OpenMP

Françoise ROCH

Ecole IN2P3 2010

Modèle de programmation multi-tâches sur architecture à mémoire partagée

- Plusieurs tâches s'exécutent en parallèle
- La mémoire est partagée (physiquement ou virtuellement)
- ***Les communications entre tâches se font par lectures et écritures dans la mémoire partagée*.**

Les processeurs multicoeurs généralistes partagent une mémoire commune

Les tâches peuvent être attribuées à des « cores » distincts

Modèle de programmation multi-tâches sur architecture à mémoire partagée

- La librairie Pthreads : librairie de threads POSIX, adoptée par la plupart des OS

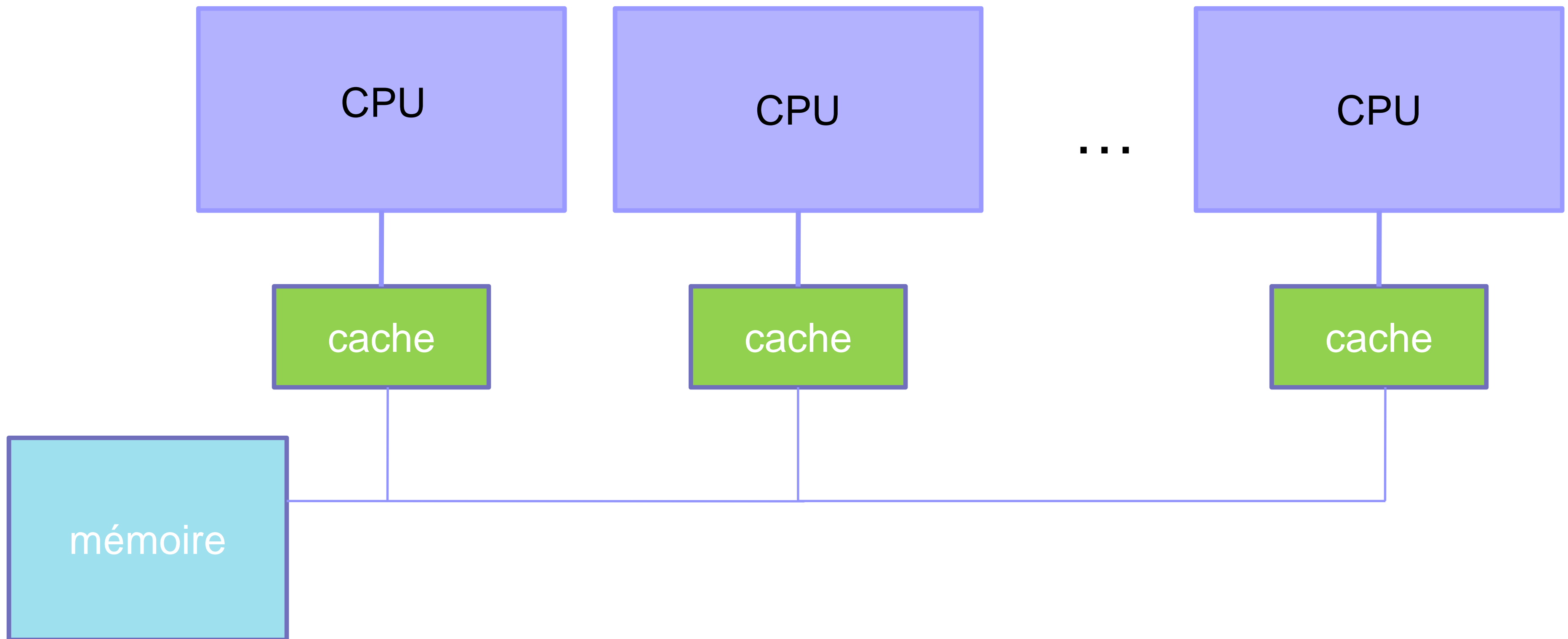
  L’écriture d’un code nécessite un nombre considérable de lignes spécifiquement dédiées aux threads

  Ex : paralléliser une boucle implique :

  déclarer les structures de thread, créer les threads, calculer les bornes de boucles, les affecter aux threads, …

- OpenMP : une alternative plus simple pour le programmeur

Programmation multi-tâches sur les architectures UMA

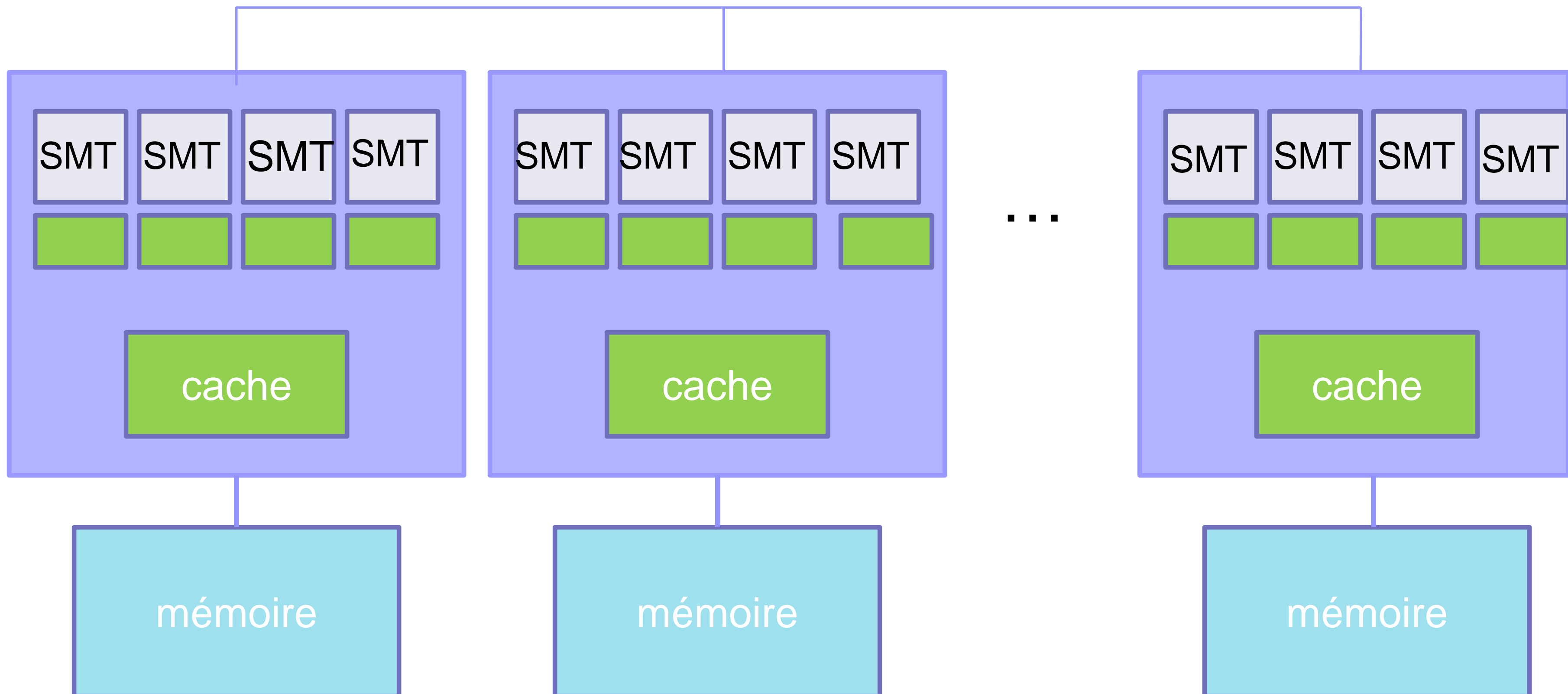


La mémoire est commune,

Des architectures à accès mémoire uniforme (UMA)

Un problème inhérent : les contentions mémoire

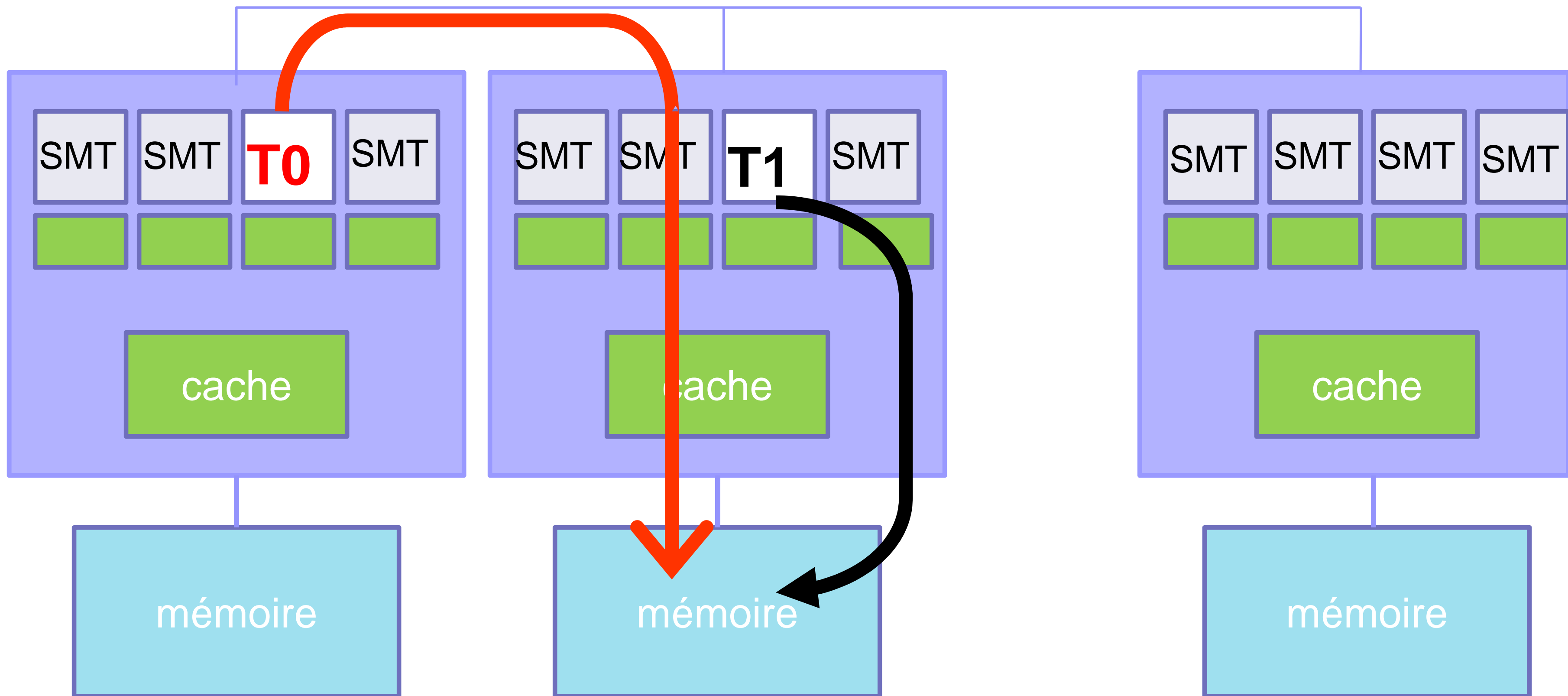
Programmation multi-tâches sur les architectures multicoeurs NUMA



La mémoire est directement attachée aux puces multicoeurs

Des architectures à accès mémoire non uniforme NUMA

Programmation multi-tâches sur les architectures multicoeurs NUMA



ACCES DISTANT

ACCES LOCAL

Caractéristiques du modèle OpenMP

- Gestion de « threads » transparente et portable
- Facilité de programmation

Mais

- Problème de localité des données
- Mémoire partagée mais non hiérarchique
- Efficacité non garantie (impact de l’organisation matérielle de la machine)
- Passage à l’échelle limité, parallélisme modéré

OpenMP
(**Open** specifications for **M**ulti**P**rocessing)

- Introduction
- Structure d’OpenMP
- portée des variables
- Constructions de partage du travail
- Construction task
- Synchronisation
- Performances
- Conclusion

Introduction : supports d'OpenMP

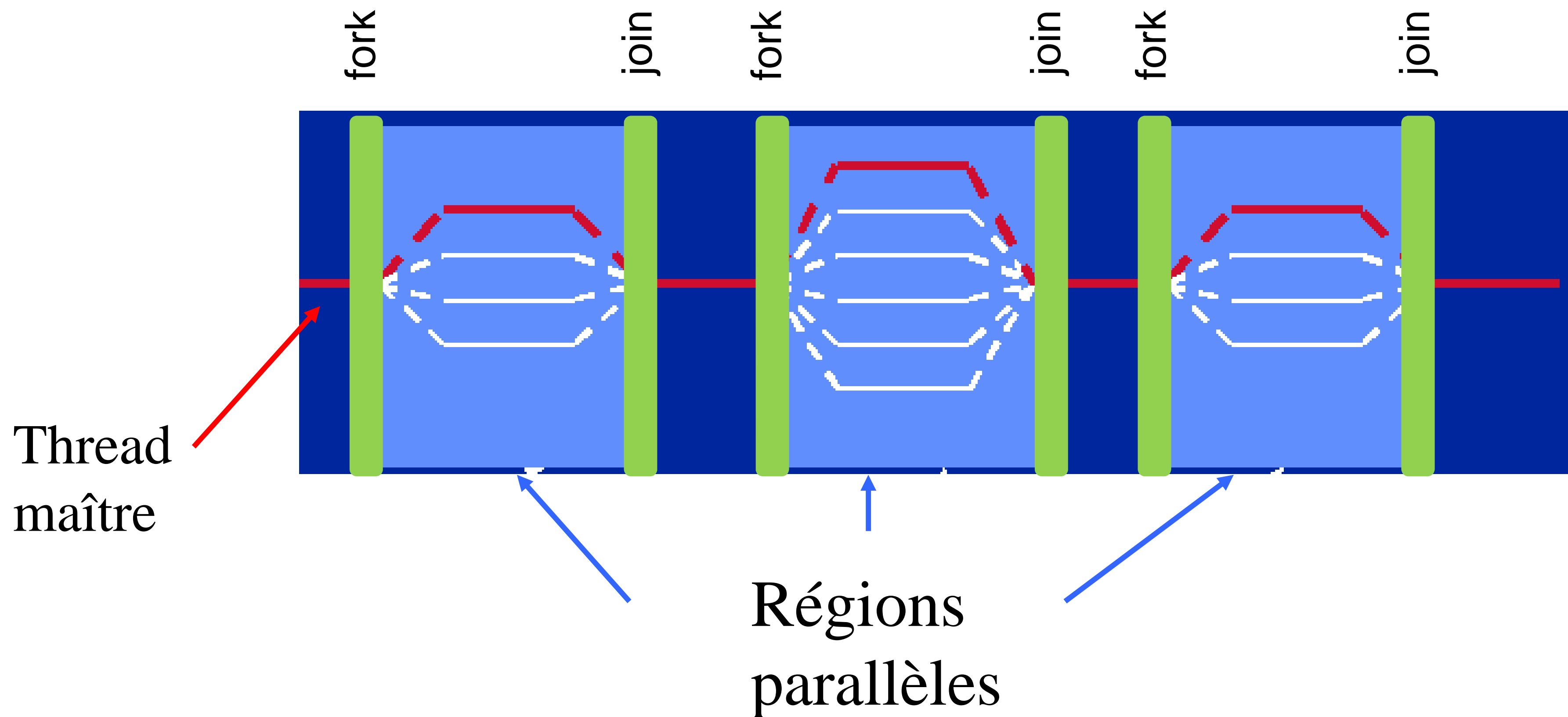
- La parallélisation multi-tâches existait avant pour certains compilateurs (Ex:Cray,NEC,IBM)
- OpenMP est une API pour un modèle à mémoire partagé
- Spécifications pour les langages C/C++, Fortran
- Supporté par beaucoup de systèmes et de compilateurs

OpenMP-2 depuis fin2000

Specs : http://openmp.org

Introduction : Modèle d'exécution

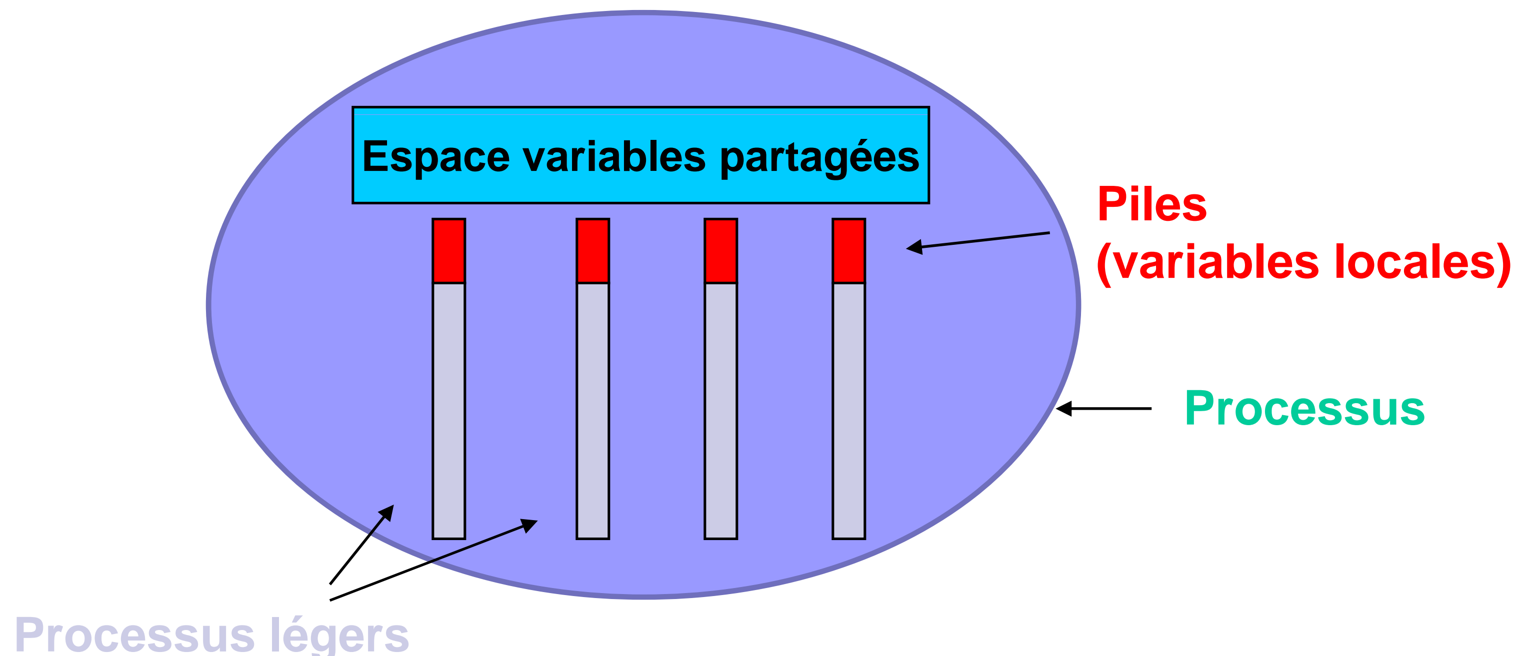
Un programme OpenMP est exécuté par un processus unique (sur un ou plusieurs cores)



Introduction : les threads

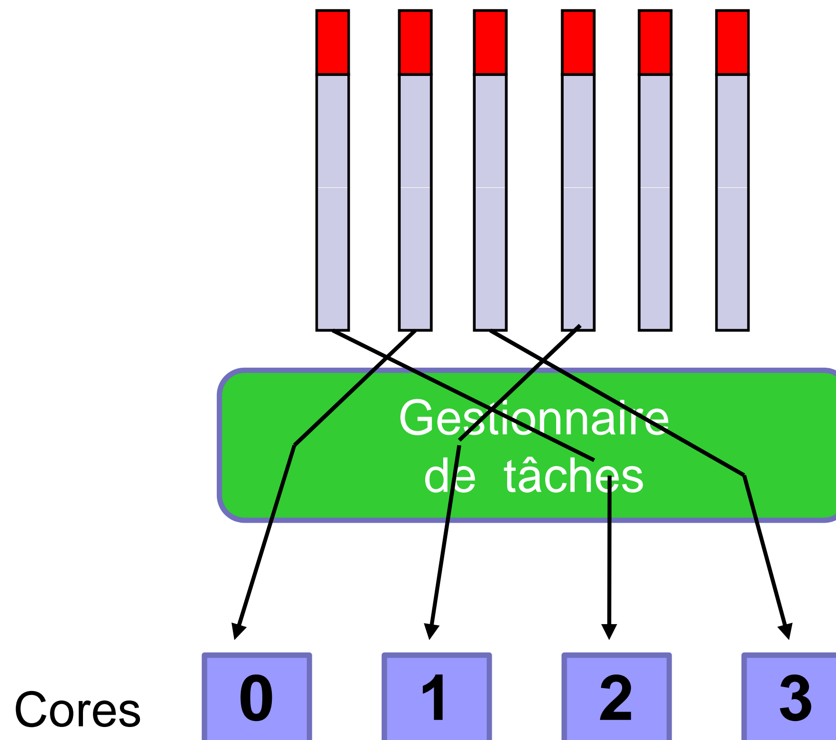
Les threads accèdent aux mêmes ressources que le processus.

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



Introduction : exécution d'un programme OpenMP sur un multicoeur

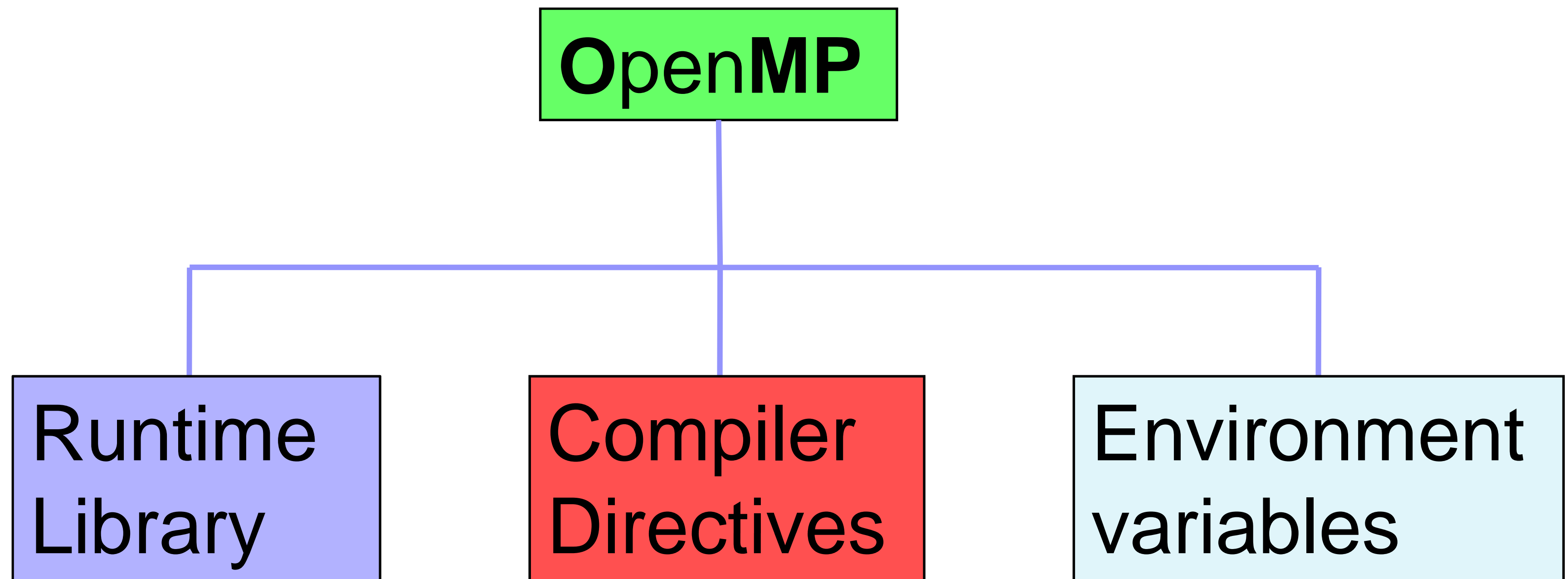
Le gestionnaire de tâches du système d'exploitation affecte les tâches aux cores.



- Introduction

- Portée des données
- Constructions de partage du travail
- Construction task
- Synchronisation
- Performances
- Conclusion

Structure d'OpenMP : architecture logicielle



Structure d'OpenMP : format des directives/pragmas

Sentinelle directive [clause[clause]..]

fortran

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

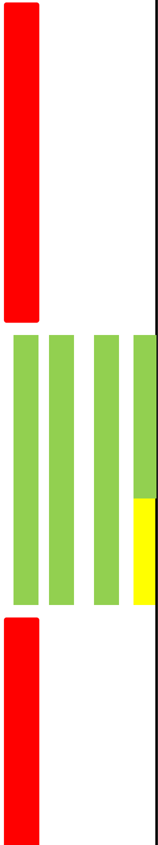
C/C++

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

La ligne est interprétée si option openmp à l'appel du compilateur sinon commentaire

➔ portabilité

Structure d’OpenMP : Construction d’une région parallèle



fortran

```
!$USE OMP_LIB

PROGRAM example

Integer :: a, b, c

!  Code sequentiel  execute par le maître

!$OMP PARALLELPRIVATE(a,b)
!$OMP& SHARED(c)
.
!  Zone parallele executee par toutes les threads

!$OMP END PARALLEL

! Code sequentiel

END PROGRAM example
```

C/C++

```
#include <omp.h>

Main  () {

Int a,b,c:

/*  Code sequentiel  execute par  le maître */

#pragma omp parallel private(a,b)  \
                              shared(c)
   {
      /* Zone parallele executee par toutes les
threads */
    }

!*  Code Sequentiel */

}
```

Clause IF de la directive PARALLEL

- Création conditionnelle d'une région parallèle clause IF(expression_logique)

fortran

```
!Code sequentiel
!$OMP PARALLEL IF(expr)
! Code parallele ou sequentiel suivant la valeur de expr
!!$OMP END PARALLEL
! Code sequentiel
```

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

Structure d'OpenMP : prototypage

Il existe :

- un module fortran 95 OMP_LIB
- un fichier d'inclusion C/C++ omp.h

qui définissent les prototypes de toutes les fonctions de la librairie OpenMP :

fortran

```
!$ use OMP_LIB
Program example
!$OMP PARALLEL PRIVATE(a,b) &
          ...
tmp= OMP_GET_THREAD_NUM()
!$OMP END PARALLEL
```

C/C++

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

Threads OpenMP

- Définition du nombre de threads

  Via une variable d'environnement OMP_NUM_THREADS

  Via la routine : OMP_SET_NUM_THREADS()

  Via la clause NUM_THREADS() de la directive PARALLEL

- Les threads sont numérotées

  le nombre de threads n'est pas nécessairement égale au nombre de cores 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

Structure d'OpenMP : Compilation et exécution

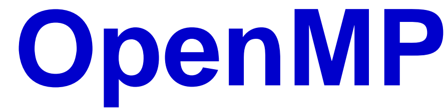
xlf_r –qsmp=omp prog.f (IBM)
ifort (ou icc) –openmp prog.f (INTEL)
f90 (ou cc ou CC) –openmp prog.f (SUN Studio)
gcc –fopenmp prog.f (GNU)

export OMP_NUM_THREADS=2

./a.out

ps –eLF
USER PID PPID LWP C NLWP SZ RSS PSR
…

OpenMP



- Introduction
- Structure d'OpenMP

  ➡ **Portée des variables**
- Constructions de partage du travail
- Construction task
- Synchronisation
- Performances
- Conclusion

Rappel : allocation mémoire- portée des variables

- **Variables statiques et automatiques**
  - statique : emplacement en mémoire défini dès sa déclaration par le compilateur
  - automatique : emplacement mémoire attribué au lancement de l'unité de programme où elle est déclarée (existence garantie que pendant 'exécution de l'unité).
- **Variables globales**

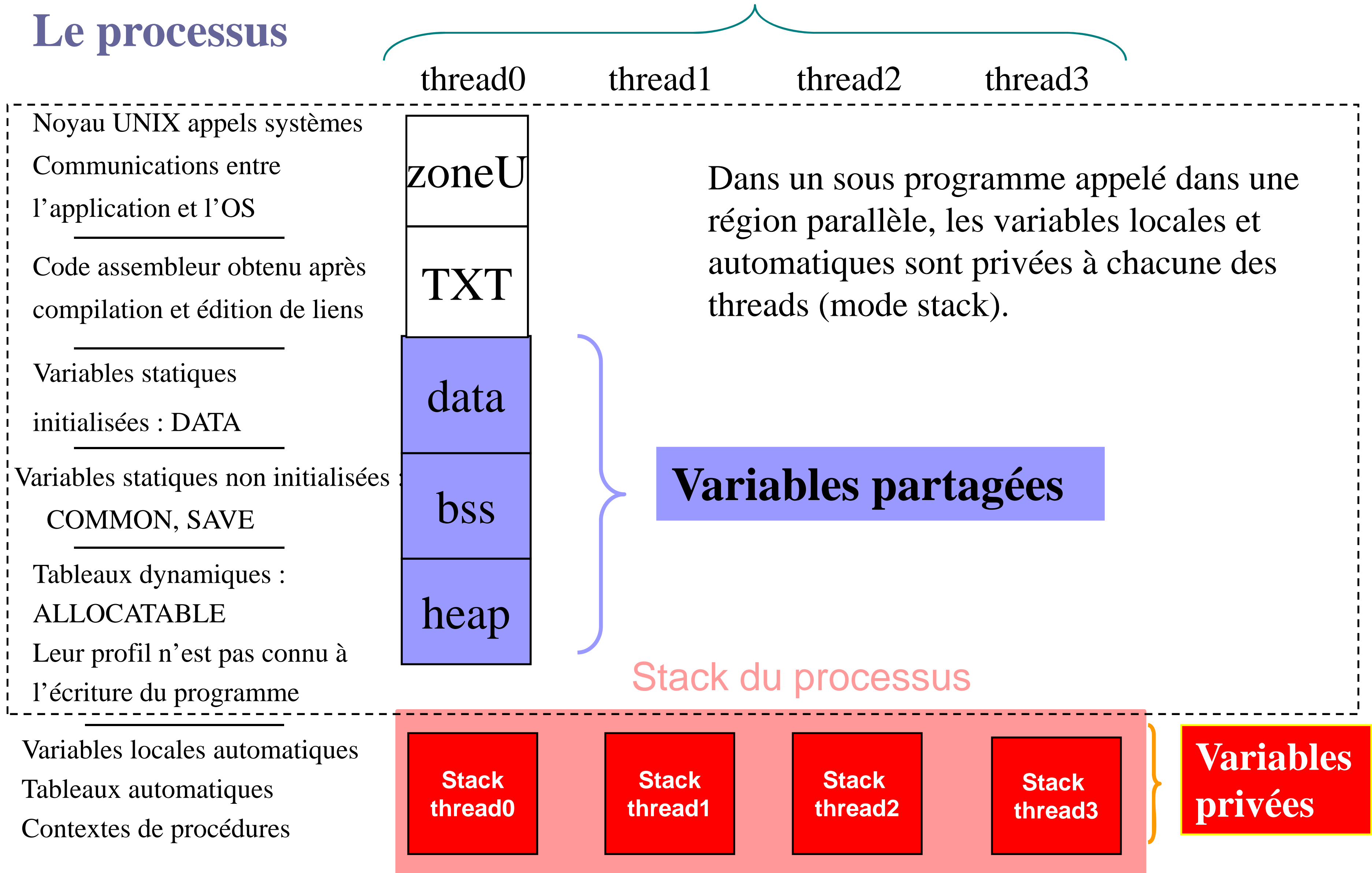
  globale : déclarée au début du programme principal, elle est statique

  2 cas :
  - initialisées à la déclaration (exemple : *parameter, data*)
  - non-initialisées à la déclaration (exemple : en fortran les *common*,

  en C les variables d'unités de fichier*, les variables externes ou static*
- **Variables locales**

  variable à portée restreinte à l'unité de programme où elle est déclarée, 2 catégories :
  - **Variables locales automatiques**
  - **Variables locales rémanentes (statiques) si elles sont :**
    - a) initialisées explicitement à la déclaration,
    - b) déclarées par une instruction de type DATA,
    - c) déclarées avec l'attribut SAVE => valeur conservée entre 2 appels

Rappel : stockage des variables



Statut d'une variable

Le statut d'une variable dans une zone parallèle est :

- soit SHARED, elle se trouve dans la mémoire globale
- soit PRIVATE, elle est dans la pile de chaque thread, sa valeur est indéfinie à l'entrée de la zone

- Déclarer le statut d'une variable

!$OMP PARALLEL PRIVATE(list)
!$OMP PARALLEL FIRSTPRIVATE(list)
!$OMP PARALLEL SHARED(list)

- Déclarer un statut par défaut

Clause DEFAULT(PRIVATE|SHARED|NONE)

```
!$USE OMP_LIB
program private_var.f
integer:: tmp =999
integer, dimension(4)::tab
Integer :: OMP_GET_THREAD_NUM
Call OMP_SET_NUM_THREADS(4)
!$OMP PARALLEL PRIVATE(tmp)
print *, tmp
 tab(OMP_GET_THREAD_NUM()+1)=tmp
tmp= OMP_GET_THREAD_NUM()
print *, OMP_GET_THREAD_NUM(), &
& tab(OMP_GET_THREAD_NUM()+1)
!$OMP END PARALLEL
print *, tmp
end
```

Clauses de la directive PARALLEL

- NONE

  Equivalent de l' IIMPLICIT NONE en Fortran. Toute variable devra avoir un statut défini explicitement
- SHARED (liste_variables)

  Variables partagées entre les threads
- PRIVATE (liste_variables)

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

  Variable initialisée avec la valeur que la variable d'origine avait juste avant la section parallèle
- DEFAULT (PRIVATE | SHARED|NONE)

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

Statut d’une variable transmise par arguments

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

```
program statut
 integer :: a=100, b
 integer OMP_GET_THREAD_NUM
!$OMP PARALLEL PRIVATE(b)
      call sub(a,b)
      print *,b
!$OMP END PARALLEL
end program  statut

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

La directive THREADPRIVATE

La directive THREADPRIVATE :

Permet de rendre privé aux threads

- un bloc COMMON (Fortran): les modifications apportées au bloc par une thread ne sont plus visibles des autres threads.
- Une variable globale, un descripteur de fichier ou des variables statiques (en C)

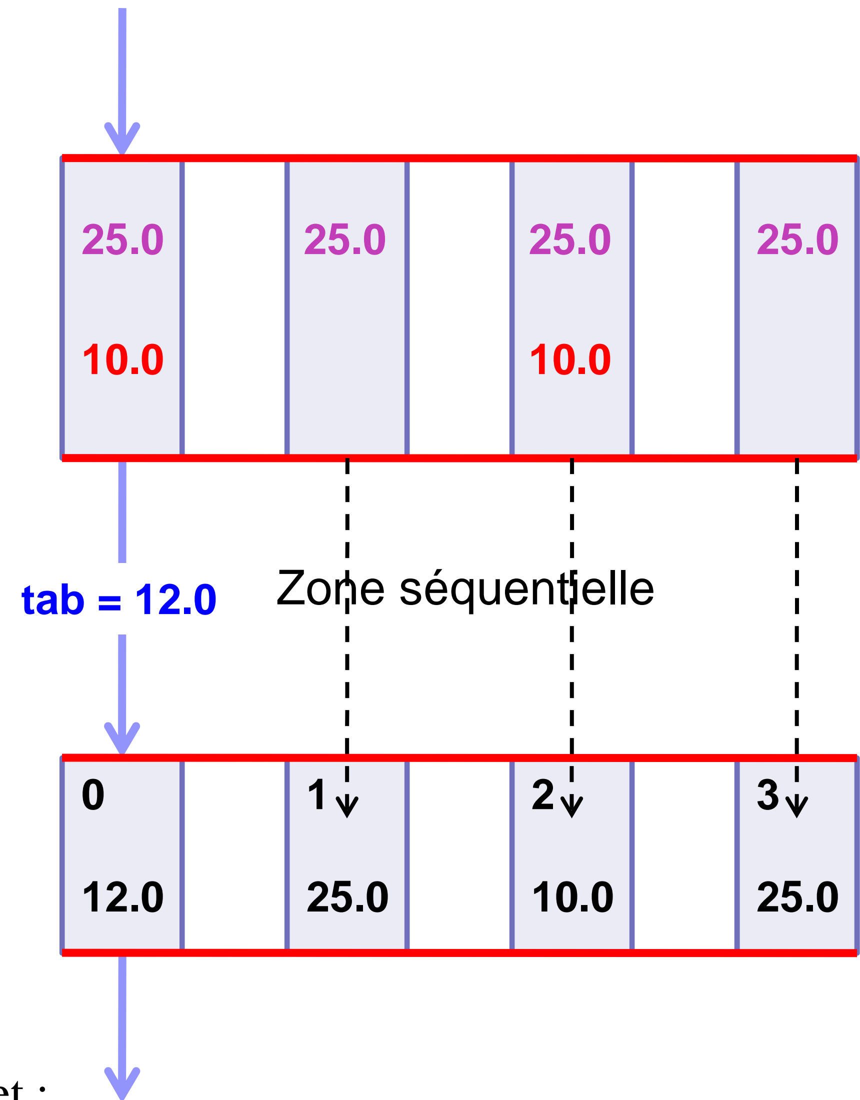
L'instance de la variable est persistante d'une région parallèle à l'autre (sauf si le mode dynamic est actif).

clause COPYIN : permet de transmettre la valeur des instances privées à toutes les tâches

```
program threadpriv
integer:: tid, x, OMP_GET_THREAD_NUM
common  /C1/ x
!$OMP THREADPRIVATE(/C1/)
!$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
```

Threadprivate

```
real, dimension(m,n) :: tab
integer :: moi,
   omp_get_thread_num
common /C1/ tab,moi
!$OMP THREADPRIVATE (/C1/)
tab(:,:) = 25.0
!$OMP PARALLEL COPYIN(/C1/)
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 *, moi, tab(m,n)
!$OMP END PARALLEL
```



La directive THREADPRIVATE a un double effet :

- Ces variables deviennent persistantes d'une région // à l'autre si le mode dynamic n'est pas activé
- L'instance des variables des zones séquentielles est aussi celle de la thread 0

Allocation mémoire

- L'option par défaut des compilateurs est généralement : variables locales allouées dans la stack => privé ,

  mais certaines options permettent de changer ce défaut et il est recommandé de ne pas utiliser ces options pour OpenMP

- Une opération d'allocation ou désallocation de mémoire sur un variable privée sera locale à chaque tâche

- Si une opération d'allocation/désallocation de mémoire porte sur une variable partagée, l'opération doit être effectuée par la tâche maître seule.

- En fortran ne mettre en équivalence que des variables de même statut (SHARED ou PRIVATE), même dans le cas d'une association par pointeur.

Allocation mémoire
Taille du “stack”

- La taille du stack est limitée , différentes variables d’environnement ou fonctions permettent d’agir sur cette taille
- La pile (stack) a une taille limite pour le shell (variable selon les machines). (ulimit –s)(ulimit –s unlimited), valeurs exprimées en 1024-bytes.
  - Ex: ulimit –s 65532
- OpenMP :

Variable d’environnement OMP_STACKSIZE : définit le nombre d’octets que chaque thread OpenMP peut utiliser pour sa stack privée

Allocation mémoire
Taille du "stack"

- INTEL :

KMP_STACKSIZE : définit le nombre d'octets que chaque thread OpenMP peut utiliser pour sa stack privée

KMP_GET_STACKSIZE() : renvoie le nombre d'octets alloué pour chaque thread pour sa stack privée

KMP_SET_STACKSIZE(size) : permet de définir le nb d'octets que chaque thread peut utiliser pour sa stack privée

Quelques précisions

- Les variables privatisées dans une région parallèle ne peuvent être re-privatisées dans une construction parallèle interne à cette région.

- fortran: les pointeurs et les tableaux ALLOCATABLE peuvent être PRIVATE ou SHARED
  mais pas LASTPRIVATE ni FIRSTPRIVATE.

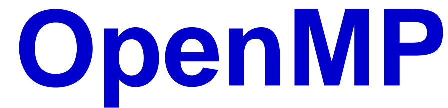
- Les tableaux à taille implicite (A(*)) et les tableaux à profil implicite (A(:)) grâce à une interface explicite Ne peuvent être déclarés ni PRIVATE, ni FIRSTPRIVATE ou LASTPRIVATE

Quelques précisions

- 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

OpenMP



- Introduction
- Structure d'OpenMP
- Portée des données
- ➡ **Constructions de partage du travail**
- Construction task
- Synchronisation
- Performances
- Conclusion

Partage du travail

- Répartition d’une boucle entre les threads (boucle //)
- Répartition de plusieurs sections de code entre les threads, une section de code par thread (sections //)
- Exécution d’une portion de code par une seule Thread
- Exécution de plusieurs occurences d’une même procédure par différentes threads (orphaning)
- Exécution par différentes threads de différentes unités de travail provenant de constructions fortran95

Portée d'une région parallèle

```
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
```

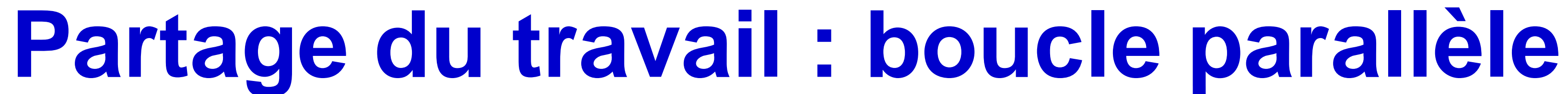
La portée d'une région parallèle s'étend :

- au code contenu lexicalement dans cette région (étendue statique)
- au code des sous programmes appelés

L'union des deux représente l'étendue dynamique

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
  - SINGLE
  - MASTER
  - WORKSHARE



Partage du travail : boucle parallèle

Directive DO (for en C) :

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)
- Une synchronisation globale est effectuée en fin de construction END DO (sauf si NOWAIT)
- Possibilité d'avoir plusieurs constructions DO dans une région parallèle.
- Les indices de boucles sont entiers et privées.
- Les boucles infinies et *do while* ne sont pas parallélisables

Directives DO et PARALLEL DO

```
Program loop
  implicit none
  integer, parameter :: n=1024
  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 loop
```

```
Program parallelloop
  implicit none
  integer, parameter :: n=1024
  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 parallelloop
```

PARALLEL DO est une fusion des 2 directives

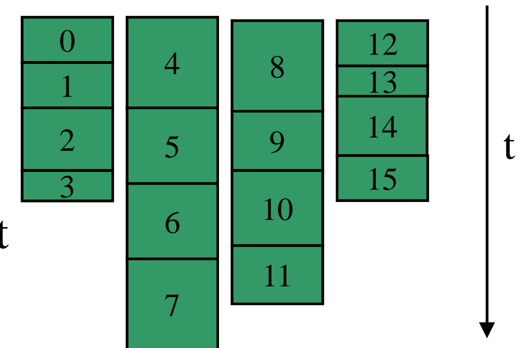
Attention : END PARALLEL DO inclut une barrière de synchronisation

Répartition du travail : clause SCHEDULE

!$OMP DO SCHEDULE(STATIC,taille_paquets)

Avec par défaut taille_paquets=nbre_itérations/nbre_thread

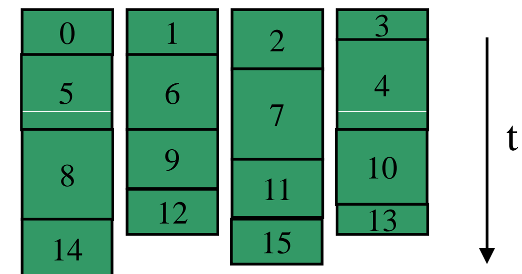
Ex : 16 itérations (0 à 15), 4 threads : la taille des paquets par défaut est de 4



!$OMP DO SCHEDULE(DYNAMIC,taille_paquets)

Les paquets sont distribués aux threads libres de façon dynamique

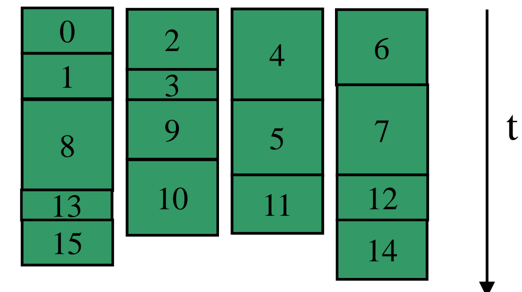
Tous les paquets ont la même taille sauf éventuellement le dernier, par défaut la taille des paquet est 1.



!$OMP DO SCHEDULE(GUIDED,taille_paquets)

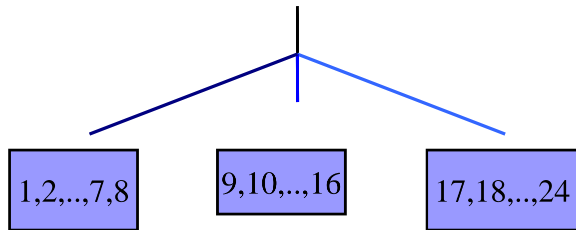
taille_paquets : taille minimale des paquets (1 par défaut) sauf le dernier.

Taille des paquets maximale en début de boucle (ici 2) puis diminue pour équilibrer la charge.

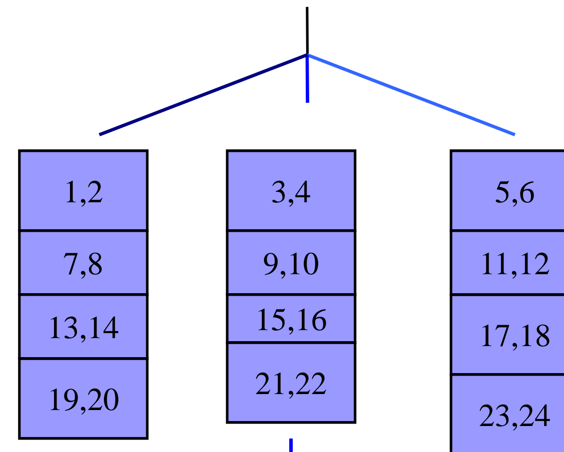


Répartition du travail : clause SCHEDULE

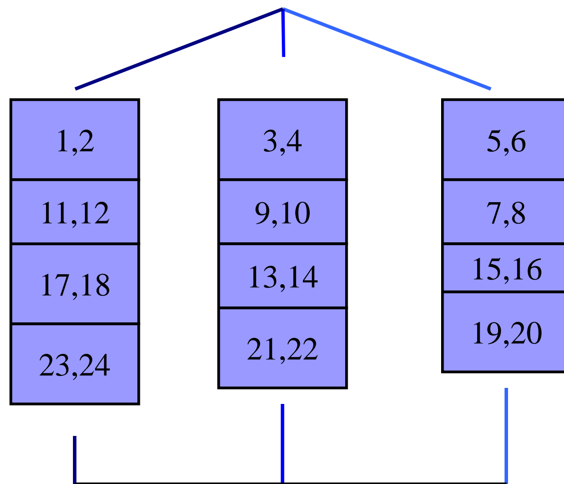
Ex: 24 itérations, 3 threads



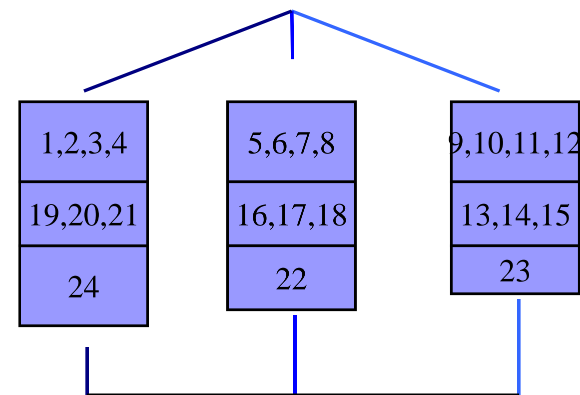
Mode static, avec
Taille paquets=nb itérations/nb threads



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

Parallélisme imbriqué

Autoriser le parallélisme imbriqué

- Via une variable d'environnement ;
  Export OMP_NESTED= TRUE
- Via la routine OMP_SET NESTED()

```
!$use OMP_LIB

  call OMP_SET_NESTED(.TRUE.)
```

```
#include <omp.h>

        omp_set_nested(1)
```

Remarque : il est fréquent d'avoir des boucles iimbriquées, de taille parfois iinsuffisante pour une parallélisation efficace. Le parallélisme sur des boucles imbriquées peut présenter un gain par le regroupement des boucles (COLLAPSING).

Parallélisme imbriqué

```
#pragma omp parallel shared(n, a, b)
{
  #pragma omp for
   for (int i=0; i<n; i++)
   {
      a[i] = i + 1;
      #pragma omp parallel for
      for (int j=0; j<n; j++)
         b[i][j] = a[i];
   }
}
```

```
#pragma omp parallel shared(n, a, b)
{
  #pragma omp for
   for (int i=0; i<n; i++)
   {
      a[i] = i + 1;
      #pragma omp for
      for (int j=0; j<n; j++)
         b[i][j] = a[i];
   }
}
```

L'imbrication de directives de partage du travail n 'est pas valide si on ne crée pas une nouvelle région parallèle.

Reduction : 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.

Puis les tâches se synchronisent pour mettre à jour le résultat final dans une variable globale, en appliquant le même opérateur aux résultats partiels.

Attention, pas de garantie de résultats identiques d'une exécution à l'autre, les valeurs intermédiaires peuvent être combinées dans un ordre aléatoire

Reduction

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

Les variables de la liste doivent être partagées dans la zone englobant la directive!;

Une copie locale de chaque variable de la liste est attribuée à chaque thread et initialisée selon l'opération (par ex 0 pour +, 1 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
```

Autres clauses de la directive DO

- PRIVATE
- 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
```

Exécution ordonnée : ORDERED

Il est parfois utile, à l'intérieur dune boucle, d'exécuter une zone séquentiellement

- Pour du débogage
- Pour des IOs ordonnées

La Clause et Directive : ORDERED

➔ l'ordre d'exécution de la zone 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
```

Dépliage de boucles imbriquées : directive COLLAPSE

La Clause COLLAPSE(N) permet de spécifier un nombre de boucle à déplier pour créer un large espace des itérations

Les boucles doivent être parfaitement imbriquées

Ex. : Si les boucles en i et j peuvent être parallélisées, et si N et M sont petits, on peut ainsi paralléliser sur l'ensemble du travail correspondant aux 2 boucles

```
!$OMP PARALLEL DO COLLAPSE(2)
    do i=1, N
        do j=1, M
            do k=1, K
                func(i,j,k)
            end do
        end do
    end do
!$OMP END PARALLEL DO
```

Dépliage de boucles imbriquées : directive COLLAPSE

Les boucles ne sont pas parfaitement imbriquées **interdit**

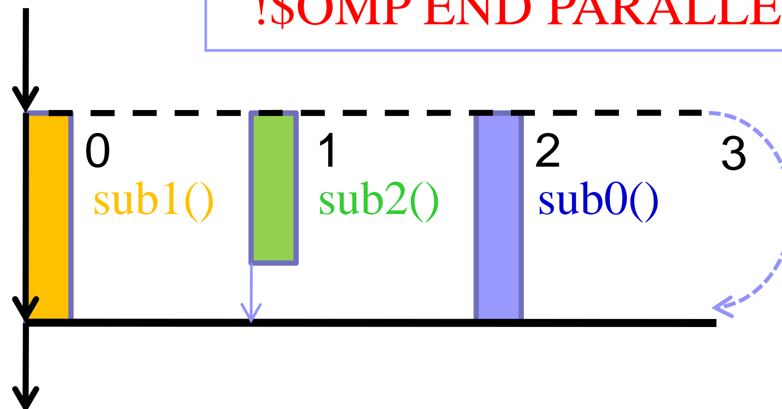
Espace d'itération triangulaire **interdit**

```
!$OMP PARALLEL DO COLLAPSE(2)
   do i=1, N
      func1(i)  interdit !
      do j=1, i        interdit !
       do k=1, K
          func(i,j,k)
        end do
       end do
   end do
!$OMP END PARALLEL DO
```

Partage du travail : SECTIONS parallèles

- But : Répartir l'exécution de plusieurs portions de code indépendantes sur différentes tâches
- Une section: 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

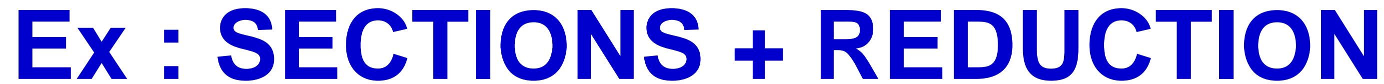
- Les directives SECTION doivent se trouver dans l'étendue lexicale de la construction

- Les clauses admises :

  PRIVATE, FIRSTPRIVATE, LASTPRIVATE, REDUCTION

LASTPRIVATE : valeur donnée par la thread qui exécute la dernière section

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



Ex : SECTIONS + REDUCTION

```
program section
Logical  b
!$OMP PARALLEL SECTIONS REDUCTION(.AND. : b)
  !$OMP SECTION
      b = b .AND. func0()
  !$OMP SECTION
      b = b .AND. func1()
  !$OMP SECTION
      b = b .AND. func2()
  !$OMP END PARALLEL SECTIONS
IF (b) THEN
        print(*, "All of the functions succeeded")
ENDIF
```

Directive WORKSHARE

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

- Les notations tableaux.
- Certaines fonctions intrinsèques
- Instruction FORALL, WHERE

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

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

WHORSHARE n'admet aucune clause.

Fusion PARALLEL WORKSHARE possible

clause NOWAIT non autorisée

```
Program workshare
…
!$OMP PARALLEL PRIVATE(K)
!$OMP WORKSHARE
A(:) = B(:)
somme = somme + SUM(A)
FORALL (I=1:M)  A(I)=2*B(I)
!$OMP END WORKSHARE &
      [NOWAIT]
!$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).
- clause NOWAIT : permet de ne pas bloquer les autres tâches qui par défaut attendent sa terminaison.
- Clauses admises : PRIVATE, FIRSTPRIVATE,
- COPYPRIVATE(var): mise à jour des copies privées de var sur toutes les tâches (après END SINGLE en Fortran)

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

Partage du travail : exécution exclusive

Construction MASTER

- Exécution d’une portion de code par la tâche maître
- Pas de synchronisation, ni en début ni en fin (contrairement à SINGLE)
- Pas de clause

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

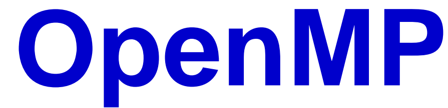
Partage du travail : procédures orphelines

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

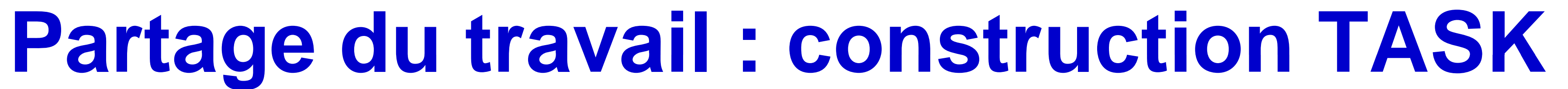
  Elles sont dites orphelines
- Attention : le contexte d'exécution peut être # selon le mode de compilation des unités de programme appelantes et appelées
- Attention aux variables locales aux subroutines (par défaut (-qsave, elles sont globales)

```
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
```



- Introduction
- Structure d'OpenMP
- Portée des données
- Constructions de partage du travail
  - → Construction TASK
- Synchronisation
- Performances
- Conclusion



Partage du travail : construction TASK

- Une “TASK” au sens OpenMP est une unité de travail dont l’exécution peut être différée. Elle peut aussi être exécutée immédiatement

- Permet de paralléliser des problèmes irréguliers
  - boucles non bornées
  - algos récursifs
  - schémas producteur/consommateur

- Une TASK est composée de :
  - un code à exécuter
  - un environnement de données initialisé à sa création
  - des variables de contrôle interne

.

Partage du travail : construction TASK

- La construction TASK définit explicitement une TASK OpenMP
- SI une thread rencontre une construction TASK, une nouvelle instance de la TASK est créé (paquet code + données associées)
- La THREAD peut soit exécuter la tâche, soit différer son exécution. La tâche pourra être attribuée à n'importe quelle THREAD de l'équipe.

```
!$OMP PARALLEL
 call sub()
!$OMP TASK

          …

!$OMP END TASK

          …
!$OMP END PARALLEL
```

Clauses :

IF, DEFAULT(PRIVATE|SHARED)
PRIVATE, SHARED
FIRSTPRIVATE
UNTIED

Partage du travail : construction TASK

- Par défaut les variables de TASKs orphelines sont firstprivate
- Sinon les variables sont firstprivate à moins qu'elles héritent d'un attribut shared du contexte qui les englobe

```
int fib (int n) {
        int x, y;
        if (n < 2) return n;
#pragma omp task shared(x)
        x = fib(n-1);
#pragma omp task  shared(y)
        y = fib(n-2);
#pragma omp taskwait
        return x+y;
}
```

Ici n est firstprivate

Partage du travail : construction TASK

Remarque : le concept existait avant la construction:

Une thread qui rencontre une construction PARALLEL

- crée un ensemble de TASKs implicites (paquet code+données)
- Crée une équipe de threads
- Les TASKs implicites sont lées (tied) aux threads, une pour chaque thread de l'équipe

- A quel moment la TASK est-elle exécutée ?

  Certaines construction ont des points de « scheduling »

  L'exécution d'une TASK générée peut être affectée à une thread lorsqu'elle atteint un point de « scheduling » de tâches , càd

  - Le point qui suit immédiatement la génération explicite de la TASK
  - après la dernière instruction d'une région de TASK
  - Dans une région de BARRIER implicite ou explicite
  - Dans une région de taskwait

Partage du travail : construction TASK

Création de N TASK *func* autant que de threads

Ici, la terminaison de toutes les TASKs *func* est garantie

Création d'une TASK *func1*

Ici, la terminaison de la TASK *func1* est garantie

```
!$OMP PARALLEL
 !$OMP TASK
        func()
!$OMP END TASK
 !$OMP BARRIER
!$OMP SINGLE
        !$OMP TASK
              func1()
        !$OMP END TASK
!$OMP END SINGLE
!$OMP END PARALLEL
```

OpenMP

- Introduction
- Structure d’OpenMP
- Portée des données
- Constructions de partage du travail
- Construction task

  ➡ Synchronisation
- Performances
- Conclusion

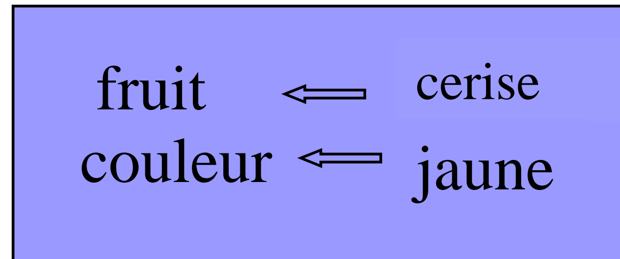
Synchronisation

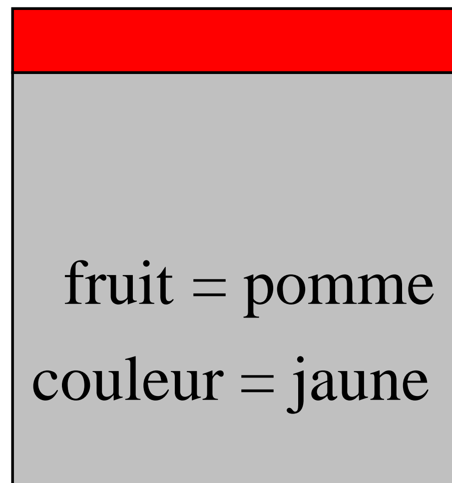
Il peut être nécessaire d'introduire une synchronisation entre tâches concurrentes pour éviter que celles-ci modifient la valeur d'une variable dans un ordre quelconque

Ex :

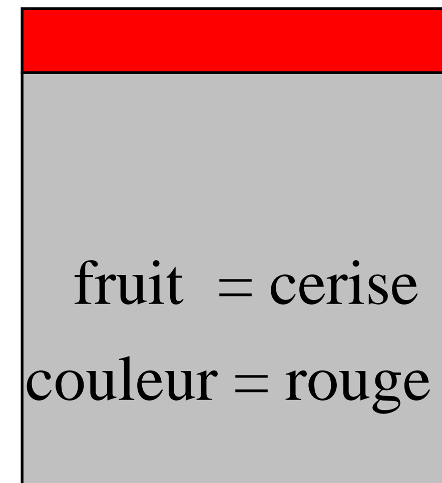
2 threads ont un espace de mémoire partagé

Espace partagé





Thread 1



Thread 2

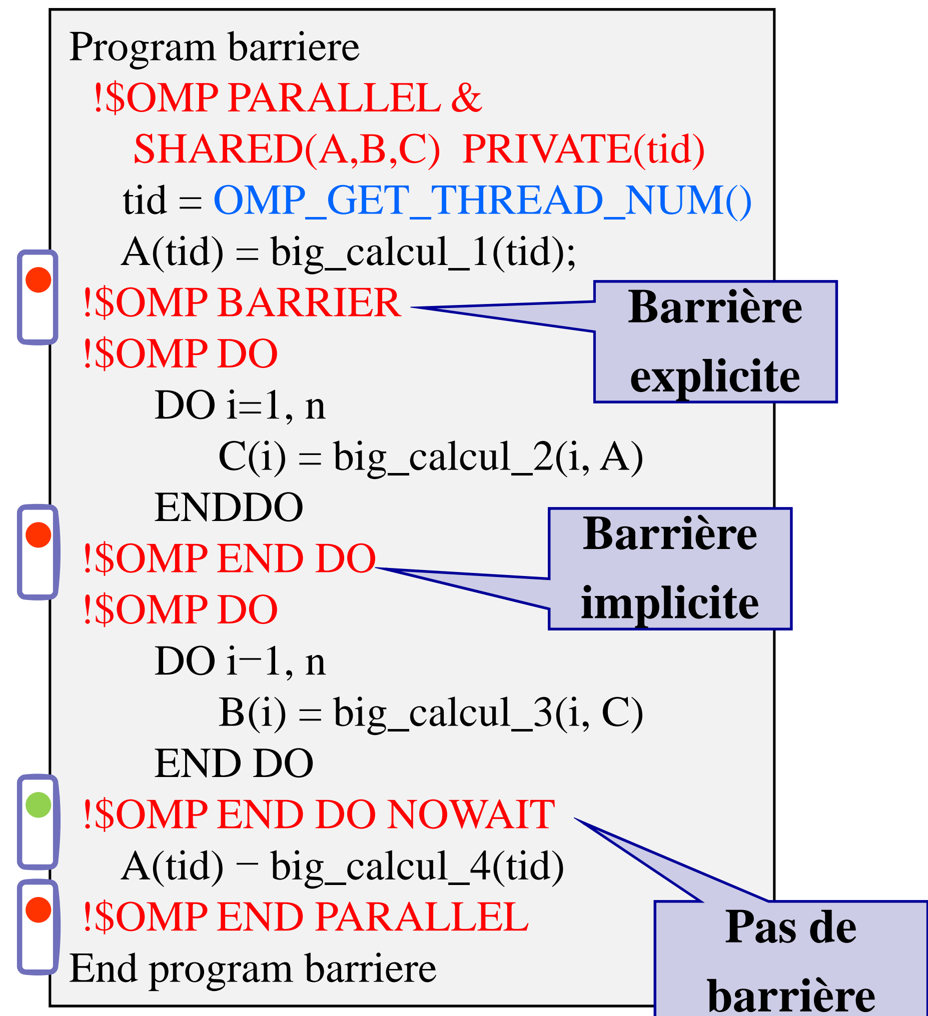
Synchronisation

- 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

Garantit l'accès en exclusion mutuelle

Son étendue est dynamique

```
Integer,dimension(m,n) :: a=1,b=0
!$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
```

Synchronisation : exclusion mutuelle mise à jour atomique

La directive ATOMIC s'applique seulement dans le cadre de la mise à jour d'un emplacement 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)**

op : +, -, *, /, .AND., .OR., .EQV., .NEQV.

f : MAX, MIN, IAND, IOR, IEOR

```
Program atomic
  implicit none
  integer :: count, rang
  integer :: 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
```

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
  integer :: OMP_GET_NUM_THREADS
  integer :: 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
```

Performances et 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) , attention aux options par défaut du compilateur
- 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 et partage du travail

- Minimiser le nombre de régions parallèles
- Eviter de sortir d'un région parallèle pour la recréer immédiatement

```
!$OMP PARALLEL
   ...
   !$OMP DO
      do i=1, N
          func1(i)
      end do
   !$OMP END DO

!$OMP END PARALLEL

!$OMP PARALLEL

   !$OMP DO
      do j=1, M
          func2(j)
      end do
   !$OMP END DO

!$OMP END PARALLEL
```

Surcoût intutile

Ici, une construction PARALLEL suffit

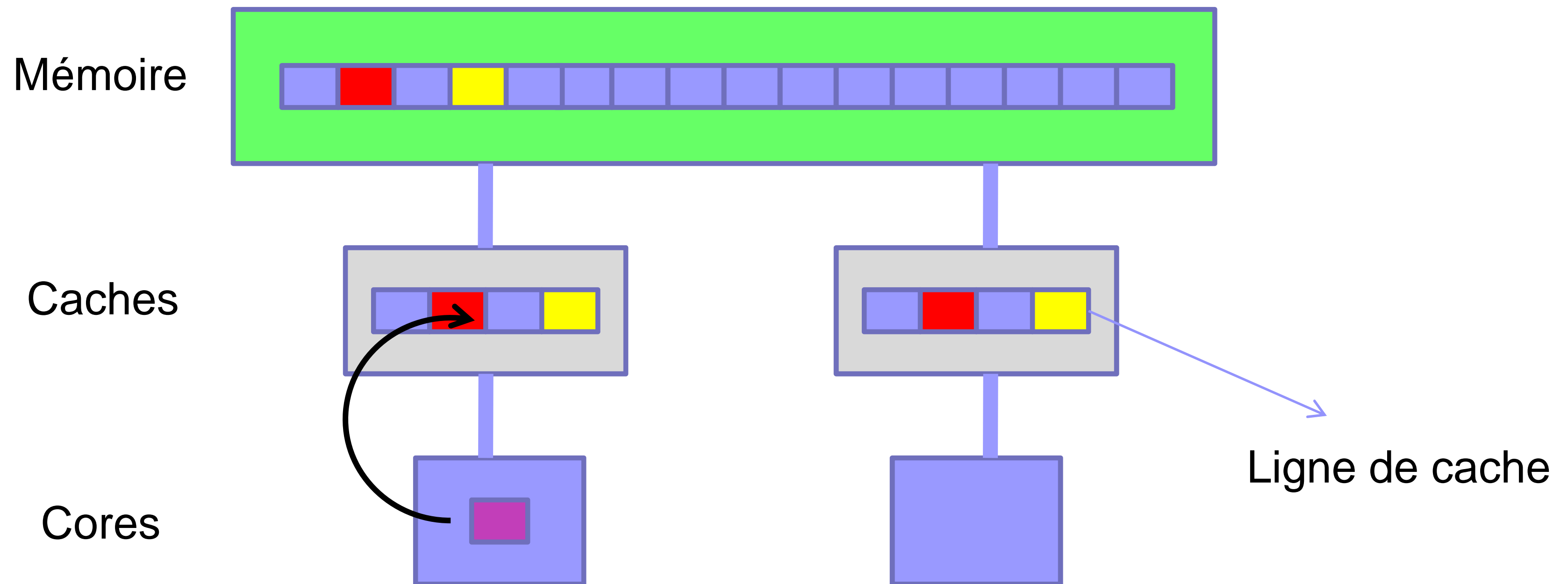
```
!$OMP PARALLEL
   ...
   !$OMP DO
      do i=1, N
          func1(i)
      end do
   !$OMP END DO

   !$OMP DO
      do j=1, M
          func2(j)
      end do
   !$OMP END DO

!$OMP ENDPARALLEL
```

Performances et partage du travail

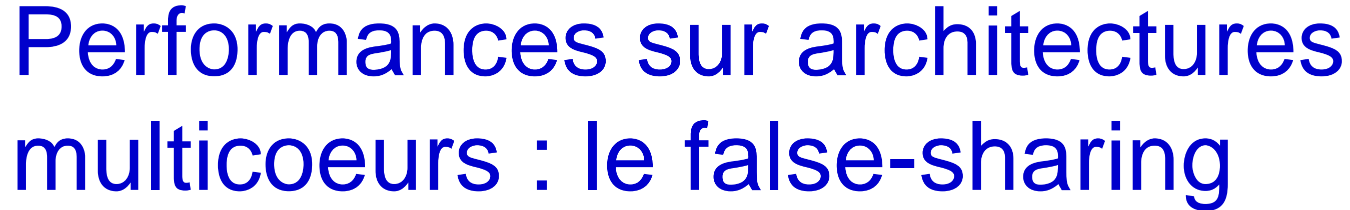
- Introduire un PARALLEL DO dans les boucles capables d'exécuter des itérations en //
- Si il y a des dépendances entre itérations, essayer de les supprimer en modifiant l'algorithme
- S'il reste des itérations dépendantes, introduire des constructions CRITICAL autour des variables concernées par les dépendances.
- Si possible, regrouper dans une unique région parallèle plusieurs structures DO.
- Dans la mesure du possible, paralléliser la boucle la plus externe
- 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(RUNTIME)
- ATOMIC et REDUCTION plus performants que CRITICAL

Performances : les effets du False Sharing

La mise en cohérence des caches et les effets négatifs du « false sharing » peuvent avoir un fort impact sur les performances



Une opération de chargement d'une ligne de cache partagée invalide les autres copies de cette ligne.

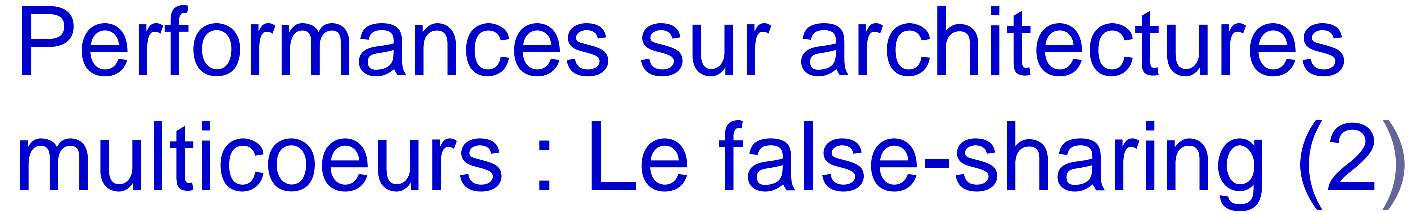


Performances sur architectures multicoeurs : le false-sharing

- L'utilisation des structures en mémoire partagée peut induire une diminution de performance et une forte limitation de la scalabilité.

- **Pourquoi ?**
  - Perfomances ➔ utilisation du cache
  - Si plusieurs processeurs manipulent des données différentes mais adjacentes en mémoire, la mise à jour d'éléments individuels peut provoquer un chargement complet d'une ligne de cache, 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 sur plusieurs cores
  - Plusieurs threads, sur # cores mettent à jour des données qui se trouvent dans la même ligne de cache.
  - Ces mises à jour ont lieu très fréquemment et simultanément



Performances sur architectures multicoeurs : 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 :**
  - En privatisant éventuellement des variables
  - Parfois en augmentant la taille des tableaux (taille des problèmes ou augmentation artificielle) ou en faisant du « padding »
  - Parfois en modifiant la façon dont les itérations d'une boucle sont partagées entre les threads (augmenter la taille du chunk)

Performances sur architectures multicoeurs : Le false-sharing (3)

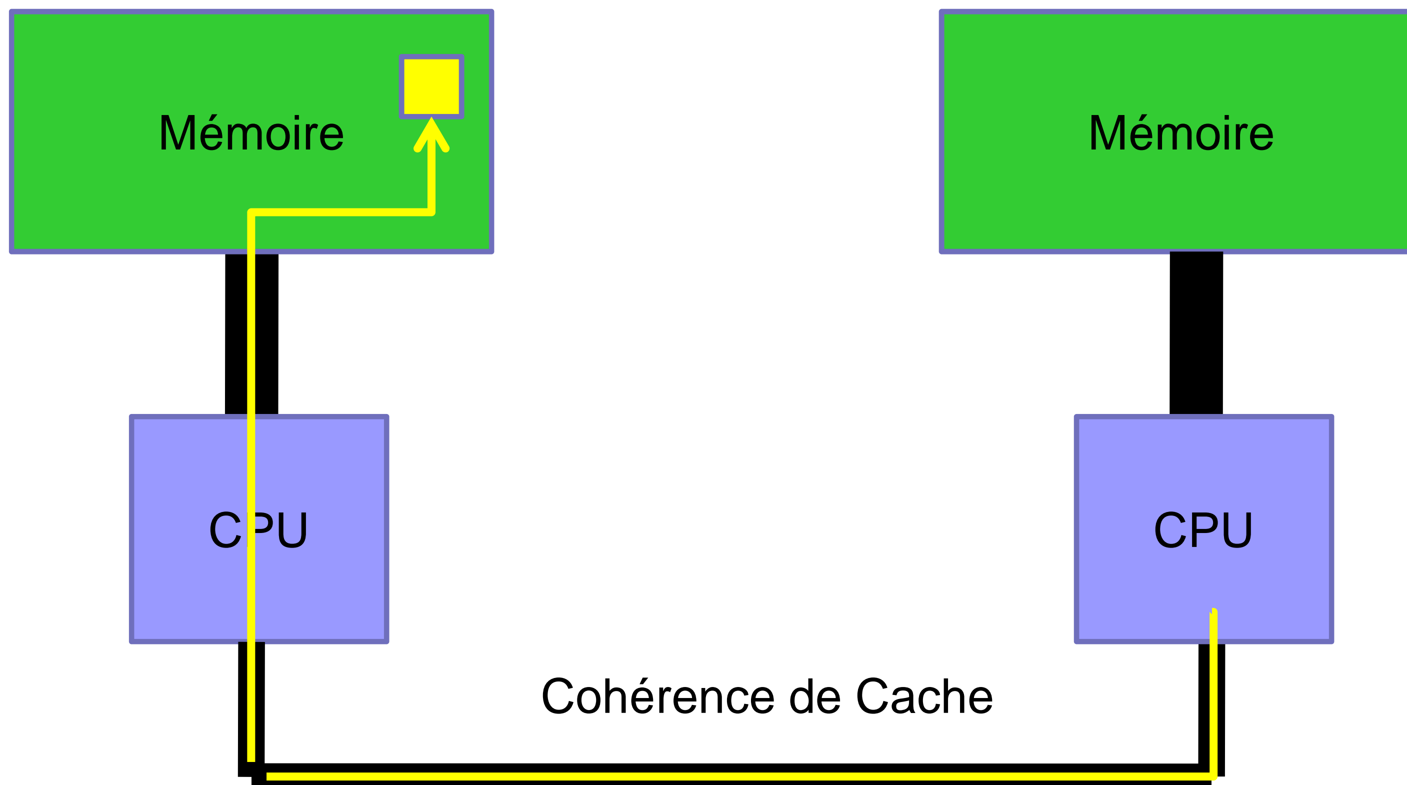
```
#pragma omp parallel for shared (Nthreads,a) schedule(static,1)
    for (i=0; i<Nthreads; i++)
            a[i] += i;
```

Nthreads : nombre de thread exécutant la boucle
Supposons que chaque thread possède un copie de a dans son cache local
La taille de paquet de 1 provoque un phénomène de false sharing à chaque mise à jour

Si une ligne de cache peut contenir **C** éléments du vecteur **a**,, on peut résoudre le problème en étendant artificiellement les dimensions du tableau (« array padding ») : on déclare un tableau **a[n][C]** et on remplace **a[i]** par **a[i][0]**

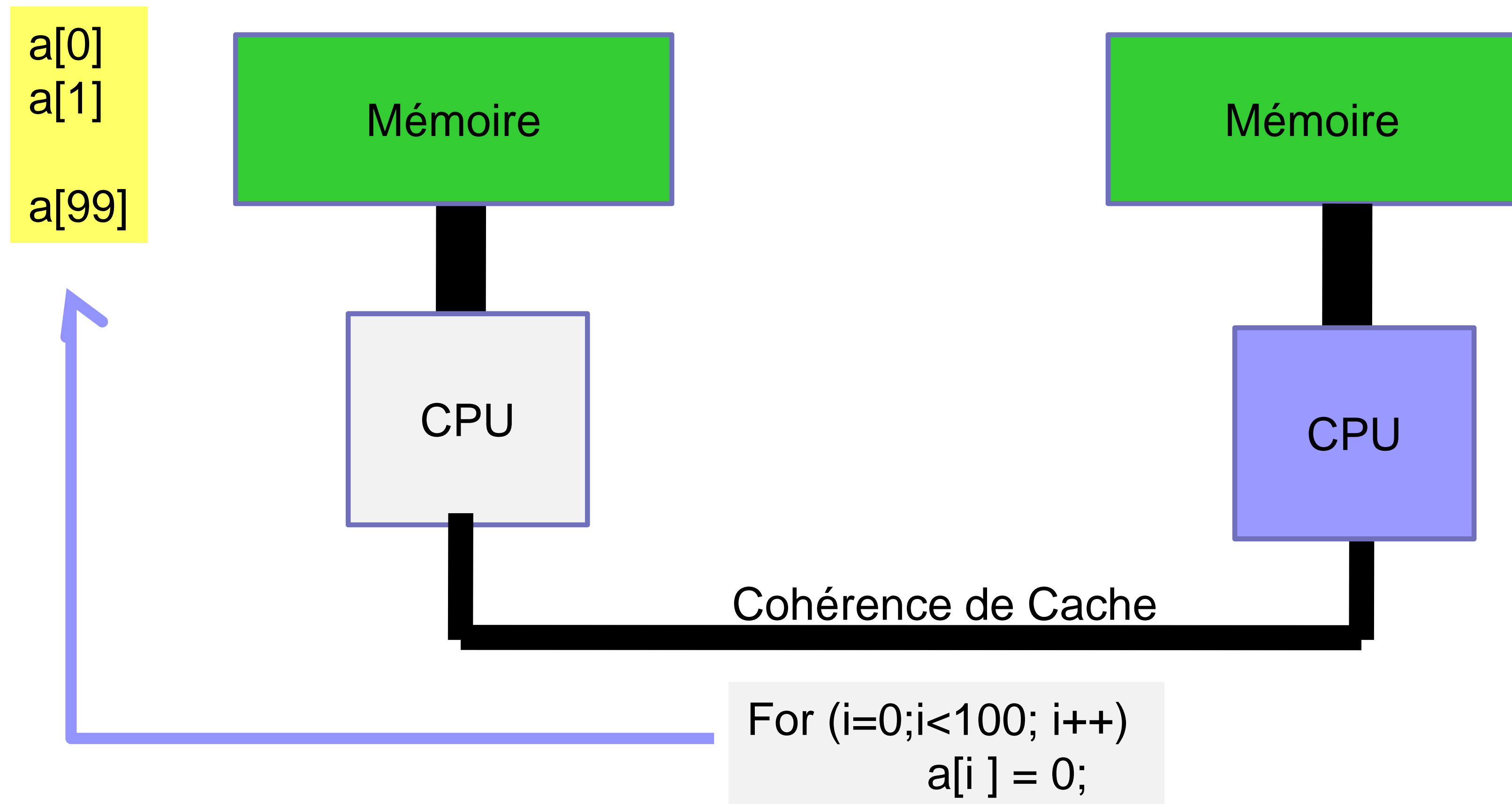
Performances : effets du CC-NUMA

- Architecture CC-NUMA

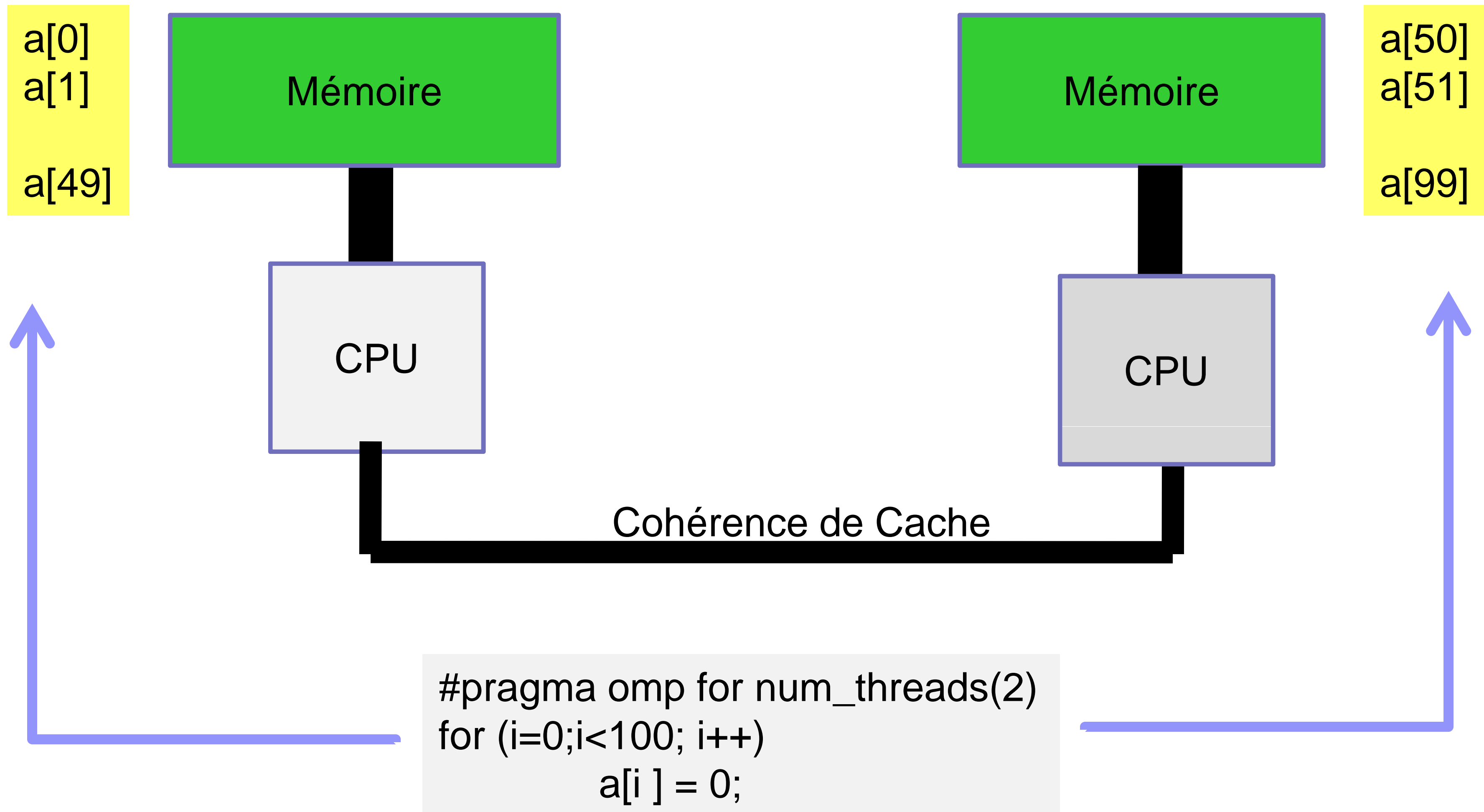


Performances : effets du CC-NUMA

- Comment placer les données pour optimiser les temps d'accès ?
- La politique de placement dépend de l'OS
- Sous linux, solaris : « first touch »



Effet de la politique « First Touch »



Performances sur architectures multicoeurs : ordonnancer les threads efficacement

- Maximiser le rendement de chaque CPU fortement dépendant de la localité des accès mémoire
- Les variables sont stockées dans une zone mémoire (et cache) au moment de leur initialisation.
  - parralléliser l'initialisation des éléments (Ex tableau) de la même façon (même mode même chunk) que le travail
- Minimiser le surcoût de l'ordonnanceur
- L'idéal serait de pouvoir spécifier des contraintes de placement des threads en fonction de affinités threads/mémoire

Performances sur architectures multicoeurs : problématique de la diversité des architectures

types de processeurs
nombres de cœurs
niveaux de cache
architecture mémoire

Et # types d’interconnexion des différents composants

Des stratégies d’optimisation qui peuvent différer d’une configuration à l’autre

Pour l'instant, OpenMP ne fournit pas de support

Pas de spécification, mais des discussions en cours.

En attendant :

utiliser les possibilités offertes par l'OS et des appels dépendant du compilateur pour attacher les threads à des cores et pour contrôler l'allocation de page.

Possibilité de lier les threads OpenMP à des cores

Notion de *thread affinity* : le lieu d’exécution de certaines threads est restreint à un sous ensemble d’unités d’exécution physiques

Avec la librairie intel :

Variable d’environnement KMP_AFFINITY

Sur SGI : notion de cpuset

outil « dplace » permet de lier une application à un ensemble de CPUs

Outil « taskset » sous linux

Performances : recouvrement des IO par le calcul en exploitant le parallélisme imbriqué

```
#pragma omp parallel sections
{
   #pragma omp section
   for (i=0; i<n; i++) {
      read_input(i);
      signal_read(i);
   }
   #pragma omp section
   for (i=0; i<n; i++) {
         wait_read(i);
         process_data(i)
         signal_processed(i);
   }
   #pragma omp section
   for (i=0; i<n; i++) {
         wait_processed(i);
         write_output(i);
   }
```

```
void processed_data(i)
{
 …
#pragma omp parallel for num_threads(4)
      for (j=0; j<m; j++) {
         do_compute(I, j);
      }
}
```

Example d'utilisation du parallélisme fonctionnel pour :

- lire les données
- effectuer les calculs
- écrire les résultats

Un total de 6 threads sera utilisé.

Performances : parallélisation conditionnelle

Utiliser la clause IF pour mettre en place une parallélisation conditionnelle

ex : ne paralléliser une boucle que si sa taille est suffisamment grande

```
Program parallel
  implicit none
  integer, parameter :: n=8192
  integer :: i,j
real, dimension(n,n) :: a,b
call random_number(a)
!$OMP PARALLEL DO SCHEDULE(RUNTIME) &
&!$OMP IF(n.gt.1024)
  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
```

Fonctions de bibliothèque

Fonctions relatives aux verrous

Un verrou est libre ou possédé par une thread.

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

« omp_test_lock() » permet d'attendre à un point du code la libération d'un verrou par une autre thread.

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()
- 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()

Mesure du temps elapsed (temps de restitution) propre à chaque thread : OMP_GET_WTIME()

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 bibliothèques de routines 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 à mémoire partagée

- Choix fortement dépendant : de la machine, du code, du temps que veut y consacrer le développeur et du gain recherché.

- Extensibilité supérieure avec MPI

- http://www.openmp.org
- http://www.openmp.org/mp-documents/spec30.pdf
- http://www.idris.fr
- http://ci-tutor.ncsa.illinois.edu/login.php
- « Using OpenMP , Portable Shared Memory Model », Barbara Chapman