Open-MP

Françoise BERTHOUD

LPMMC (Méso-centre CIMENT, Grenoble)

Ecole d'Automne Informatique Scientifique

Décembre 2008

(Ce cours est construit à partir du cours proposé
dans le cadre de CIMENT – auteurs : Françoise ROCH/Françoise BERTHOUD)

Modèles à mémoire partagée

- Plusieurs processus s'exécutent en parallèle
- Les processus peuvent être attribués à des « cœurs » distincts
- La mémoire est partagée (physiquement ou virtuellement)
- **Les communications entre processeurs se font par lectures et écritures dans la mémoire partagée.**

➔ adaptés aux architectures à mémoire partagée :

•Multiprocesseurs

- •SMP (type IBM power VI)
- •CC-NUMA (type SGI altix 450)

•Multicoeurs : xeon bi/quadri core

Caractéristiques du modèle

- Facilité de programmation
- Gestion des synchronisations doit se faire explicitement (pas par des messages)

Mais

- Problème de localité des données (cache, cc-numa)
- Efficacité non garantie (impact de l'organisation matérielle de la machine)

Open_MP

- Introduction
- Structure d’Open_MP
- Visibilité des données
- Partage du travail
- Synchronisation
- Conclusion

Supports d'Open_MP

- Constructeurs hardware
  - Intel, HP (+Compaq), SGI, IBM, SUN
- Vendeurs logiciels
  - Intel,KAI, PGI, PSR, APR, Absofts, Lahey, gaussian, lib scsl (sgi), sun

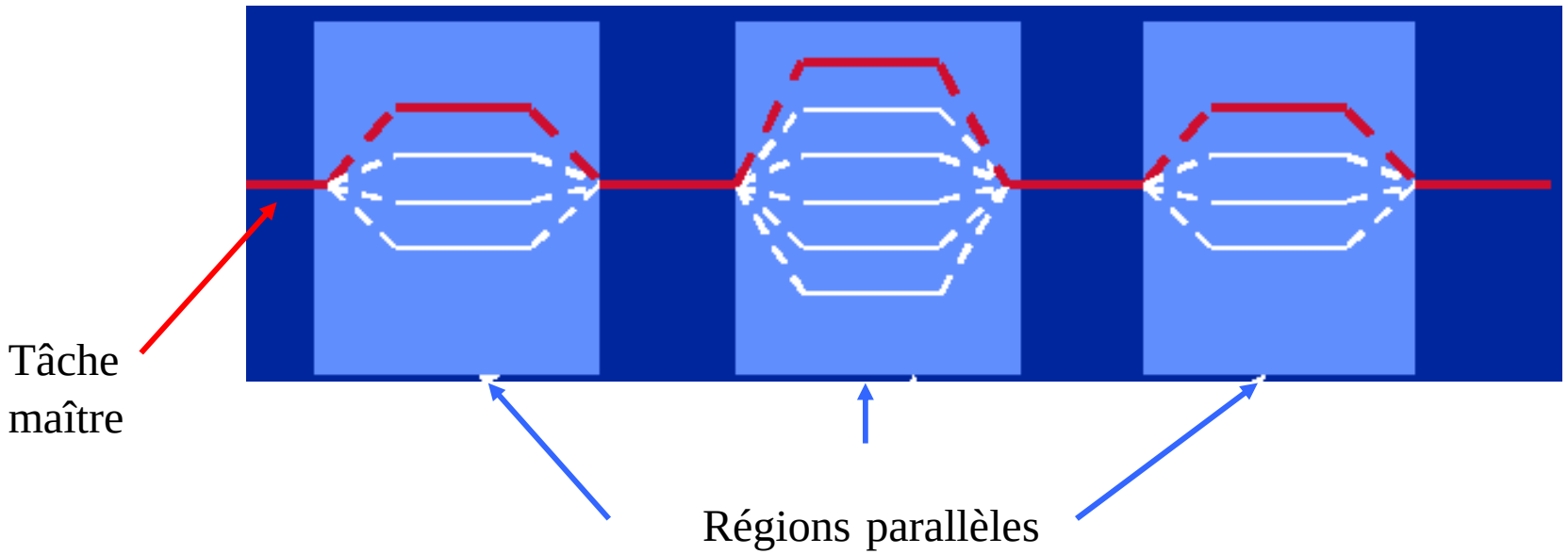
Remarque :

Différents niveaux de support : V1, V2, Fortran, C, C++

Modèle d'exécution

Un programme Open_MP est exécuté par un processus unique (sur un ou plusieurs processeurs)

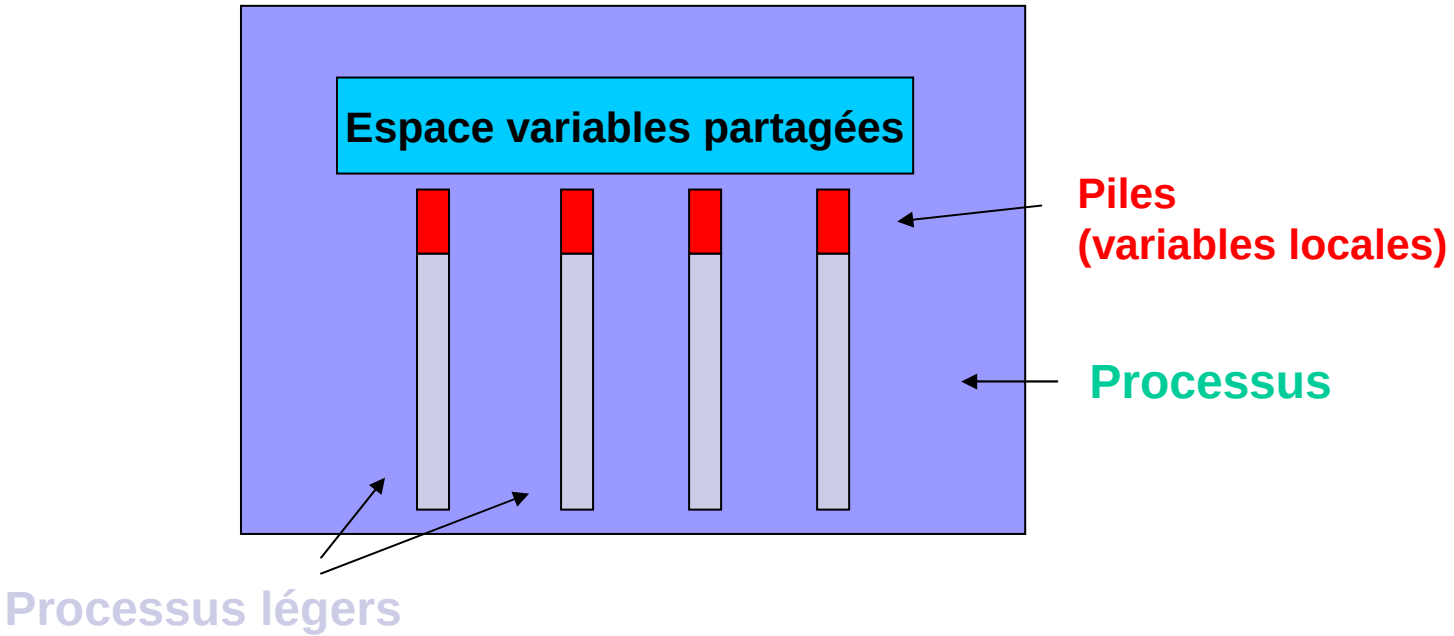
- Région parallèle



Processus légers (threads)

Les threads accèdent aux mêmes ressources que le processus.

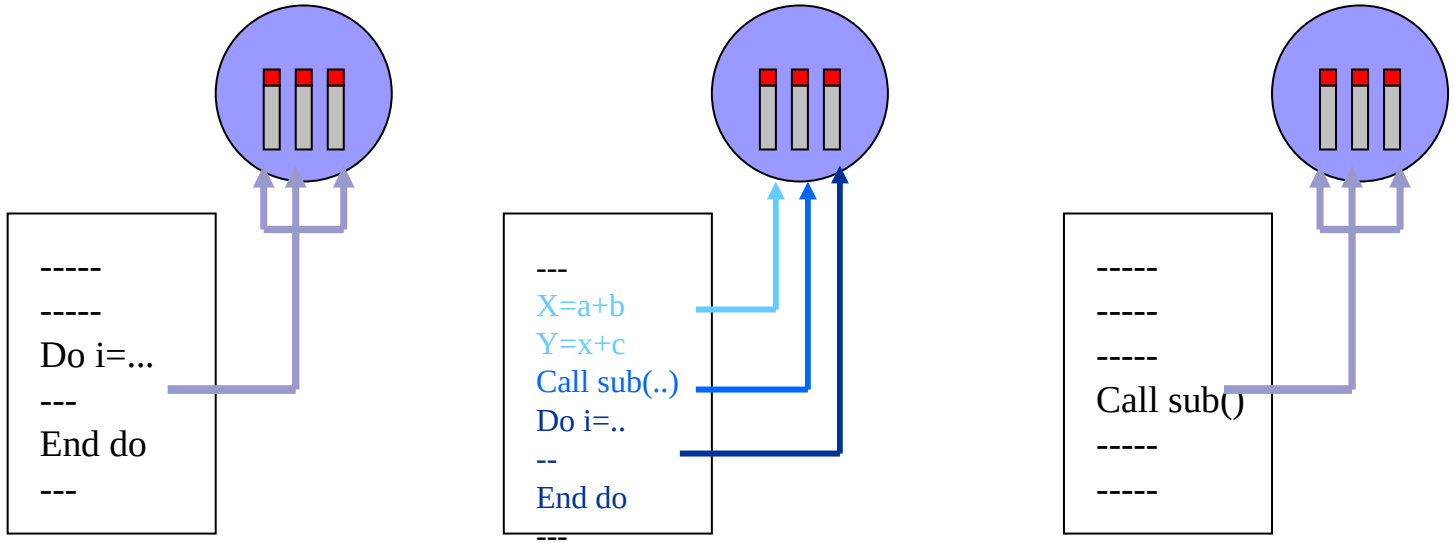
Elles ont une pile (stack, pointeur de pile et pointeur d'instructions propres)



PARTAGE DES DONNEES

Partage du travail

- Exécuter une boucle par répartition entre les tâches (boucle //)
- Exécuter plusieurs sections, une section de code par tâche (sections //)
- Exécuter plusieurs occurences d'une même procédure par différentes tâches (orphaning)

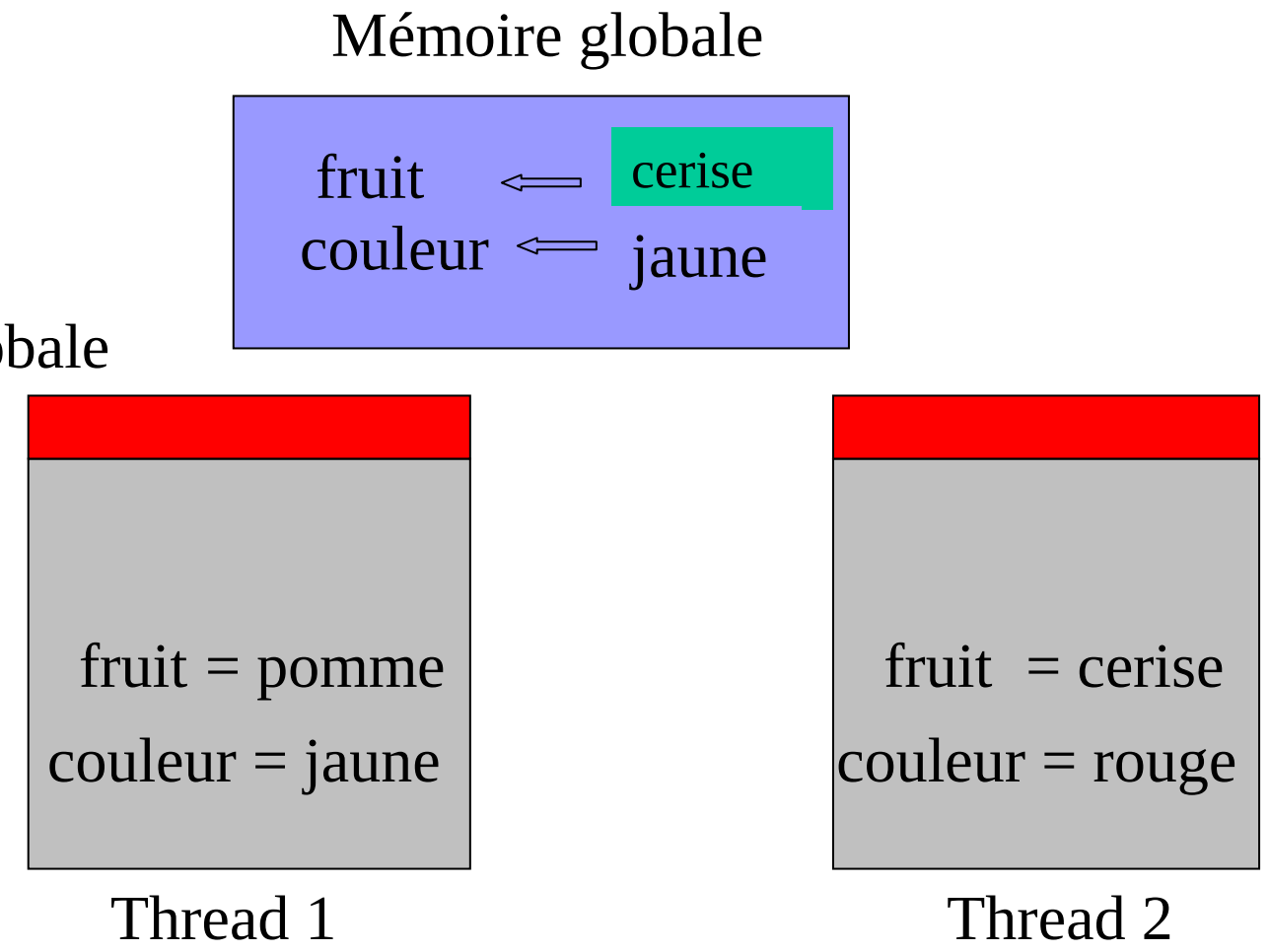


Synchronisation

Il peut être nécessaire d'introduire une synchronisation entre tâches concurrentes

Ex :

2 threads se partagent la mémoire globale

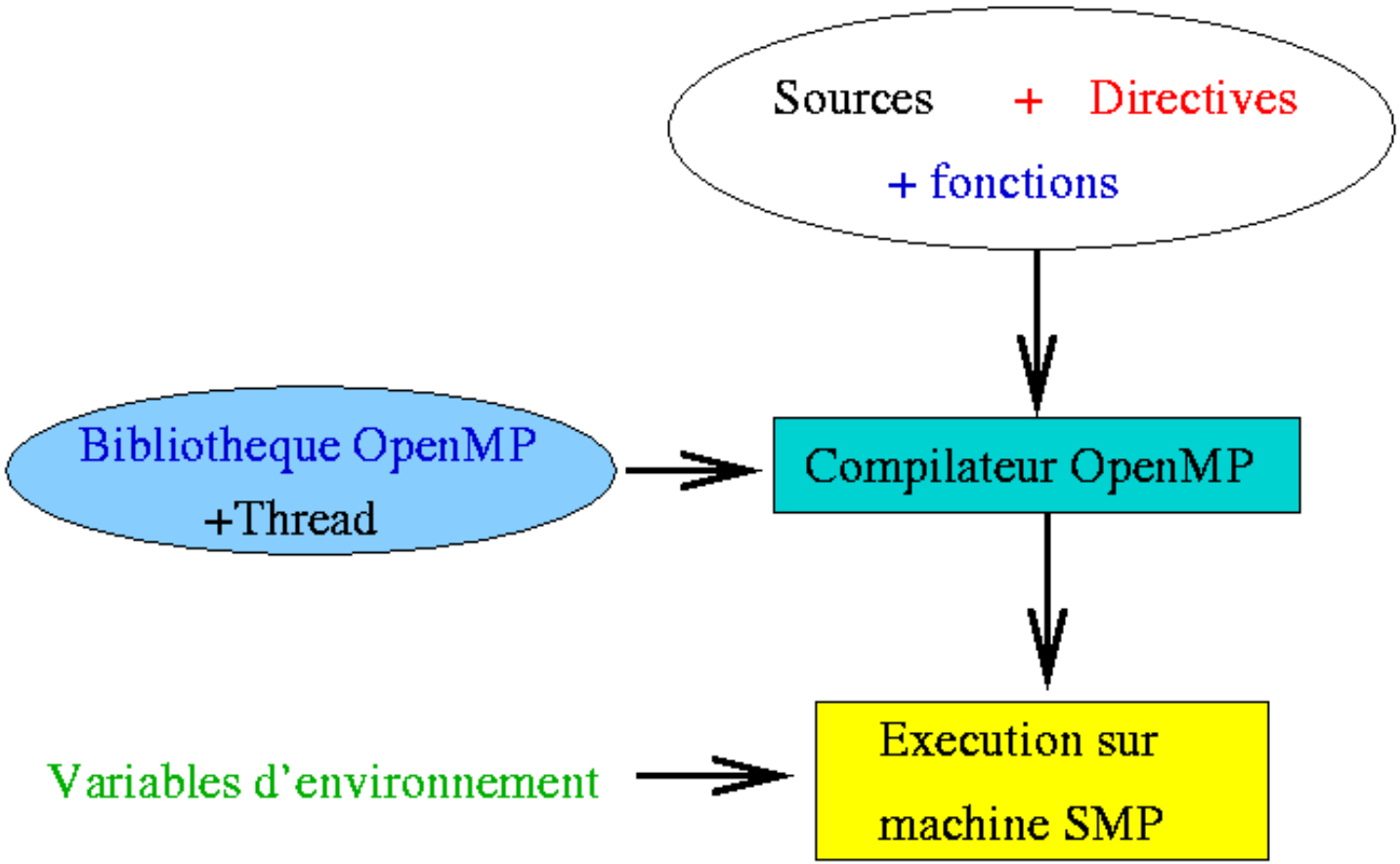


Open-MP

- Introduction
  - **Structure d'OPEN_MP**
- Visibilité des données
- Partage du travail
- Synchronisation
- Conclusion

Structure d'OpenMP

- Architecture logicielle



Principes

- Une directive OpenMP possède la forme générale suivante (ligne interprétée si option openmp à l'appel du compilateur – préprocesseur sinon vue comme un commentaire ➔ portable) :

Sentinelle directive [clause[clause]..]

Ex (fortran)

```
!$OMP PARALLEL PRIVATE(a,b) &
        !$OMP FIRSTPRIVATE(c,d,e)
...
!$OMP END PARALLEL
```

Ex (fortran fixe)

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

Ex (C ou C++)

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

Threads OpenMp

- Le nombre de threads peut être défini :

  Soit via la variable d'environnement OMP_NUM_THREADS

  Soit via la routine : OMP_SET_NUM_THREADS()

  Soit via la clause (directive PARALLEL) OMP_THREADS()[2]

- Les threads sont numérotées

  Attention : le nombre de threads n'est pas nécessairement égale au nombre de processeurs physiques

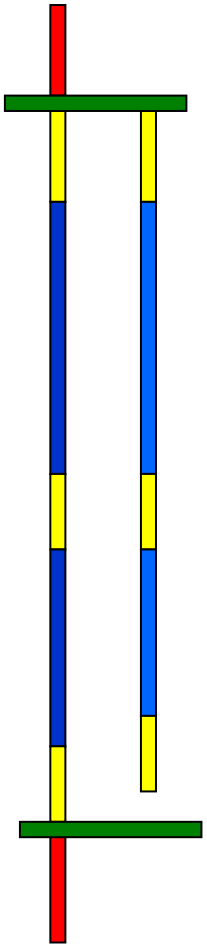
  La thread de numéro 0 est la tâche maître

  OMP_GET_NUM_THREADS() : nombre de threads

  OMP_GET_THREAD_NUM() : numéro de la thread

  OMP_GET_MAX_THREADS() : nb max de threads

Exemple : cas de 2 threads



```
Program main
…
!$omp parallel
….
….
!$omp sections
!$omp section
…
!$omp section
…

!$omp end sections
…
!$omp do
  do
  …
  end do

!$omp end do nowait
…
!$omp end parallel
…
end
```

```
! Début de l'exécution séquentielle
…
! Début de la construction parallèle
! Cette portion du code est répliquée, chaque thread exécute le même
      code
! Début d'une construction de travail partagé
! Début d'une unité de travail

! Début d'une autre unité de travail

! Fin de la construction de travail partagé (attente)
! Code répliqué
! Début d'une construction de travail partagé
! Chaque itération est une unité de travail,
! le travail est distribué sur toutes les threads

! Fin de la construction sans attendre que chacune des threads
! ait terminé l'exécution de sa portion de code
! Fin de la construction parallèle
! Exécution en série
```

Compilation

xlf_r –qsmp=omp -qfree=f90 prog.f (sous aix)
ifc (ou icc ou icpc) –openmp prog.f (compilateur intel IA32)
ifort (ou ecc ou ecpc) –openmp prog.f (compilateur intel)
f90 (ou cc ou CC) –openmp prog.f (compilateur sun studio10)
export OMP_NUM_THREADS=2 (ou setenv OMP_NUM_THREADS 2)
./a.out

```
# ps –H  (top –H)
USER   PID    PPID    TID   ST  CP PRI SC    WCHAN       F       TT BND COMMAND
roch   26910 27274         - A  240 120  2      e600acbc   200001  pts/3   -    a.out
    -          -       -  41465 S  120 120  1                 -  4400010      -   -         -
    -          -       -  44111 S  120 120  1       e600acbc   8400400      -   -         -
```

Ici les threads de numéros 41465 et 44111 sont des sous-processus du processus de numéro 26910.

Clauses de la directive PARALLEL

- La création d'une région parallèle peut être conditionnelle grâce à la clause **IF(expression_logique)** (ex $OMP PARALLELE IF())

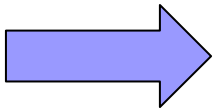
  Cette expression logique sera évaluée avant le début de la région parallèle.

- Le nombre de tâches de cette région parallèle peut être fixé grâce à la clause **NUM_THREADS(N) où (N > 0)** [2]

  - si l'argument de cette clause est une expression scalaire entière, elle sera évaluée avant le début de la région parallèle.)
  - cette clause a précédence sur OMP_SET_NUM_THREADS(N) qui a lui même précédence sur la variable d'environnement OMP_NUM_THREADS)

Open-MP

- Introduction
- Structure d'OPEN_MP



Visibilité des données

- Partage du travail
- Synchronisation
- Conclusion

Rappel : différents types de variables d'un point de vue allocation en mémoire

Définitions : Variables statiques et automatiques

variable statique : emplacement en mémoire défini dès sa déclaration par le compilateur

variable automatique : emplacement mémoire attribué au lancement de l'unité de programme où elle est déclarée et désallouée à la fin de l'exécution de celle-ci

- **Variables globales**

  variable globale : déclarée au début du programme principal, on la considère donc statique, 2 cas :
  - celles initialisées à la déclaration (exemple : *data*)
  - celles non-initialisées à la déclaration (exemple : *common*, *en C : variables d'unités de fichier, static)*

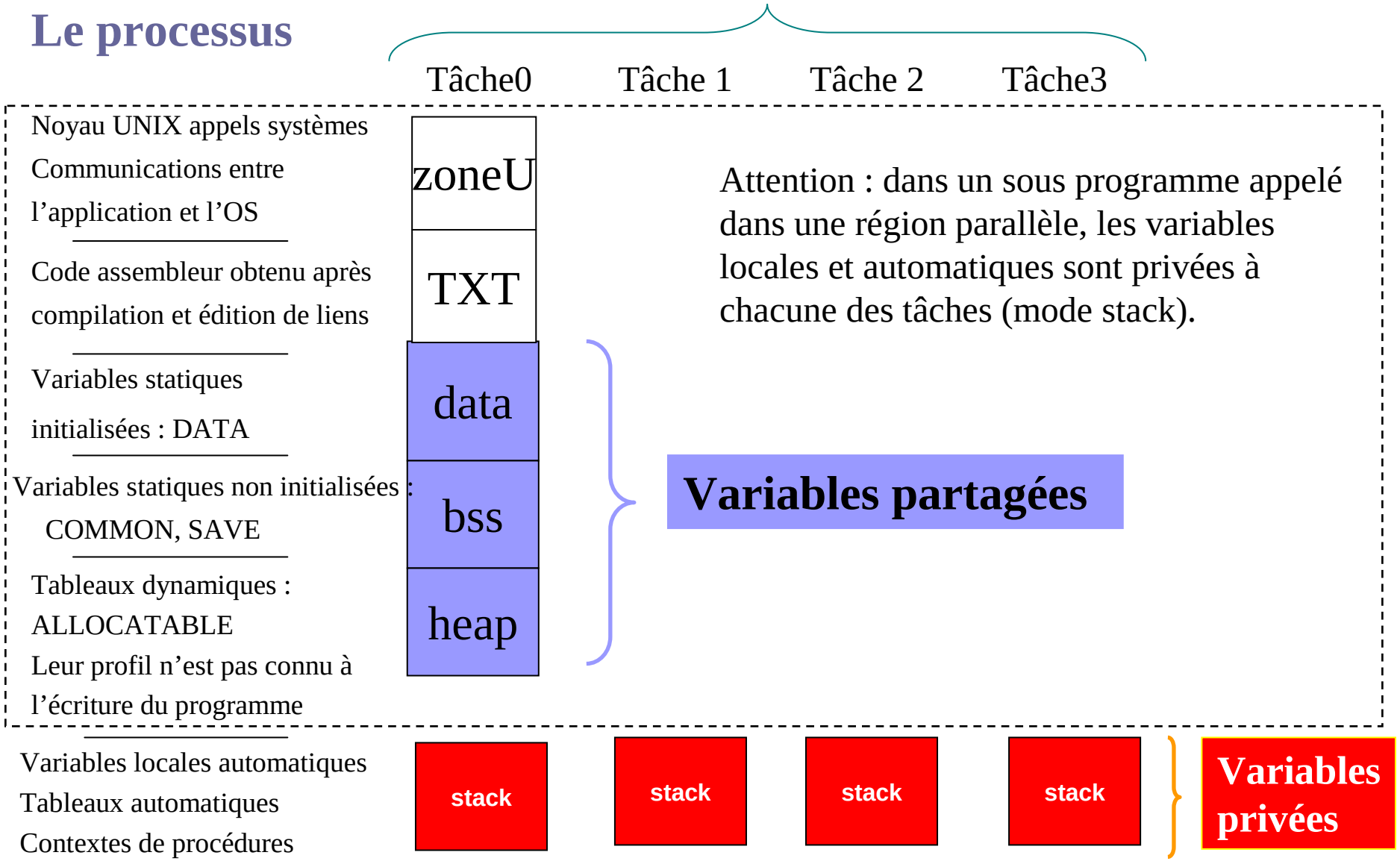
- **Variables locales**

  variable à portée restreinte à l'unité de programme où elle est déclarée, 2 catégories :
  - **Variables locales automatiques (ex: a, local1, local2)**
  - **Variables locales rémanentes (statiques) si elles sont :**
    - a) initialisées explicitement à la déclaration (ex m,n),
    - b) déclarées par une instruction de type DATA,
    - c) déclarées avec l'attribut SAVE.

```
Callsub()
subroutine sub()
integer, parameter :: m=400, n=500
real :: a
real, dimension(m,n) ::local1
integer, dimension(100,300) ::
      local2
end subroutine sub
```

Rappel : stockage des variables

Le processus



Taille de la pile

- La pile (stack) a une taille limite pour le shell (variable selon les machines). (ulimit –s)(ulimit –s unlimited), valeurs exprimées en 1024-bytes.

- Des variables d'env permettent de définir la taille de la stack privée des threads (par exemple KMP_STACKSIZE (linux altix))

Portée d'une variable

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

```
program ex1_var_globale.f
                implicit none
                integer tmp
                integer, dimension(4)::tmp1
                integer OMP_GET_THREAD_NUM
                tmp=999
                print *, "debut de la zone parallele«
                !$omp  parallel
                tmp1(OMP_GET_THREAD_NUM())=tmp
                tmp= OMP_GET_THREAD_NUM()
                print *,
tmp1(OMP_GET_THREAD_NUM()),
                                tmp,
OMP_GET_THREAD_NUM()
                !$omp  end parallel
                end
```

Ex !$OMP PARALLEL DEFAULT(PRIVATE) SHARED (X)

Clauses de la directive PARALLEL (2)

- NONE

  Aucun statut défini par défaut par le programmeur (attention aux statuts définis par défaut par le compilateur)
- SHARED (variables)

  Variables partagées entre les threads
- PRIVATE (variables)

  Variables privées à chacune des threads, indéfinies en dehors du bloc PARALLEL
- FIRSTPRIVATE (variables)

  Variable initialisée avec la valeur de la variable d'origine avant l'entrée dans la section parallèle
- DEFAULT (PRIVATE | SHARED)

ex1_var_globale.f, ex1_var_private.f, ex1_var_private_2.f

Cas de la transmission par arguments

- Dans une procédure, toutes les variables transmises par argument héritent du statut défini dans l'étendue lexicale de la région (code contenu entre les directives PARALLEL).

```
Program ex2
  implicit none
  integer :: a,b
  a=100
!$OMP PARALLEL PRIVATE(b)
      call sub(a,b)
      print *,b
!$OMP END PARALLEL
End program ex2

Subroutine sub(x,y)
integer x,y,OMP_GET_THREAD_NUM
  y = x + OMP_GET_THREAD_NUM()
End subroutine sub
```

ex2.f

Cas des variables statiques (common)

La directive THREADPRIVATE :

- Permet de rendre privé
  - un bloc common (les modifications apportées au bloc common par une thread ne sont pas visibles des autres threads). L'instance du common est persistante d'une région parallèle à l'autre (sauf si le mode dynamic est actif).
  - *Un descripteur de fichier ou des variables static (en C)*
- Doit apparaître après chaque déclaration du bloc common
- La clause COPYIN permet de transmettre la valeur des instances statiques à toutes les tâches

```
program threadpriv3
                implicit none
                integer tid,x
                common /bidon/x
                integer OMP_GET_THREAD_NUM
                !$omp THREADPRIVATE(/bidon/)
                call OMP_SET_NUM_THREADS(4)
                !$omp PARALLEL PRIVATE(tid)
                tid = OMP_GET_THREAD_NUM()
                x=tid*10+1
                print *,"T:",tid,"dans la premiere region // x=",x
                !$omp end PARALLEL
                x=2
                print *,"T:",tid,"en dehors de la region //  x=",x
                !$omp PARALLEL PRIVATE(tid)
                tid = OMP_GET_THREAD_NUM()
                print *,"T:",tid,"dans la seconde region // x=",x
                !$omp  end PARALLEL
                end
```

threadpriv1.f threadpriv2.f threadpriv3.f threadpriv4.f

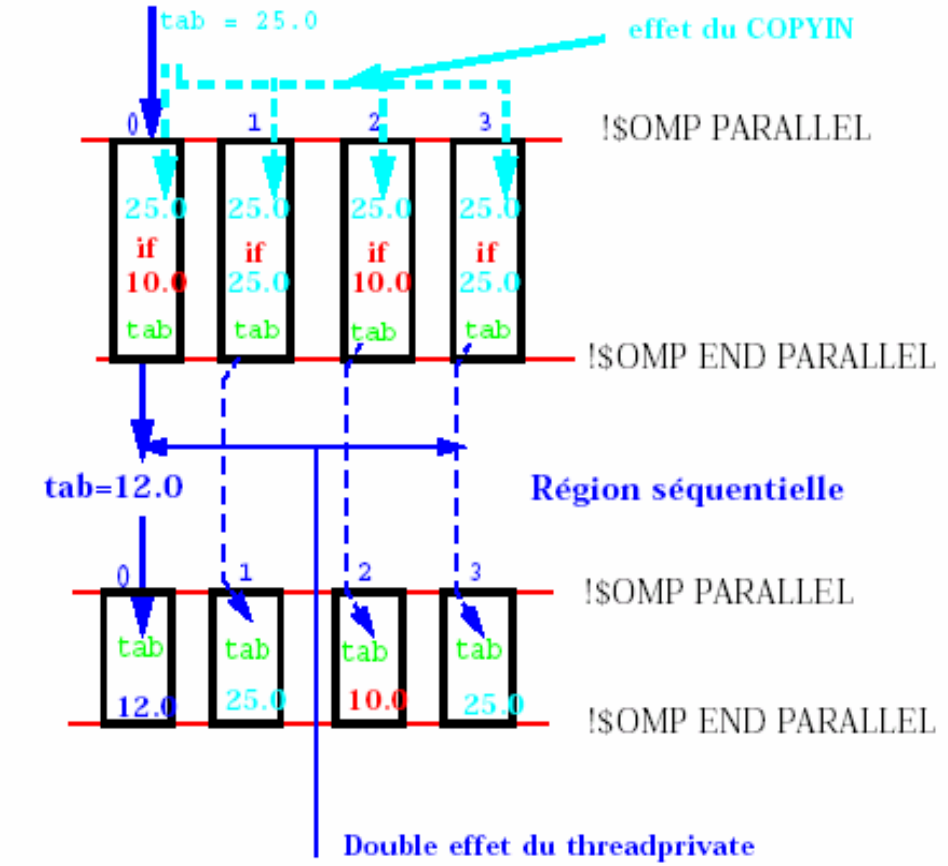
Threadprivate

```
real, dimension(m,n) :: tab,x
integer :: moi, omp_get_thread_num
common /commun1/tab,moi
!$OMP THREADPRIVATE
      (/commun1/)
tab(:,:) = 25.0

!$OMP PARALLEL
      COPYIN(/commun1/)
moi = omp_get_thread_num()
if (mod(moi,2)) == 0) tab(:,:) = 10.0
!$OMP END PARALLEL

tab = 12.0

!$OMP PARALLEL
moi = omp_get_thread_num()
print *, 'PE : ', moi,tab(m,n)
!$OMP END PARALLEL
```



La directive THREADPRIVATE a un double effet :

- Ces variables deviennent **persistantes** d'une région // à l'autre
- L'instance des variables des zones séquentielles est aussi celle de la **tache 0**

Cas particuliers

Attention : sous aix : l'option de compilation -qsave permet de rendre statiques les variables locales : si c'est l'option par défaut lors de l'appel de xlf_r, f77, xlf (cf fichier de configuration /etc/xlf.cfg), il faut alors spécifier l'option –qnosave pour que les variables locales automatiques restent privées.

L'option par défaut des compilateurs intel et sun studio10 est –auto/-stackvar (variables locales allouées dans la stack => privé).

compilateur	ifort ou ifc (intel)	F90 (sun studio10)	xlf (ibm)
Variables locales privées (stack) Cas des tableaux	-auto (défaut) -stack_temps	-Stackvar (défaut)	-qnosave
Variables locales statiques	N/A	N/A	-qsave (défaut)

Cas particuliers : attention

- Si une opération d'allocation/désallocation de mémoire porte sur une variable partagée, il est plus prudent que seule la tâche maître se charge de cette opération.
- Ne mettre en équivalence que des variables de même statut (SHARED ou PRIVATE), même dans le cas d'une association par pointeur.

dynalloc.f

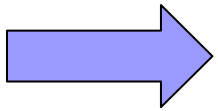
Quelques précisions

- Les variables privatisées dans une région parallèle ne peuvent pas être re-privatisées dans une construction parallèle interne à la région parallèle.
- En fortran les pointeurs et les tableaux « allocatable » peuvent être PRIVATE ou SHARED mais pas LASTPRIVATE ni FIRSTPRIVATE.
- Les tableaux à taille implicite (A(*)) et les tableaux à profil implicite (A(:)) grâce à une interface explicite Ne peuvent être déclarés ni PRIVATE, ni FIRSTPRIVATE ou LASTPRIVATE
- Quand un bloc common est listé dans un bloc PRIVATE, FIRSTPRIVATE ou LASTPRIVATE, les éléments le constituant ne peuvent pas apparaître dans d'autres clauses de portée de variables. Par contre, si un élément d'un bloc common SHARED est privatisé, il n'est plus stocké avec le bloc common
- Un pointeur privé dans une région parallèle sera ou deviendra forcément indéfini à la sortie de la région parallèle

Open-MP

Introduction

- Structure d’OPEN_MP
- Visibilité des données



Partage du travail

- Synchronisation
- Conclusion

Portée d'une région parallèle : définition

- La portée d'une région parallèle s'étend :
  - au code contenu lexicalement dans cette région (étendue statique)
  - au code des sous programmes appelés

  L'union des deux représente l'étendue dynamique

```
Program portee
implicit none
!$OMP PARALLEL
        call sub()
!$OMP END PARALLEL
End program portee
Subroutine sub()
Logical :: p, OMP_IN_PARALLEL
!$ p = OMP_IN_PARALLEL()
print *, "Parallel prog ? ", p
End subroutine sub
```

portee.f

Partage du travail

- Trois directives permettent de contrôler la répartition du travail, des données et la synchronisation des tâches au sein d'une région parallèle :
  - DO
  - SECTIONS
  - WORKSHARE²

Partage du travail : boucle parallèle

- La directive DO (FOR en C) permet un parallélisme par répartition des itérations d'une boucle.
- Le mode de répartition des itérations peut être spécifié dans la clause SCHEDULE (codé dans le programme ou grâce à une variable d'environnement)
- Par défaut, une synchronisation globale est effectuée en fin de construction END DO (sauf si NOWAIT)
- Il est possible d'introduire plusieurs constructions DO dans une région parallèle.
- Les indices de boucles sont privées même si ils sont indiqués comme partagés.

Directives DO et PARALLEL DO

```
Program boucle
  implicit none
  integer, parameter :: n=256
  integer            :: i, j
  real, dimension(n, n) :: tab
  !$OMP PARALLEL
        ...                ! Code répliqué
    !$OMP DO
       do j=1, n      ! Boucle partagée
         do i=1, n     ! Boucle répliquée
               tab(i, j) = i*j
         end do
       end do
    !$OMP END DO
  !$OMP END PARALLEL
end program boucle
```

```
Program parallelboucle
  implicit none
  integer, parameter :: n=256
  integer            :: i, j
  real, dimension(n, n) :: tab
  !$OMP PARALLEL DO
     do j=1 n        ! Boucle partagée
        do i=1, n     ! Boucle répliquée
              tab(i, j) = i*j
       end do
      end do
  !$OMP END PARALLEL DO
end program boucle
```

PARALLEL DO est une fusion des 2 directives

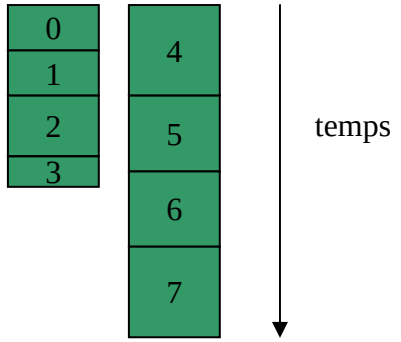
Attention : END PARALLEL DO inclut une barrière de synchronisation

Répartition du travail : clause SCHEDULE

!$OMP DO SCHEDULE(STATIC,taille_paquets)

Avec par défaut taille_paquets=nbre_itérations/nbre_thread

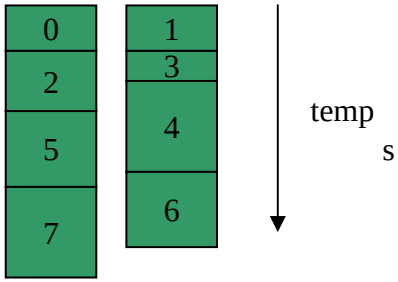
Ex : 8 itérations (0 à 7), 2 threads : la taille des paquets par défaut est de 4



!$OMP DO SCHEDULE(DYNAMIC,taille_paquets)

Les paquets sont distribués aux threads libres de façon dynamique

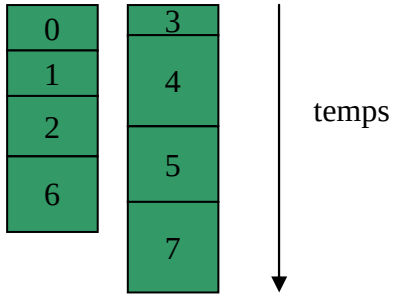
Tous les paquets ont la même taille sauf le dernier, par défaut la taille des paquet est 1.



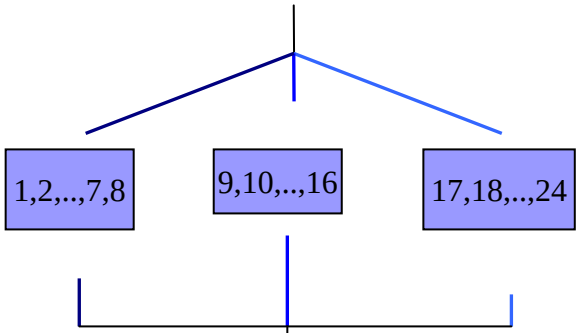
!$OMP DO SCHEDULE(GUIDED,taille_paquets)

taille_paquets : taille minimale des paquets (1 par défaut) sauf le dernier.

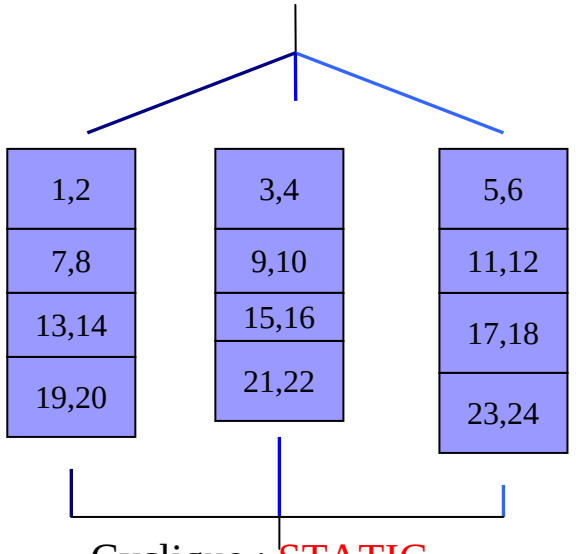
Maximise la taille des paquets en début de boucle (ici 3) puis diminue pour équilibrer la charge (ici 1)



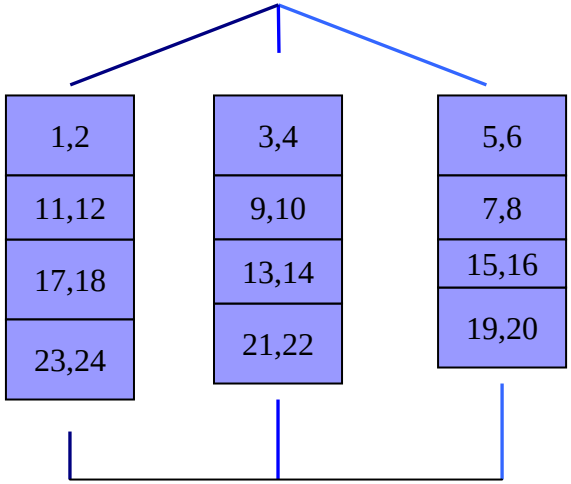
Répartition du travail : clause SCHEDULE



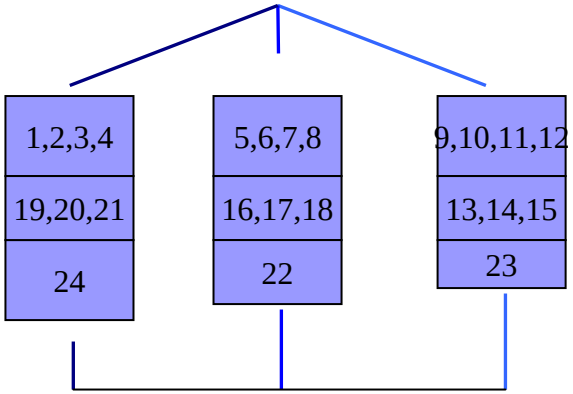
Mode static, avec
Taille paquets=nb itérations/nb threads



Cyclique : STATIC



Glouton : DYNAMIC



Glouton : GUIDED

Répartition du travail : clause SCHEDULE

Le choix du mode de répartition peut être différé à l'exécution du code :

Avec SCHEDULE(RUNTIME)

Prise en compte de la variable d'environnement **OMP_SCHEDULE**

Ex :

export OMP_SCHEDULE ="DYNAMIC,400"

Taille des paquets utilisés, par ordre de priorité :

1. la valeur spécifiée dans la clause SCHEDULE du DO ou PARALLEL DO courant
2. Avec la clause SCHEDULE(RUNTIME), la valeur spécifiée dans la variable d'environnement **OMP_SCHEDULE**
3. Pour les types DYNAMIC et GUIDED, la valeur par défaut est 1
4. Pour le type STATIC, la valeur par défaut est nb itérations/nb threads

Réduction : pourquoi ?

Ex séquentiel :

```
Do i=1,N
   X=X+a(i)
enddo
```

En parallele :

```
!$OMP PARALLEL DO SHARED(X)
      do i=1,N
         X = X + a(i)
      enddo
!$OMP END PARALLEL DO
```

Reduction : opération associative appliquée à des variables scalaires partagées

Chaque tâche calcule un résultat partiel indépendamment des autres. Les réductions intermédiaires sur chaque thread sont visibles en local.

Les taches se synchronisent pour mettre à jour le résultat final dans une variable globale. (attention aux arrondis)

Réduction

Ex : !$ OMP DO REDUCTION(op:list) (op est un opérateur ou une fonction intrinsèque)

Une copie locale de chaque variable de la liste est attribuée à chaque thread et initialisée selon l'opération (par ex 0 pour +)

La clause s'appliquera aux variables de la liste si les instructions sont d'un des types suivants :

x = x opérateur expr
x = expr opérateur x
x = intrinsic (x,expr)
x = intrinsic (expr,x)
x est une variable scalaire
expr est une expression scalaire ne référençant pas x
intrinsic = MAX, MIN, IAND, IOR, IEOR
opérateur = +,*, .AND., .OR., .EQV., .NEQV.

```
program reduction
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 reduction
```

reduction.f

Autres clauses de la directive DO

- PRIVATE : privatise une variable
- FIRSTPRIVATE : privatise et assigne la dernière valeur affectée avant l'entrée dans la région //
- LASTPRIVATE : privatise et permet de conserver, à la sortie de la construction, la valeur calculée par la tâche exécutant la dernière itération de la boucle

```
program parallel
  implicit none
  integer, parameter :: n=9
  integer                  :: i,rang
  real                     :: temp
  integer   :: OMP_GET_THREAD_NUM
  !$OMP PARALLEL PRIVATE (rang)
     !$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
```

lastprivate.f

Cas d'une exécution ordonnée

- Il est parfois utile (debogage) d'exécuter une boucle de façon ordonnée.
- La Clause et Directive : ORDERED

  ➔ l'ordre d'exécution sera identique à celui d'une exécution séquentielle

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

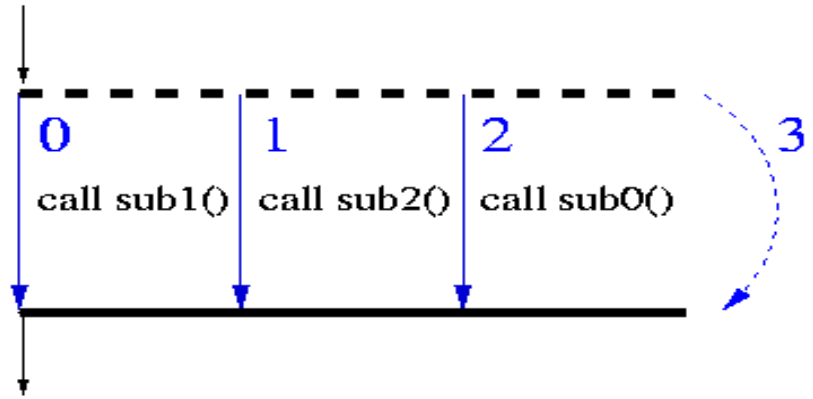
Essayez avec OMP_SCHEDULE="STATIC,2" par exemple

ordered.f

Partage du travail : sections parallèles

- But : Répartir l'exécution de plusieurs portions de code indépendantes sur différentes tâches
- Une section est une portion de code exécutée par une et une seule tâche
- Directive SECTION au sein d'une construction SECTIONS

```
program section
call OMP_SET_NUM_THREADS(4)
 !$OMP PARALLEL SECTIONS
 !$OMP SECTION
  call sub0()
 !$OMP SECTION
  call sub1()
 !$OMP SECTION
  call sub2()
 !$OMP END PARALLEL SECTIONS
```



Sections parallèles

- Les directives SECTION doivent se trouver dans l'étendue lexicale de la construction

- Les clauses admises :

  PRIVATE, FIRSTPRIVATE,
  LASTPRIVATE, REDUCTION

- END PARALLEL SECTIONS inclut une barrière de synchronisation (NOWAIT interdit)

Directive WORKSHARE[2]

destinée à permettre la parallélisation d'instructions Fortran 95 intrinsèquement parallèles comme :

- Les notations tableaux.
- Certaines fonctions intrinsèques (SUM, MATMUL, ..)
- Le FORALL
- Le WHERE

Restrictions :

Cette directive ne s'applique qu'à des variables partagées, dans l'extension lexicale de la construction WORKSHARE.

Une fonction ne peut être appelée que si elle est élémentaire (*elemental*).

Tout branchement vers l'extérieur de l'extension, de la directive WORKSHARE est illégal.

Cette directive n'est pas supportée par le compilateur intel

```
Program workshare
…
!$OMP PARALLEL PRIVATE(K)
!$OMP WORKSHARE !---[ Pas de clause prévues]
A(:) = B(:)
!$OMP ATOMIC
somme = somme + SUM(A) !--- C'est SUM(A) qui est
        partagée
WHERE (A /= 0.0)
F = 1 / A
ELSEWHERE (
F == A
END WHERE
FORALL (I=1:M, A(I)/=0.0) F=1/A(I)
K = SUM(A) !--- Attention K est indefini
!$OMP END WORKSHARE [NOWAIT]
!$OMP END PARALLEL
…
End program
```

Partage du travail : exécution exclusive

- Construction SINGLE

  Exécution d'une portion de code par une et une seule tâche (en général, la première qui arrive sur la construction).

  La clause NOWAIT permet de ne pas bloquer les autres tâches qui par défaut attendent sa terminaison.

  Clauses admises : PRIVATE, FIRSTPRIVATE

```
!$OMP SINGLE [clause [clause ...] ]
    ...
!$OMP END SINGLE
```

Partage du travail : exécution exclusive

- Construction MASTER

Exécution d'une portion de code par la tâche maître seule

Pas de synchronisation, ni en début ni en fin

Pas de clause

```
!$OMP MASTER
...
!$OMP END MASTER
```

Partage du travail : procédures orphelines

- Procédures appelées dans une région // et contenant des directives OpenMP

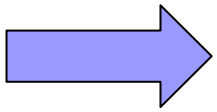
  Elles sont dites orphelines
- Attention : le contexte d'exécution peut être # selon le mode de compilation des unités de programme appelantes et appelées
- Attention aux variables locales aux subroutines (par défaut (-qsave, elles sont globales)

```
Program main
  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
  !$OMP PARALLEL
     call orphan(a,x,y,n)
  !$OMP END PARALLEL
End program main
```

```
Subroutine orphan(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
!$OMP DO
  DO i=1,n
    y(i) = SUM(a(i, : ) * x(:) )
  END DO
!$OMP END DO
End subroutine orphan
```

Open-MP

- Introduction
- Structure d'OPEN_MP
- Visibilité des données
- Partage du travail

   Synchronisation
- Conclusion

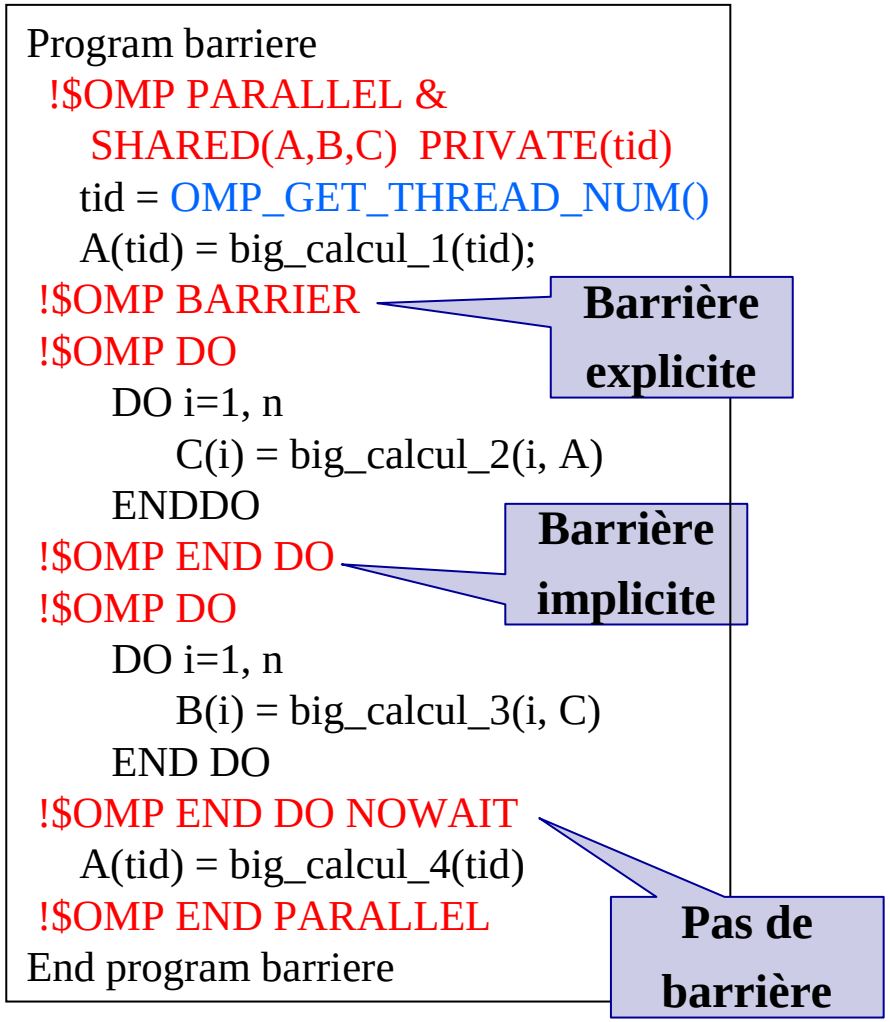
Synchronisation

- Synchronisation de toutes les tâches sur un même niveau d'instruction (barrière globale)
- Ordonnancement de tâches concurrentes pour la cohérence de variables partagées (exclusion mutuelle)
- Synchronisation de plusieurs tâches parmi un ensemble (mécanisme de verrou)

Synchronisation : barrière globale

- Par défaut à la fin des constructions parallèles, en l'absence du NOWAIT
- Directive BARRIER

  Impose explicitement une barrière de synchronisation: chaque tâche attend la fin de toutes les autres



Synchronisation : exclusion mutuelle régions critiques

- Directive CRITICAL

Elle s'applique sur une portion de code

Les tâches exécutent la région critique dans un ordre non-déterministe, mais une à la fois

Son étendue est dynamique

```
!$OMP PARALLEL & DEFAULT(SHARED)
    PRIVATE(i,j)
  DO j=1,n
!$OMP DO
      DO i=1,m
!$OMP CRITICAL
            somme = somme+a(i,j)
!$OMP END CRITICAL
      END DO
!$OMP END DO
!$OMP BARRIER
!$OMP SINGLE
      b(j) = somme
!$OMP END SINGLE
  END DO
!$OMP END PARALLEL
```

critical.f

Synchronisation : exclusion mutuelle mise à jour atomique

La directive ATOMIC s'applique seulement dans le cadre d'une mise à jour d'une case mémoire

Une variable partagée est lue ou modifiée en mémoire par une seule tâche à la fois

Agit sur l'instruction qui suit immédiatement si elle est de la forme :

- **x = x (op) exp**
- **ou x = exp (op) x**
- **ou x = f(x,exp)**
- **ou x = f(exp,x)**

Avec :

- **op** : +, -, *, /, .AND., .OR., .EQV., .NEQV.
- **f** : MAX, MIN, IAND, IOR, IEOR

```
Program atomic
  implicit none
  integer :: count, rang, &
      OMP_GET_THREAD_NUM
!$OMP PARALLEL PRIVATE(rang)
!$OMP ATOMIC
      count = count + 1
      rang=OMP_GET_THREAD_NUM()
      Print *, "rang : ", rang, &
            "count:", count
!$OMP END PARALLEL
Print *, "count:", count
End program atomic
```

Synchronisation : directive FLUSH

- Les valeurs des variables globales peuvent rester temporairement dans les registres pour des raisons d'optimisation

➔La directive FLUSH garantit que chaque thread a accès aux valeurs des variables globales modifiées par les autres threads.

```
Program anneau
  implicit none
  integer :: rang, nb_taches, synch=0, &
  OMP_GET_NUM_THREADS,
     OMP_GET_THREAD_NUM
!$OMP PARALLEL PRIVATE(rang,nb_taches)
      rang=OMP_GET_THREAD_NUM()
      nb_taches=OMP_GET_NUM_THREADS()
            if (rang == 0) then ; do
                        !$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
```

flush.f sans_flush.f

Standard openMP-2

- Mesure du temps elapsed (temps de restitution) propre à chaque thread : Omp_get_wtime()
- Directive WORKSHARE : parallélisation des notations f95 intrinsèquement parallèles (WHERE, FORALL, les notations tableaux)
- Clause COPYPRIVATE de la directive END SINGLE
- Accepte les notations tableaux de Fortran 90
- Ajout de la clause NUM_THREAD à la directive PARALLEL
- La clause REDUCTION est étendue aux tableaux

Performances – Partage des données

Objectif : réduire le temps de restitution d'un code et estimer son accélération par rapport à une exécution séquentielle

➔ **Rechercher les régions les plus coûteuses (profiling)**

➔ **Partager les données**

- Placer les variables locales dans la pile (stack) , déclarer de facon explicite le statut des variables (PRIVATE ou SHARED) – attention aux options par défaut du compilateur (-qnosave ou –auto)
- Lister les variables présentes dans les boucles avec la clause adaptée : SHARED, PRIVATE, LASTPRIVATE, FIRSTPRIVATE ou REDUCTION
- Les blocs common ne doivent pas être placés dans la liste PRIVATE, si leur visibilité globale doit être préservée.
- Toutes les entrées/sorties de variables globales en région parallèle doivent être synchronisées.

Performances – partage du travail

➔ **Partager le travail :**

- Minimiser le nombre de régions parallèles
- Adapter le nombre de tâches à la taille du problème à traiter afin de minimiser les surcoûts de gestion des tâches par le système (dont synchronisation) - Utiliser SCHEDULE(RUNTINE)
- ATOMIC et REDUCTION plus performants que CRITICAL
- Introduire un PARALLEL DO dans les boucles capables d'exécuter des itérations en //
- Essayer de supprimer les dépendances entre itérations en modifiant l'algorithme
- S'il reste des itérations dépendantes, introduire des constructions CRITICAL autour des variables concernées par les dépendances.
- Si possible, regrouper dans une unique région parallèle plusieurs structures DO.
- Dans la mesure du possible, paralléliser la boucle la plus externe.

Performances : Le false-sharing

- L'utilisation des structures en mémoire partagée peut induire une diminution de performance et une grosse limitation de scalabilité et peut finalement générer une sérialisation des calculs.
- **Pourquoi ?**
  - Perfomances ➔ utilisation du cache
  - Si plusieurs processeurs manipulent des données différentes mais adjacentes en mémoire, chaque mise à jour d'éléments individuels provoque un chargement complet d'une ligne de cache (mécanisme pour que les caches soient en cohérence avec la mémoire)

- Le False sharing dégrade les performances lorsque toutes les conditions suivantes sont réunies :
  - Des données partagées sont modifiées par plusieurs processeurs.
  - Des processeurs mettent à jour des données qui se trouvent dans la même ligne de cache.
  - Ces mises à jour ont lieu très fréquemment

Performances : Le false-sharing (2)

- Lorsque les données partagées ne sont que lues, cela ne génère pas de false sharing.

- **En général, le phénomène de false sharing peut être réduit en :**
  - Privatisant le plus possible les variables de type structure (tableaux)
  - Parfois en augmentant la taille des tableaux (taille des problèmes ou augmentation artificielle)
  - Parfois en modifiant la façon dont les itérations d'une fois sont partagées entre les threads (taille des paquets)

Performances : parallélisation conditionnelle

La clause IF permet de mettre en place une parallélisation conditionnelle (ex : ne paralléliser une boucle que si sa longueur est suffisamment grande)

```
Program parallel
  implicit none
  integer, parameter :: n=1025
  integer :: i,j
real, dimension(n,n) :: a,b
call random_number(a)
!$OMP PARALLEL DO SCHEDULE(RUNTIME) &
!$OMP IF(n.gt.512)
  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
```

parallel_if.f

En plus …

- Les variables sont stockées dans la zone mémoire (et cache) au moment de leur initialisation. Il est donc nécessaire de paralléliser l'initialisation des éléments d'un tableau de la même façon (même mode même taille des paquets) que le travail ..
- Placer correctement les données en mémoire : (encore plus vrai dans le cas des machines ccNUMA)

Fonctions de bibliothèque

Fonctions relatives aux verrous (s'appliquent à une thread)

- omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock()

Fonctions de l'environnent d'exécution

- Modifier/vérifier le nombre de threads
  - omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()
- Autoriser ou pas l'imbrication des régions parallèles et l'ajustement dynamique du nombre de threads dans les régions parallèles
  - omp_set_nested(), omp_set_dynamic(), omp_get_nested(), omp_get_dynamic()
- Tester si le programme est actuellement dans une région parallèle
  - omp_in_parallel()
- Combien y a t-il de processeurs reconnus par le système
  - omp_num_procs()

OpenMP versus MPI

- OpenMP exploite la mémoire commune à tous les processus. Toute communication se fait en lisant et en écrivant dans cette mémoire (et en utilisant des mécanismes de synchronisation)
- MPI est une bib de prgm permettant la communication entre différents processus (situés sur différentes machines ou non), les communications se font par des envois ou réceptions explicites de messages.
- Pour le programmeur : réécriture du code avec MPI, simple ajout de directives dans le code séquentiel avec OpenMP
- possibilité de mixer les deux approches dans le cas de cluster de machines smp (performance)
- Choix fortement dépendant : de la machine, du code, du temps que veut y consacrer le développeur et du gain recherché.
- Extensibilité supérieure avec MPI

Références

- www.openmp.org
- www.openmp.org/specs/mp-documents/cspec10.pdf
- http://www.openmp.org/specs/mp-documents/fspec20.pdf
- http://www.idris.fr/data/cours/parallel/openmp/choix_doc.html
- http://www.cerfacs.fr/~gondet/mem_stdd_vpp.pdf