

OpenMP

Parallélisation multitâches pour machine à mémoire partagée

v23.03

Intervenants :

Rémi Lacroix

Thibaut Véry

Contact : prenom.nom at idris .fr

Auteurs originaux :

Jalel Chergui

Pierre-Francois Lavallée



①	Introduction	3
	Historique	4
	Spécifications	5
	Terminologie et définitions	6
	Concepts généraux	7
	Modèle d'exécution	7
	Processus légers	8
	Fonctionnalités	11
	OpenMP versus MPI	12
	Bibliographie	14
②	Principes	15
	Interface de programmation	15
	Syntaxe générale	15
	Compilation	18
	Construction d'une région parallèle	19
	Statut des variables	21
	Variables privées	21
	Clause DEFAULT	23
	Allocation dynamique	24
	Équivalence variables Fortran	26
	Étendue d'une région parallèle	27
	Transmission par arguments	29
	Variables statiques	30
	Compléments	32
	TP	34

③ Partage du travail	35
Introduction	35
Boucle parallèle	36
Clause SCHEDULE	37
Exécution ordonnée : Directive et Clause ORDERED	42
Clause REDUCTION	43
Fusion d'un nid de boucles : Clause COLLAPSE	44
Compléments	47
TP	48
Construction WORKSHARE	49
TP	52
Sections parallèles	53
Construction SECTIONS	53
Compléments	55
Exécution exclusive	56
Construction SINGLE	57
Construction MASTER	59
Procédures orphelines	60
Récapitulatif	62

4	Synchronisations	63
	Barrière explicite : BARRIER	65
	Mise à jour atomique : ATOMIC	66
	Régions critiques : CRITICAL	68
	TP	70
	Directive FLUSH	71
	Exemple avec un piège facile	72
	Exemple avec un piège difficile	73
	Commentaires sur les exemples précédents	74
	Code corrigé	75
	Nid de boucles avec dépendances	76
	Récapitulatif	82
	TP	83
5	Vectorisation SIMD	84
	Vectorisation d'une boucle	85
	Parallélisation ET vectorisation d'une boucle	86
	Vectorisation de fonction scalaire	87
6	Les tâches OpenMP : TASK	88
	Les bases du concept	89
	Modèle d'exécution	90
	Quelques exemples	92
	Dépendances entre tâches	96
	Statut des variables	98
	Exemple de MAJ des éléments d'une liste chaînée	99
	Exemple d'algorithme récursif	100
	Clauses MERGEABLE et FINAL	101
	Synchronisation de type TASKGROUP	102
	TP	104

7	Affinités	105
	Affinité des threads	105
	Commande <i>cpuinfo</i>	106
	Utilisation de <i>KMP_AFFINITY</i>	108
	Affinité des threads avec OpenMP 4.0	110
	Affinité mémoire	112
	Stratégie <i>First Touch</i>	115
	Exemples d'impact sur les performances	116
	TP	120
8	Performances	121
	Règles de bonnes performances	122
	Mesures du temps	124
	Accélération	125
9	Conclusion	126
10	Annexes	128
	Parties non abordées ici	128
	Quelques pièges	129

1 - Introduction

1 Introduction

- **OpenMP** est un modèle de programmation parallèle qui initialement 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 - Historique

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 - Spécifications **OpenMP**

1.2 Spécifications

- 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 concentrent 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 - Terminologies et définitions

1.3 Terminologie et définitions

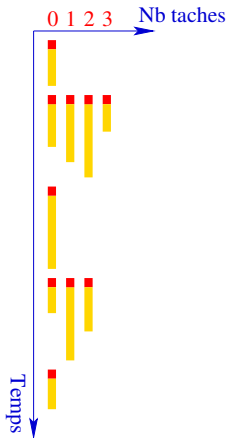
- **Thread** : Entité d'exécution avec une mémoire locale (*stack*)
- **Team** : Un ensemble de un ou plusieurs *threads* qui participent à l'exécution d'une région parallèle.
- **Task/Tâche** : Une instance de code exécutable et ses données associées. Elles sont générées par les constructions **PARALLEL** 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 - Concepts Généraux - Modèle d'exécution

1.4 Concepts généraux

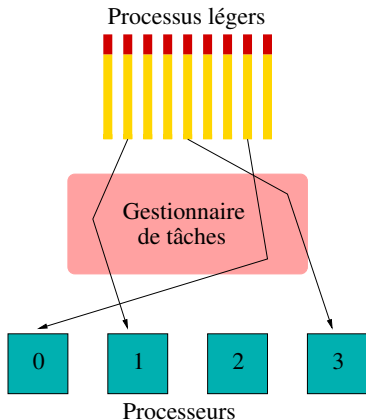
1.4.1 Modèle d'exécution

- À son lancement, un programme **OpenMP** est séquentiel. Il est constitué d'un processus unique, le thread maître dont le rang vaut 0, qui exécute 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.



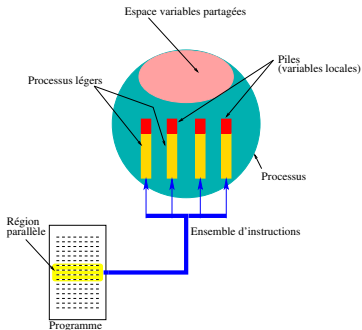
1.4.2 Processus légers

- 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.

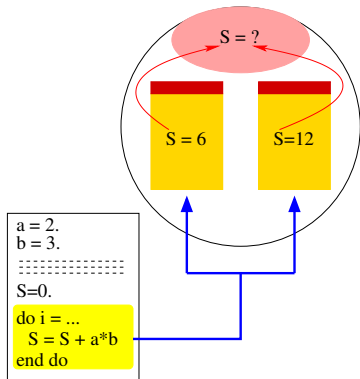


1 - Concepts Généraux - Processus légers (*threads*) suite

- 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.

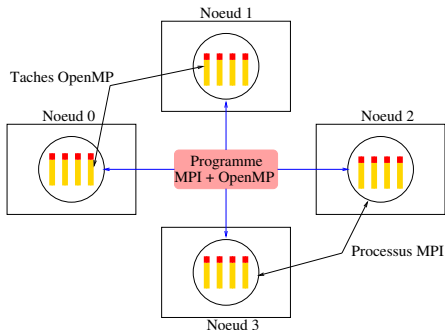
1.6 OpenMP versus MPI

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.

1 - OpenMP vs MPI - Modèles de programmation

- 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

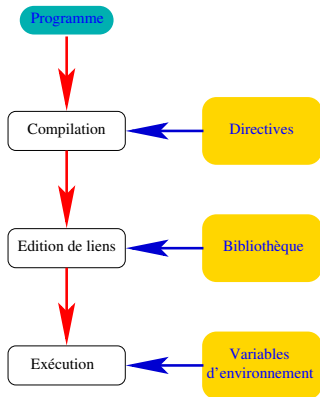
- Premier livre sur **OpenMP** : R. CHANDRA & *al.*, *Parallel Programming in OpenMP*, éd. Morgan Kaufmann Publishers, oct. 2000.
- Livre plus récent sur **OpenMP** : B. CHAPMAN & *al.*, *Using OpenMP*, MIT Press, 2008.
- Un livre plus récent : R. VAN DER PAS, E. STOTZER & Ch. TERBOVEN, *USING OPENMP - THE NEXT STEP Affinity, Accelerators, Tasking, and SIMD*, MIT Press, 2017.
- Spécifications du standard **OpenMP** : <https://www.openmp.org/>

2 - Principes - Interface de programmation

2.1 Interface de programmation

2.1.1 Syntaxe générale

- **Directives et clauses de compilation** : elles servent à définir le partage du travail, la synchronisation et le statut privé ou partagé des données ;
- **Fonctions et sous-programmes** : ils font partie d'une bibliothèque chargée à l'édition de liens du programme.
- **Variables d'environnement** : une fois positionnées, leurs valeurs sont prises en compte à l'exécution.



- 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.

2 - Principes - Interface de programmation

- Pour Fortran, en format libre :

```
!$ use OMP_LIB
...
! C'est un commentaire
!$OMP PARALLEL PRIVATE(a,b) &
    !$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**

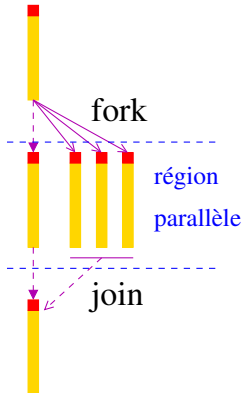
```
pgfortran/nvfortran -mp prog.f90 # Compilateur Fortran
```

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.



2 - Principes - Construction d'une région parallèle

- 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.

```
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 *, "Parallele ?:", p
end program parallel
```

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

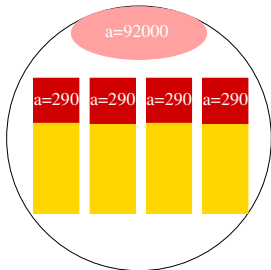
```
A vaut : 92290
A vaut : 92290
A vaut : 92290
A vaut : 92290
Parallelele ? : T
```

2 - Principes - Statut des variables

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.



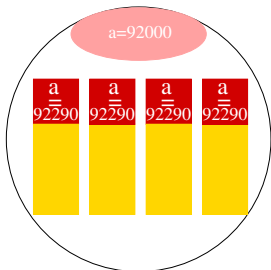
```
program parallele
!$ use OMP_LIB
implicit none
real      :: a
integer   :: rang

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 parallele
```

```
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
```


2 - Principes - Statut des variables

- 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 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*,"Parallele ?:", 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").

2 - Principes - Statut des variables

```
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 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
```

2 - Principes - Statut des variables

- 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 - Principes - Transmission par arguments

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 - Principes - Variables statiques

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
```

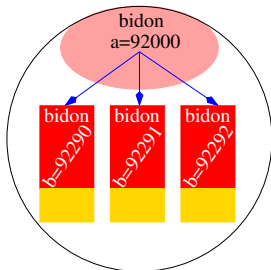
```
ifort -fopenmp var_stat.f90 prog.f90
export OMP_NUM_THREADS=2
./a.out
```

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
```

2 - Principes - Variables statiques

- 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 !
```

2 - Principes - Compléments

- 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 parallele
  !$ 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 parallele
```

```
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
```

2 - À vous de jouer

Liste des TP que vous pouvez faire :

- TP 0 : Hello World

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 - Partage du travail - Boucle parallèle

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**

- La clause **SCHEDULE** permet de définir comment les itérations vont être réparties entre les threads.
- Il est possible de définir 3 modes:

```
!$OMP DO SCHEDULE(mode, batch_size)
```

(STATIC, batch_s)

- Paquets de taille identique (batch_s)¹
- Attribués au début

(DYNAMIC, batch_s)

- Paquets de taille identique (batch_s)¹
- Distribués quand un thread est libre

(GUIDED, min_batch_s)

- Paquets de taille exponentiellement décroissante
- Taille minimale donnée par min_batch_s¹
- Distribués quand un thread est libre

¹sauf le dernier paquet (plus petit) si le nombre d'itérations n'est pas divisible par le nombre de threads

3 - Partage du travail - Boucle parallèle

La répartition la plus simple est le mode *STATIC*.

```
!$OMP DO SCHEDULE(STATIC, batch_size)
```

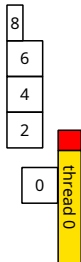
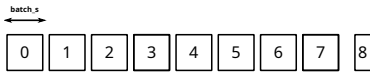
- La taille des paquets est fixée
- Si *batch_size* est omis il est fixé à la taille la plus grande possible
- Les paquets sont distribués de façon cyclique entre les threads

Utile quand :

- Les itérations ont une charge de travail similaire

Commentaires :

- Le compilateur peut faire des optimisations
- Il est préférable de choisir *batch_size* le plus grand possible



3 - Partage du travail - Boucle parallèle

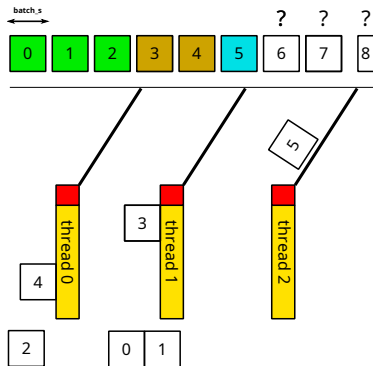
On peut libérer la contrainte de la distribution fixe avec *DYNAMIC*.

```
!$OMP DO SCHEDULE(DYNAMIC, batch_size)
```

- La taille des paquets est fixée
- Si *batch_size* est omis il est fixé à 1 !
- Le paquet suivant est distribué à un thread libre

Utile quand :

- Les ressources de la machine sont partagées
- La charge de travail varie suivant l'itération



3 - Partage du travail - Boucle parallèle

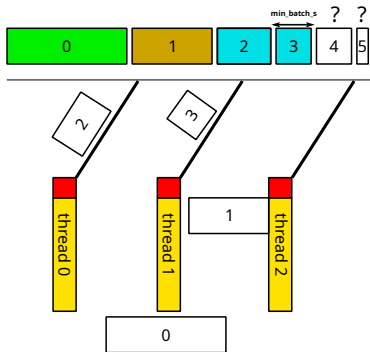
On peut libérer la contrainte de la distribution et de la taille de paquets fixes avec *GUIDED*.

```
!$OMP DO SCHEDULE(GUIDED, min_batch_size)
```

- La taille des paquets est proportionnelle aux nombre d'itérations restantes divisées par le nombre de threads
- *min_batch_size* fixe la taille de paquet minimale
- Si *batch_size* est omis il est fixé à 1 !
- Le paquet suivant est distribué à un thread libre

Utile quand :

- La charge de travail croit avec les itérations
- Par exemple : Matrice triangulaire supérieure



Il existe deux autres options pour la clause **SCHEDULE**:

- *AUTO* : Délègue le choix au compilateur et au runtime
- *RUNTIME* : Le mode de répartition est choisi grâce à la variable d'environnement

OMP_SCHEDULE.

```
#export OMP_SCHEDULE='STATIC,500'  
#export OMP_SCHEDULE='DYNAMIC,200'  
export OMP_SCHEDULE='GUIDED,4'  
OMP_NUM_THREADS=4 a./out
```

Quelques conseils :

- Le choix du mode de répartition et du *batch_size* peut avoir une grande influence sur les performances.
- Utiliser l'option *RUNTIME* permet de ne pas recompiler le code pendant les tests.

3.2.2 Exécution ordonnée : Directive et Clause **ORDERED**

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

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

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

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

3.2.3 Clause **REDUCTION**

- Une réduction est une opération associative appliquée à une variable partagée.
- L'opération peut être :
 - arithmétique : +, -, × ;
 - 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 Fusion d'un nid de boucles : Clause **COLLAPSE**

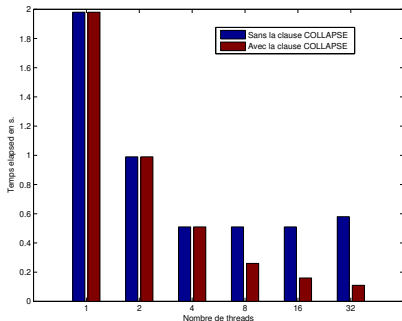
- Dans le cas de boucles parfaitement imbriquées et sans dépendances, il peut être intéressant de les fusionner pour obtenir un espace d'itération plus grand.
- Ainsi, on augmente la granularité de travail de chacun des threads ce qui peut parfois améliorer significativement les performances.
- La clause **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
```

3 - Partage du travail - Boucle parallèle

- 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.



```
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
```


3 - Partage du travail - Boucle parallèle

- 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
```

```
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
```

3.2.5 Compléments

- 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 - À vous de jouer

Un petit mot sur la mesure de performance à la page 128
Liste des TP que vous pouvez faire :

- TP 1 : Produit de matrices
- TP 2 : Méthode de Jacobi
- TP 3 : Calcul de π (1 seule façon pour le moment)

3.4 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**).

3 - Partage du travail - 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
```

3 - Partage du travail - WORKSHARE

- 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 - À vous de jouer

Liste des TP que vous pouvez faire :

- TP4 : Gradient conjugué

3.6 Sections parallèles

3.6.1 Construction **SECTIONS**

- 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 - Partage du travail - Sections parallèles

```
program parallele
  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 parallele
```

```
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
```

3.6.2 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 déjà :
 - **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**.

3.7 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.

3.7.1 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**.

```
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
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 - Partage du travail - Exécution exclusive

- 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**.

```
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
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.
```

3.7.2 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
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.
```

3.8 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
```

3 - Partage du travail - Procédures orphelines

- 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 **P**arallèle) à la compilation ainsi que les directives de l'unité appelée (le travail peut être **D**istribué) ;
 - la directive **PARALLEL** de l'unité appelante est interprétée à la compilation (l'exécution peut être **P**arallèle) mais pas les directives contenues dans l'unité appelée (le travail peut être **R**épliqué) ;
 - la directive **PARALLEL** de l'unité appelante n'est pas interprétée à la compilation. L'exécution est partout **S**équentielle même si les directives contenues dans l'unité appelée ont été interprétées à la compilation.

Prog. principal compilé	Bibliothèque compilée	
	avec OpenMP	sans OpenMP
avec OpenMP	P + D	P + R
sans OpenMP	S	S

Table: Contexte d'exécution selon le mode de compilation

3 - Partage du travail - Récapitulatif

3.9 Récapitulatif

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

4 - Synchronisations

La synchronisation devient nécessaire dans les situations suivantes :

- pour s'assurer que tous les threads concurrents aient atteint un même niveau d'instruction dans le programme (barrière globale) ;
- 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).
- pour synchroniser au moins deux threads concurrents parmi tous les autres (mécanisme de verrou).

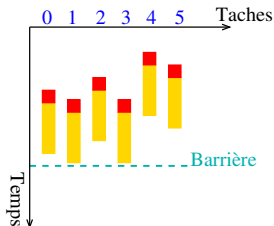
4 - Synchronisations

- Comme nous l'avons déjà 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 - Synchronisations - Barrière

4.1 Barrière explicite : **BARRIER**

- 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 - Synchronisations - Mise à jour atomique

4.2 Mise à jour atomique : **ATOMIC**

- 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
```

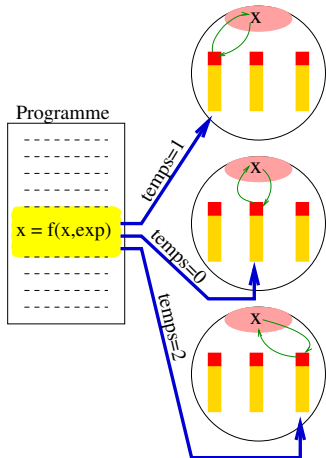
```
Rang : 1 ; compteur vaut : 92291
Rang : 0 ; compteur vaut : 92292
Rang : 2 ; compteur vaut : 92293
Rang : 3 ; compteur vaut : 92294
Au total, compteur vaut : 92294
```

4 - Synchronisations - Mise à jour atomique

- L'instruction en question doit avoir l'une des formes suivantes :

- $x = x \text{ (op) exp}$;
- $x = \text{exp (op) } x$;
- $x = f(x, \text{exp})$;
- $x = f(\text{exp}, x)$.

- (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 : **CRITICAL**

- 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.

4 - Synchronisations - Régions critiques

```
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
OMP_NUM_THREADS=4 ./a.out
```

```
s = 8 ; p = 16
```


4 - À vous de jouer

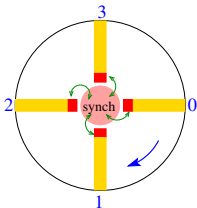
Liste des TP que vous pouvez faire :

- TP 3 : Calcul de π (les 2 autres façons)
- TP 7 : Gradient biconjugué
- TP 8 : Méthode de Poisson

4 - Synchronisations - FLUSH

4.5 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 - Synchronisations - FLUSH

4.5.1 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 - Synchronisations - FLUSH

4.5.2 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 - Synchronisations - FLUSH

4.5.3 Commentaires sur les exemples 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 - Synchronisations - FLUSH

4.5.4 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 - Synchronisations - FLUSH

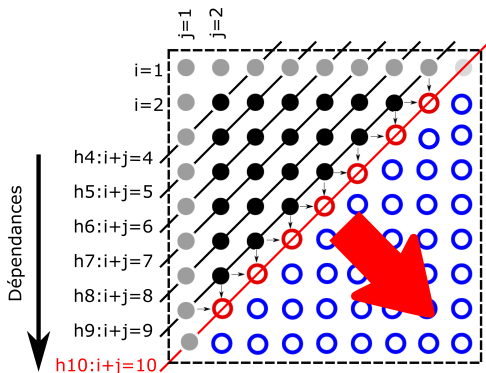
- Considérons le code suivant :

```
! Boucles avec double dépendance
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
```

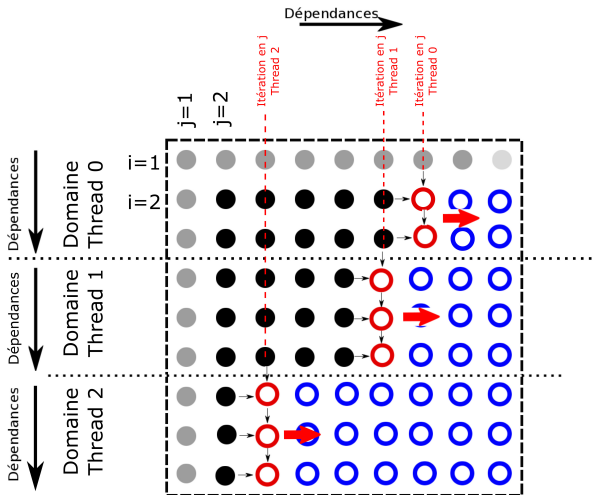
- Une fois le code réécrit, la parallélisation est très simple et ne nécessite pas d'avoir recours à des synchronisations fines.
- Les performances obtenues ne sont hélas pas optimales (médiocre utilisation des caches due aux accès en diagonale, donc non contigu en mémoire).

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.

4 - Synchronisations - FLUSH

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
  ! précédent 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 - Synchronisations - Récapitulatif

4.6 Récapitulatif

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

4 - À vous de jouer

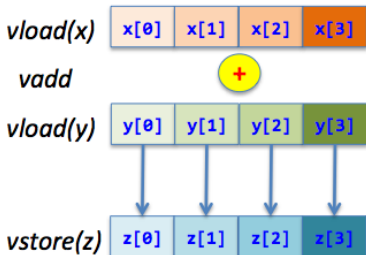
Liste des TP que vous pouvez faire :

- TP 9 : Nid de boucles avec dépendances (Avec **FLUSH**)

5 Vectorisation SIMD

- 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.

```
for (i = 0; i < n; i++)  
    z[i] = x[i] + y[i];
```



5.1 Vectorisation 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'entraîne 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
```


5.2 Parallélisation ET vectorisation 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
```

5.3 Vectorisation de fonction scalaire

- 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
```

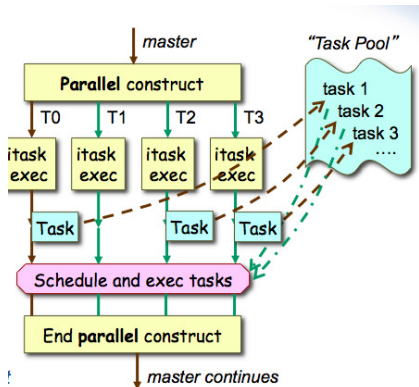
6 - Les tâches OpenMP - Introduction

- 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 nouveau 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).

6.1 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.

6.2 Modèle d'exécution



---> may be deferred

←... scheduling

- ♦ *implicit tasks cannot be deferred*
- ♦ *explicit tasks could be deferred*

6 - Les tâches OpenMP - 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.

6.3 Quelques exemples

```
program task_print
implicit none

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

end program task_print
```

```
ifort ... -fopenmp task_print.f90
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
OMP_NUM_THREADS=2 ./a.out
```

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

6 - Les tâches OpenMP - Quelques exemples

```
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
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" ?

6 - Les tâches OpenMP - Quelques exemples

- Si on rajoute un *print* juste avant la fin de la région **SINGLE**, ça 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
```

6 - Les tâches OpenMP - Quelques exemples

- 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
```

6.4 Dépendances 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.

6 - Les tâches OpenMP - Dépendance entre tâches

- 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
OMP_NUM_THREADS=2 ./a.out
```

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

6.5 Statut des variables

- 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.

6.6 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.

6.7 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
```

6.8 Clauses **MERGEABLE** et **FINAL**

- 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 avoir recours à un "*cut off*" manuel dans le code...

6.9 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 :: arbre
! 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
```

6 - Les tâches OpenMP - TASKGROUP

```
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
```

6 - À vous de jouer

Liste des TP que vous pouvez faire :

- TP 9 : Nid de boucles avec dépendances (avec les tâches)
- TP 10 : Algorithme de Strassen

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
...
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)
```

7 - Affinités - Affinité des threads

```
$ cpufreqinfo <= 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 cœurs physiques du noeud
Processors(CPUs)   : 64    <= Nb de cœurs logiques du noeud
Cores per package  : 8     <= Nb de cœurs physiques par socket
Threads per core   : 2     <= Nb de cœurs 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
...
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
...
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)
```

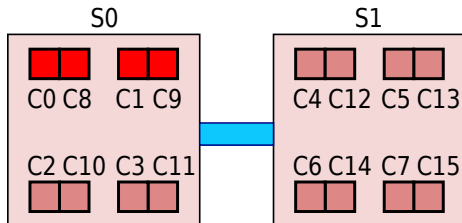
7.1.2 Utilisation de `KMP_AFFINITY`

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).

```
export KMP_AFFINITY=granularity=thread,  
compact,verbose
```

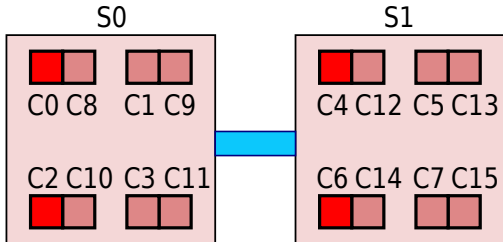


Exemple sur une architecture bi-sockets quadri-cœurs, avec l'hyperthreading activé.

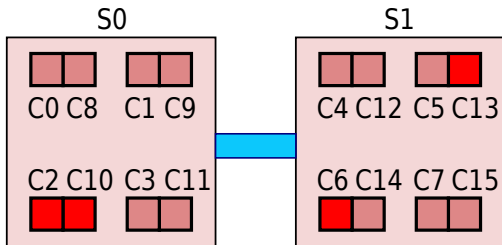
7 - Affinités - Affinité des threads

```
export KMP_AFFINITY=granularity=thread,scatter,verbose
```

- 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.



```
export KMP_AFFINITY=proclist=[2,10,13,6],explicit,verbose
```



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.

7 - Affinités - Affinité des threads

- 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.

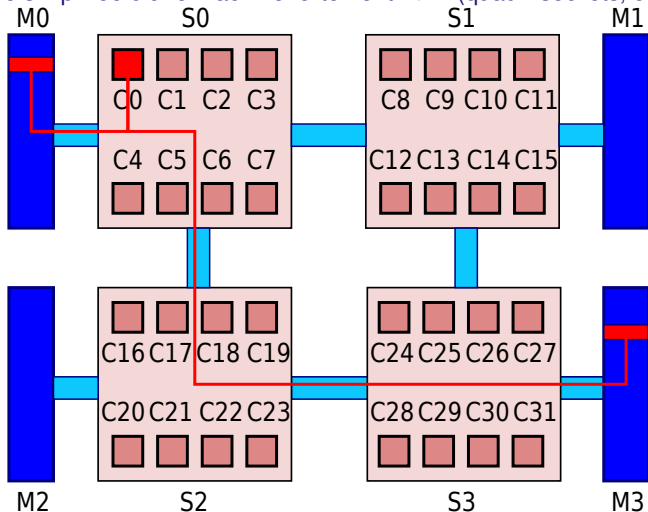
```
export OMP_PLACES="{0,8,1,9},{2,10,3,11},{4,12,5,13},{6,14,7,15}"
Soit 4 places p0={0,8,1,9}, p1={2,10,3,11}, p2={4,12,5,13} et p3={6,14,7,15}
!$OMP PARALLEL PROC_BIND(SPREAD) NUM_THREADS(2)
    !$OMP PARALLEL PROC_BIND(CLOSE) NUM_THREADS(4)
    ....
Dans la premiere region parallele
Th0 s'executera 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 influencer 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*.

7 - Affinités - Affinité mémoire

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



7 - Affinités - Affinité mémoire

- 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
```

```
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
```

```
OMP_NUM_THREADS=32 ./a.out
```

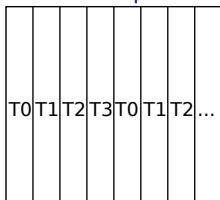
```
Temps elapsed = 10.22 s.
```

- L'utilisation de la stratégie de type “*First Touch*” permet un gain de l'ordre d'un facteur 10 sur cet exemple !

7 - Affinités - Exemples d'impact sur les performances

- 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



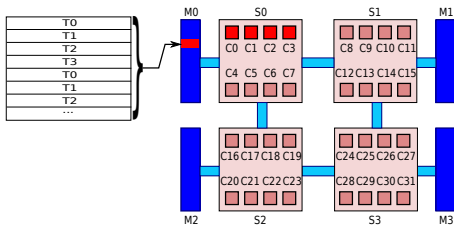
Itération impaire

T0
T1
T2
T3
T0
T1
T2
...

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

7 - Affinités - Exemples d'impact sur les performances

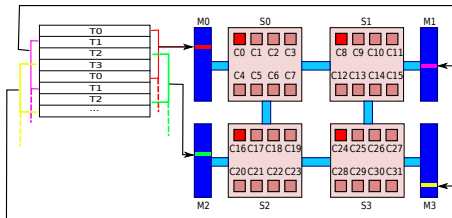
• Binding de type *compact*



```
OMP_NUM_THREADS=4 ./a.out
```

Temps elapsed = 33.46 s.

• Binding de type *scatter*



```
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* !

7 - À vous de jouer

Liste des TP que vous pouvez faire :

- TP 5 : Réduction de tableau
- TP 6 : FFT multiple

Objectifs

- Réduire le temps pour obtenir une solution
- Conserver des résultats corrects !

Performances

- Dépendantes de l'architecture
 - processeurs
 - mémoire
 - liens d'interconnexion

8.1 Règles de bonnes performances

Pendant le développement

- Vérifier la qualité de l'implémentation du compilateur
- Minimiser le nombre de régions parallèles dans le code
- Dans la mesure du possible, paralléliser la boucle la plus externe
- Éviter de paralléliser la boucle des éléments contigus en mémoire (pour les nids de boucles)
- Utiliser la clause **SCHEDULE(RUNTIME)** pour tester plusieurs configurations
- Utiliser la directive **SINGLE** et la clause **NOWAIT** (*Synchronisations !*)
- Coût : **REDUCTION** < **ATOMIC** < **CRITICAL**
- Si la machine est NUMA : Utiliser le "First Touch"

À l'exécution

- Activer le mécanisme de binding des threads sur les coeurs
- Adapter le nombre de threads à la taille du problème → minimisation du surcoût de gestion

8 - Performances - Règles de bonnes performances

- Utiliser la clause **IF** pour mettre en place une parallélisation conditionnelle

```
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
```

8.2 Mesures du temps

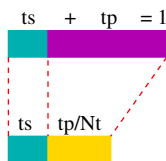
- **OpenMP** offre deux fonctions :
 - **OMP_GET_WTIME** pour mesurer le temps de restitution en secondes ;
 - **OMP_GET_WTICK** pour connaître la précision des mesures en secondes.
- Ce que l'on mesure est le temps écoulé depuis un point de référence arbitraire dans le code.
- Cette mesure peut varier 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
```

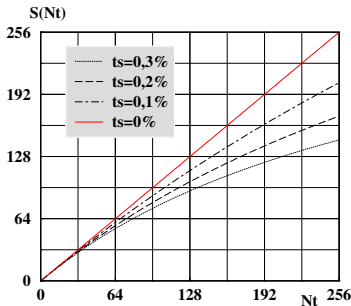
8 - Performances - Accélération

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.



$$S(N_t) = \frac{1}{t_s + \frac{t_p}{N_t}}$$



9 - Conclusion

Nécessite

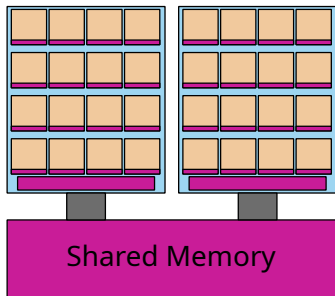
- Machine multicoeur
- Mémoire partagée

Programmation

- Modifications simples avec des directives
- Portage incrémental
- Création de bibliothèques (procédures orphelines)
- *Surcouts liés à la gestion des threads !*

Régions parallèles

- Coeur des gains de performances
- Partage du travail : **do, sections, task, ...**
- Synchronisations : calculer vite et juste !
barrier, atomic, critical, reduction, ...
- Exécutions exclusives : évite de fermer une région parallèle. **master, single**

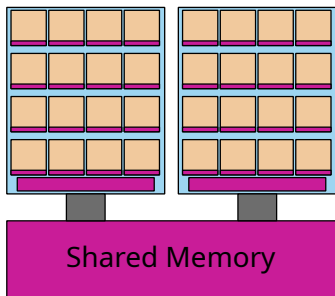


```
program
!$omp parallel
    !$omp do
        . . .
    !$omp end parallel
```

Pièges courants

- Problème de statut des variables : **private**, **firstprivate**, **shared** → Race condition, résultats faux
- Désynchronisation des threads : blocage du code
- Support des directives par les compilateurs

Cours Débogage HPC de l'IDRIS



```
program
!$omp parallel
    !$omp do
        .
        .
        .
    !$omp end parallel
```

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 - Annexes - Quelques pièges

10.2 Quelques pièges

```
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
```

Quel est le souci ?

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

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

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> OMP_NUM_THREADS=3 ./a.out
s=92290. ; a( 9 )=92290.
s=0.    ; a( 9 )=92290.
s=0.    ; a( 9 )=92290.

```

```

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

```

10 - Annexes - Quelques pièges

Quel est le souci ?

```
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> OMP_NUM_THREADS=3 ./a.out
s = 1.0
s = 2.0
s = 2.0
```

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

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

```
IBM SP> OMP_NUM_THREADS=3 ./a.out
s = 1.0
s = 2.0
s = 2.0
```

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

Quel est le souci ?

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


10 - Annexes - Quelques pièges

```
!$OMP PARALLEL          Iteration = 0
do 1
  !$OMP SINGLE
  iteration = iteration + 1
  !$OMP END SINGLE
  if( iteration >= 3 ) exit
end do
!$OMP END PARALLEL
```

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

10 - Annexes - Quelques pièges

```
!$OMP PARALLEL          Iteration = 1
do 1
  !$OMP SINGLE
  ● iteration = iteration + 1
  !$OMP END SINGLE ●
  if( iteration >= 3 ) exit
end do
!$OMP END PARALLEL
```

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

10 - Annexes - Quelques pièges

```
!$OMP PARALLEL           Iteration = 1  
do 1  
  !$OMP SINGLE  
  iteration = iteration + 1  
  !$OMP END SINGLE ●●  
  if( iteration >= 3 ) exit  
end do  
!$OMP END PARALLEL
```

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

10 - Annexes - Quelques pièges

```
!$OMP PARALLEL          Iteration = 2
do 2
  !$OMP SINGLE
  iteration = iteration + 1
  !$OMP END SINGLE
  if( iteration >= 3 ) exit
end do
!$OMP END PARALLEL
```

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

10 - Annexes - Quelques pièges

```
!$OMP PARALLEL          Iteration = 2
do 2
  !$OMP SINGLE
  iteration = iteration + 1
  !$OMP END SINGLE
  if( iteration >= 3 ) exit
end do
!$OMP END PARALLEL
```

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

10 - Annexes - Quelques pièges

```
!$OMP PARALLEL          Iteration = 3
do 3
  !$OMP SINGLE
  iteration = iteration + 1
  !$OMP END SINGLE
  if( iteration >= 3 ) exit
end do
!$OMP END PARALLEL
```

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

10 - Annexes - Quelques pièges

```
!$OMP PARALLEL          Iteration = 3
do 3
  !$OMP SINGLE
  iteration = iteration + 1
  !$OMP END SINGLE
  if( iteration >= 3 ) exit
end do
!$OMP END PARALLEL
```

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

10 - Annexes - Quelques pièges

```
!$OMP PARALLEL           Iteration = 3
do 3
  !$OMP SINGLE
  iteration = iteration + 1
  !$OMP END SINGLE
  if( iteration >= 3 ) exit
end do
!$OMP END PARALLEL
```

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


10 - Annexes - Quelques pièges

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