touch* » est utilisée.
- On va comparer un binding de type *compact* avec un binding de type *scatter*.

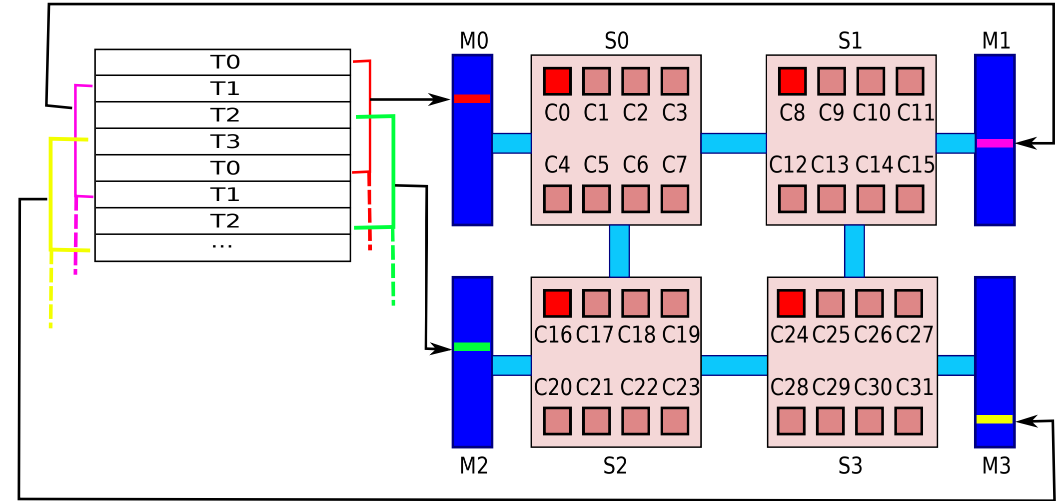
☞ Binding de type *compact*




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

```
Temps elapsed = 33.46 s.
```

☞ Binding de type *scatter*




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

```
Temps elapsed = 171.52 s.
```

☞ Dans cet exemple, le mode *compact* est plus de 5 fois plus performant que le mode *scatter* !

8 – 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.

8.1 – Règles de bonnes performances

- Vérifier que le mécanisme de binding des threads sur les cœurs d'exécution est bien opérationnel.
- Minimiser le nombre de régions parallèles dans le code.
- Adapter le nombre de threads demandé à la taille du problème à traiter, afin de minimiser les surcoûts de gestion des threads 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 dans leur usage mais plus performantes que la directive `CRITICAL`.

- Utiliser la clause **IF** pour mettre en place une parallélisation conditionnelle (p. 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 peuvent dégrader sensiblement les performances (conflits de banc mémoire sur une machine vectorielle ou de défauts de cache sur une machine scalaire).
- Sur les machines de type `NUMA`, il faut optimiser l'affinité mémoire en utilisant la stratégie « *First Touch* ».
- Indépendamment de l'architecture des machines, la qualité de l'implémentation `OpenMP` peut affecter assez sensiblement l'extensibilité des boucles parallèles.

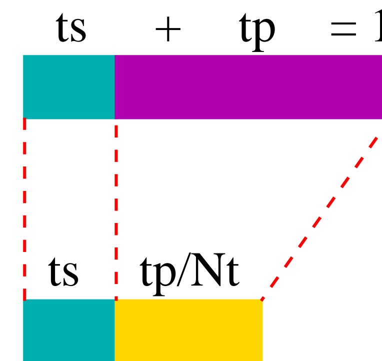
8.2 – Mesures du temps

- 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 dans le 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.

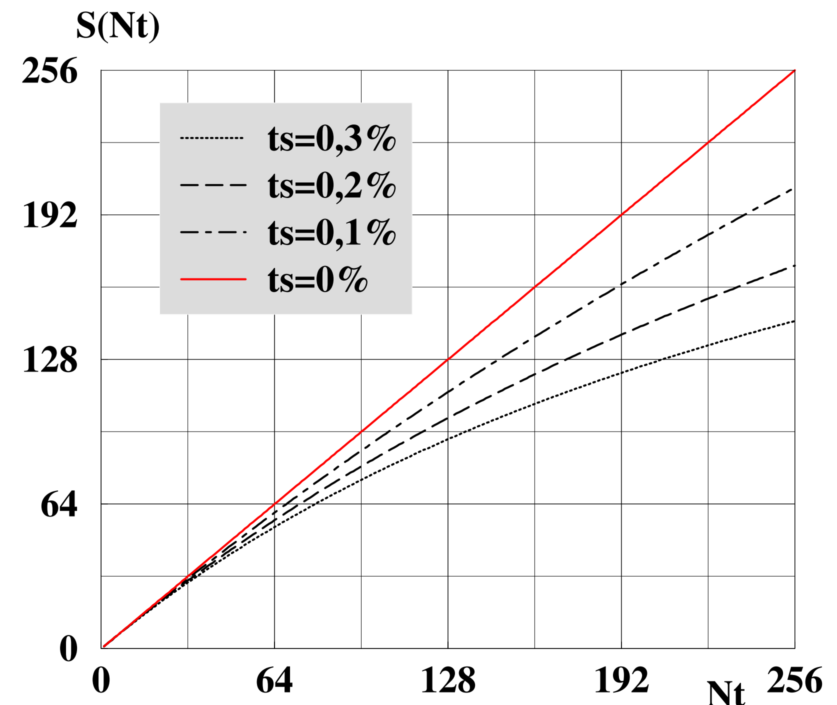
```
program mat_vect
 !$ use OMP_LIB
 implicit none
 integer,parameter   :: n=1025
 real,dimension(n,n) :: a
 real,dimension(n)   :: x, y
 real(kind=8) :: t_ref, t_final
 integer :: rang
 call random_number(a)
 call random_number(x) ; y(:)=0.
 !$OMP PARALLEL &
  !$OMP PRIVATE(rang,t_ref,t_final)
  rang = 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
```

8.3 – 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.



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



9 – 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, aux sections parallèles et aux tâches. Mais on peut aussi singulariser un thread 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.

10 – Annexes

10.1 – Parties non abordées ici

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

- les procédures « verrou » pour la synchronisation point à point ;
- d'autres sous-programmes de service ;
- la parallélisation mixte MPI & OpenMP ;
- les apports d'OpenMP 4.0 relatifs à l'utilisation des accélérateurs.

10.2 – 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> 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 une autre plateforme. Une 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> 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
```

☞ Dans le troisième exemple ci-contre, il peut se produire un blocage de type « deadlock » dû à une désynchronisation entre les tâches (une tâche ayant du retard peut sortir de la boucle, alors que les autres tâches ayant de l'avance attendent indéfiniment sur la barrière implicite de la construction SINGLE). La solution consiste à rajouter une barrière, soit avant la construction SINGLE, soit après le test « if ».

```
program faux_3
  implicit none
  integer :: iteration=0

  !$OMP PARALLEL
  do
     !$OMP SINGLE
     iteration = iteration + 1
     !$OMP END SINGLE
     if( iteration >= 3 ) exit
  end do
  !$OMP END PARALLEL
  print *,"Outside // region"
end program faux_3
```

```
 Intel> export OMP_NUM_THREADS=3;a.out
... rien ne s'affiche à l'écran ...
```