Introduction à OpenMP

Kamel Mazouzi

Mésocentre de calcul de Franche-Comté

Plan

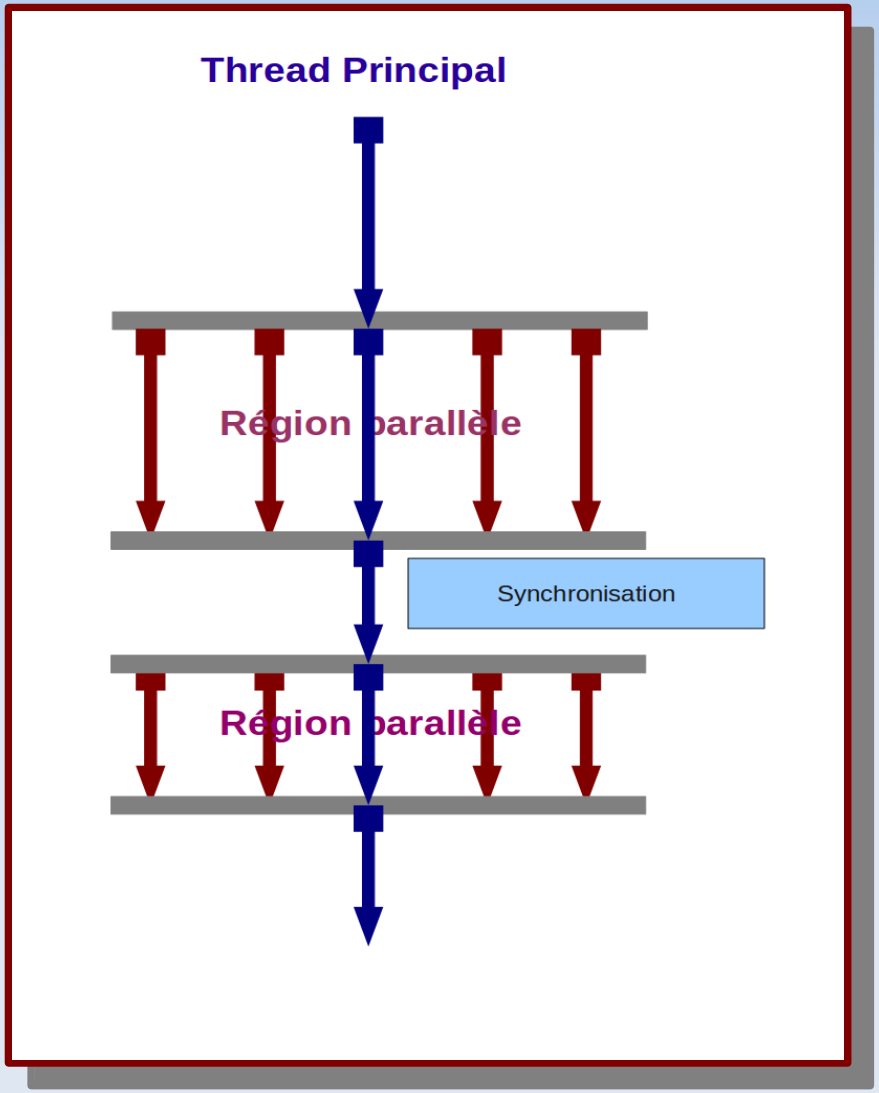
- Introduction
- Région parallèle
- Partage de travail
- Synchronisation
- Dépendance de données
- Performances

OpenMP

- Une API pour écrire des applications multithreadés sur des architectures à mémoire partagée
- Ensemble de directives de compilation et une bibliothèque de routines
- Relativement simple pour développer des applications parallèles en Fortran, C/C++
- Standard pour les architectures SMP

OpenMP : Modèle d'exécution

- Basé sur le modèle **fork/Join**
- Le master crée des threads à l'entrée de la région parallèle
- Les threads fils exécutent un bloc d'instructions en parallèle
- À la sortie de la région parallèle, seul le thread master poursuit son exécution

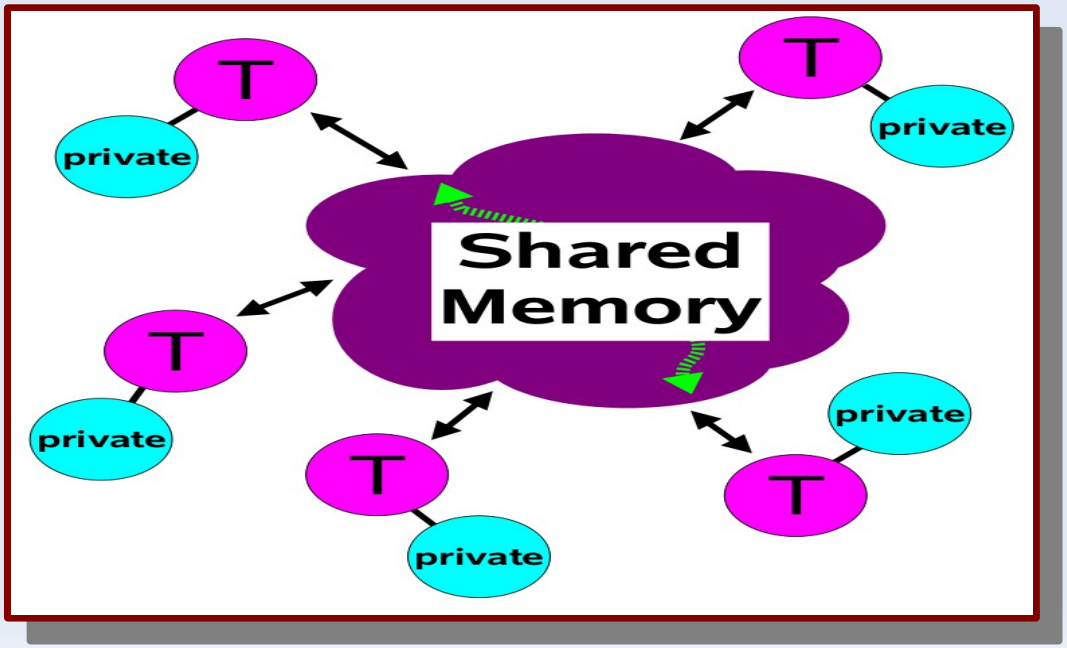


OpenMPI : Principes

- parallélisation incrémentale
- peu de code supplémentaire
- limité au parallélisme présent dans le code initial
- Le programmeur insère des **directives** OpenMP dans le code source
- Le compilateur interprète ces directives et appel des **routines** OpenMP pour paralléliser une partie du code
- La gestion de la synchronisation et des dépendances des données est à la charge du programmeur !

```
Initialisation();
  for (i=0; i<N; i++)
     Calcul(i);
Autre_calcul();
```

```
Initialisation();
#pragma omp parallel for private(i)
  for (i=0; i<N; i++)
     Calcul(i);
Autre_calcul();
```



Modèle d'exécution

- Le nombre de threads peut être contrôlé à partir du programme ou en utilisant la variable d'environnement **OMP_NUM_THREADS**

  export OMP_NUM_THREADS=4

- Compilation :

C/CC++

```
icc -openmp omp_pgm.c -o pgm
gcc -fopenmp omp_pgm.c -o pgm
```

Fortran

```
ifort -openmp omp_pgm.f -o pgm
gfortran -fopenmp omp_pgm.f -o pgm
```

Région parallèle

Région parallèle

Un bloc de code exécuté par plusieurs threads en parallèle

- Fortran :

```
!$OMP PARALLEL [ clause [ [ , ] clause ] ... ]
  structured-block
!$OMP END PARALLEL
```

- C/C++ :

```
#pragma omp parallel [ clause [ clause ]...]
    structured-block
```

Région parallèle : clauses

if	(scalar expression)
private	(list)
shared	(list)
default	(none|shared|private)
reduction	(operator: list)
firstprivate	(list)
num_threads	(scalar_int_expr)

Région parallèle : principe

- Dans une région parallèle, par défaut, le statut des variables est **partagé (`shared`)**
- Au sein d'une même région parallèle, tous les threads exécutent le même code
- Il existe une barrière implicite de synchronisation en fin de région parallèle
- Il est interdit d'effectuer des « branchement » (ex. **`GOTO`**, **`CYCLE`**, …) vers l'intérieur ou vers l'extérieur d'une région parallèle

Principe : construction d'une région parallèle

- Chaque thread a un identifiant unique **[0, N-1]** (N : le nombre total de thread)
  - Cet ID est obtenue en appelant la fonction :

```
omp_get_thread_num()
```

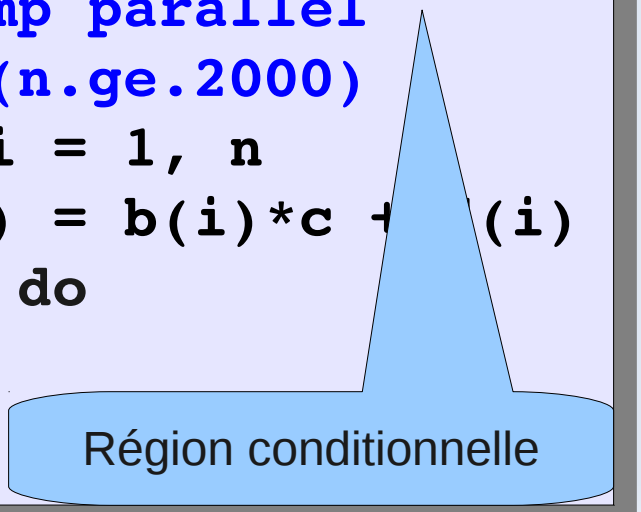
- Le nombre de total de thread depuis le programme :

```
omp_get_num_threads()
```

Région parallèle : Exemple (1)

```
program hello
!$OMP PARALLEL
print *,'hello world'
!$OMP END PARALLEL
stop
end program hello
```

```
!$OMP PARALLEL
 ...
 id=sqrt(x)
 if(conv(res)) goto 20
!$OMP END PARALLEL
20 print *, id
```



```
!$omp parallel
 if(n.ge.2000)
do i = 1, n
a(i) = b(i)*c +  (i)
end do
```

Région conditionnelle

```
int myId=-1;
#pragma omp parallel private(i)
    {
     myid = omp_get_thread_num();
     if (myid == 0)
      do_something();
      else
       do_something_else(myid);
    }
```

Région parallèle : Exemple (2)

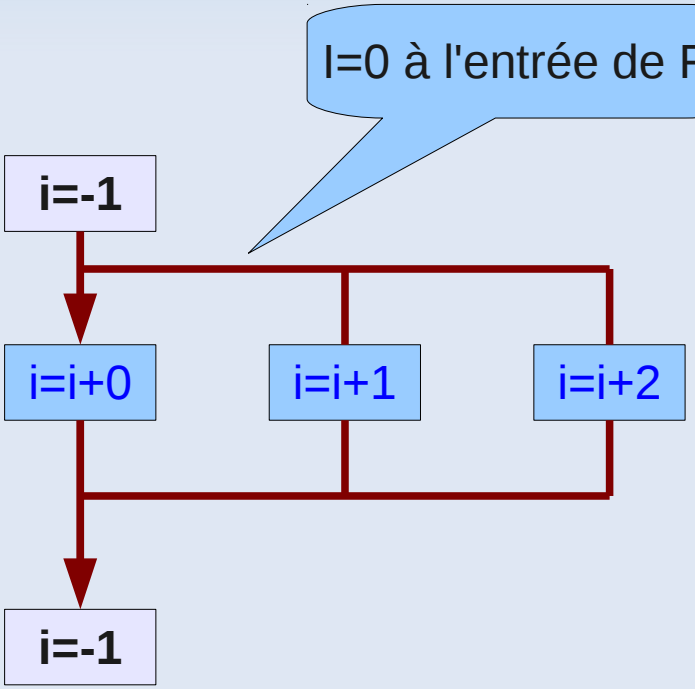
```
#include <stdio.h>
#include <omp.h>
int main(int argc, char** argv)
{  int i=-1;
  #pragma omp parallel private (i)
    {
     i=omp_get_thread_num();
     printf( "hello world from %d",i);
    }
  return 0;
}
```

Région parallèle

```
$./hello
hello world from 1
hello world from 0
hello world from 2
```

Région parallèle : portée de variables

private (list) déclare une liste de variables privées à chaque thread.




```
i=-1
!$OMP PARALLEL PRIVATE(i)
 i=i+OMP_GET_THREAD_NUM()
!$OMP END PARALLEL
 print *,'i'
```

```
i=-1;
#pragma omp parallel \
  private(i) {
 i=i+omp_get_thread_num();
 }
 printf("%d",i);
```

Région parallèle : portée de Variables

firstprivate(list)

- comme **private**
- force l'initialisation des variables privées à leur dernière valeur avant l'entrée dans la région parallèle

```
i=200
!$OMP PARALLEL FISRTPRIVATE(i)
 i=i+OMP_GET_THREAD_NUM()
  print *,"i=", i
!$OMP END PARALLEL
```

```
i=200;
#pragma omp parallel \
   firstprivate(i) {
   printf("i=%d",i);
  i=i+omp_get_thread_num();
  }
```

```
i=200
i=201
i=202
```

Région parallèle : portée de Variables

Default(none|private|shared)

Change le statut par défaut des variables dans une région parallèle

```
 real ::a
 a=2010.
!$OMP PARALLEL DEFAULT(PRIVATE)
  a=a+10.
  print *,"a interne=", a
!$OMP END PARALLEL
 print *,"a externe=", a
```

```
$export OM_NUM_THREAD=2
$a.out

a interne=10
a interne=10
a externe=2010
```

Région parallèle : appel de procédure

Toutes les variables transmises par arguments héritent du statut défini dans la région

```
$export OM_NUM_THREAD=3
$a.out
b = 2010
b = 2011
b = 2012
```

```
 integer ::a,b
 a=2010
!$OMP PARALLEL SHARED(a)
!$OMP PRIVATE(b)
  Call sub(a,b)
  print *,"b=", b
!$OMP END PARALLEL
subroutine sub(x,y)
 integer ::x,y
 use OMP_LIB
 y=x+OMP_GET_THREAD_NUM
end subroutine sub
```

Région parallèle : compléments

- La clause **REDUCTION** est utilisée pour les opérations de réduction avec synchronisation implicite entre les tâches (CF. boucle parallèle)
- La clause **NUM_THREAD** permet de spécifier le nombre de threads à l'entrée d'une région parallèle. Il s'agit de l'équivalent de l'appel de fonction **OMP_SET_NUM_THREADS**
- Le nombre de threads peut être différents d'une région parallèle à l'autre. Ce mode peut être activé en utilisant :
  - **CALL OMP_SET_DYNAMIC**
  - **OMP_DYNAMIC=true**

```
!$OMP PARALLEL NUM_THREAD(2)
   print *,"Hello!"
!$OMP END PARALLEL

!$OMP PARALLEL NUM_THREAD(3)
   print *,"Bonjour !"
!$OMP END PARALLEL
```

```
export NUM_THREAD=4
export OMP_DYNAMIC=true
```

```
$ Hello!
$ Hello!
$ Bonjour !
$ Bonjour !
$ Bonjour !
```

Région parallèle : compléments

Il est possible d'imbriquer (*nesting*) des régions parallèles. Pour ce faire il faut activer ce mode :

- `CALL OMP_SET_NESTED`
- `export OMP_NESTED=true`

```
export OMP_NESTED=true
export OMP_DYNAMIC=true
```

```
!$OMP PARALLEL NUM_THREAD(2)&
!$OMP PRIVATE(id)
  id=OMP_GET_THREAD_NUM;
  print *,"id region 1=",id
!$OMP PARALLEL NUM_THREAD(1)&
!$OMP_PRIVATE(id)
  id=OMP_GET_THREAD_NUM;
  print *,"id region 2=",id
!$OMP END PARALLEL
!$OMP END PARALLEL
```

```
$ id region1 = 0
$   id region2 = 0
$ id region1 = 1
$   id region2 = 0
```

Partage du travail

Partage du travail : principes

- Divise l'exécution d'une région parallèle sur l'ensemble de threads (partage du travail)
- Les directives (worksharing) doivent être incluses dans une région parallèle
  - Pas de nouveaux threads à l'entrée de la région
  - Pas de barrière implicite à l'entrée de la région

Les directives (worksharing) doivent être appliquées par tous les membres d'une équipe ou aucun

Partage du travail : les directives

Boucle parallèle

```
#pragma omp
for
{
    ....
}
```

```
!$OMP DO
    ....
!$OMP END DO
```

Section parallèle

```
#pragma omp
 section
{
   ....
}
```

```
!$OMP SECTIONS
      ....
!$OMP END SECTIONS
```

Exécution exclusive

```
#pragma omp
 single
 {
   ....
 }
```

```
!$OMP SINGLE
    ....
!$OMP END SINGLE
```

Boucles parallèles

Boucle parallèle (DO / FOR)

- Distribution des itérations d'une boucle sur l'ensemble de threads
- Pas de synchronisation à l'entrée de la boucle
- Un thread attend la fin d'exécution de tous les autres threads pour continuer
- Un thread utilise la clause **`nowait`** pour continuer l'exécution sans attendre
- Il est possible d'introduire plusieurs constructions **`DO/FOR`** dans une même région parallèle

```
#pragma omp parallel
 shared(a,b) private(j)
 {
 #pragma omp for
 for (j=0; j<N; j++)
  a[j] = a[j] + b[j];
 }
```

Boucle parallèle : syntaxe

- Fortran

```
!$omp do[clause [clause]…]
 do loop
[!$omp end do [nowait]]
```

```
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
ordered
schedule(kind [, chunk_size])
```

Clauses disponibles

Boucle parallèle : syntaxe

- C/C++

```
#pragma omp for[clause [clause]…]
 for loop
```

```
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
ordered
schedule(kind [, chunk_size])
nowait
```

Clauses disponibles

Clause Schedule

La clause **`schedule(type,[chunk])`** permet de spécifier le mode de répartition des itérations sur l'ensemble de threads :

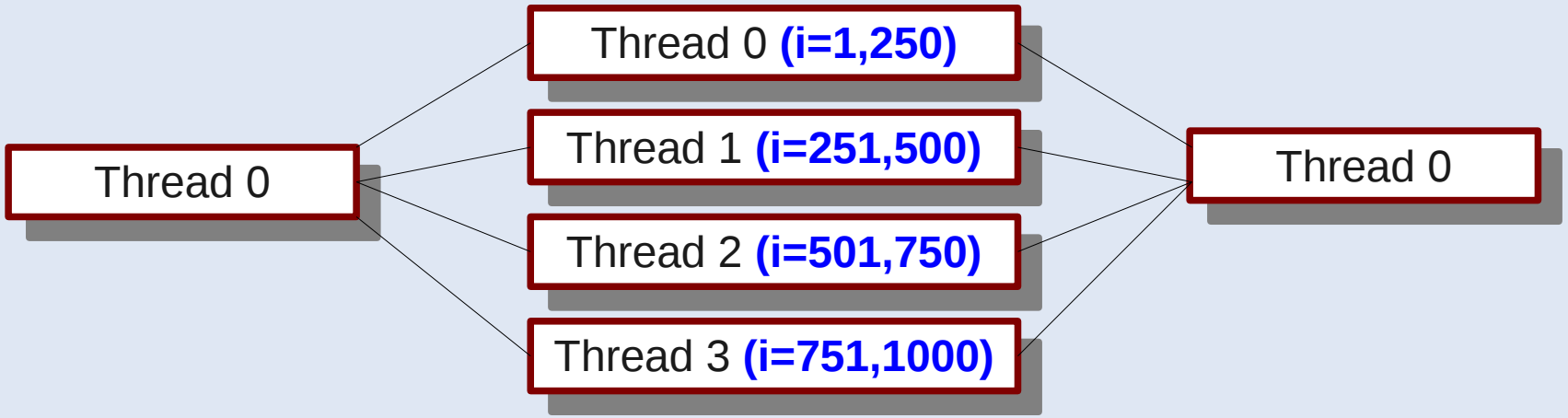
- « **`chunk`** » est la taille de chaque bloc à traiter (nombre d'itérations)
- « **`type`** » mode de distribution :
  - **`static`**
  - **`dynamic`**
  - **`guided`**
  - **`Runtime`**

Le mode de répartition est un atout majeur pour l'équilibrage de la charge sur l'ensemble des processeurs

schedule(static)

- Les itérations sont réparties de manière équitable entre les threads

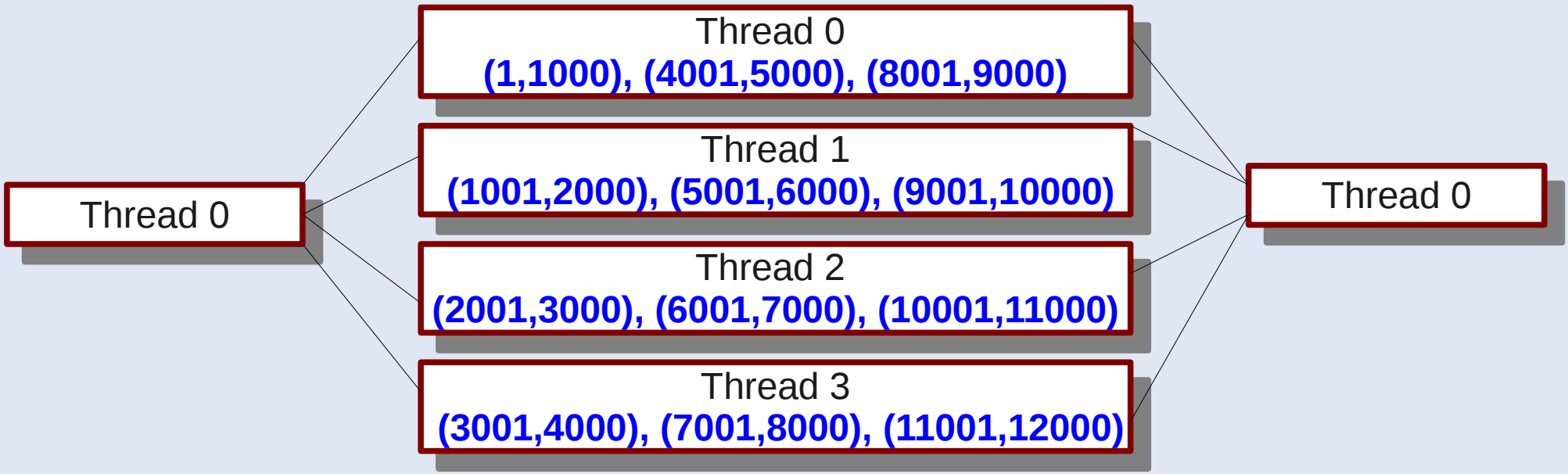
```
!$omp do shared(x) private(i)&
!$omp schedule(static)
 do i = 1, 1000
   x(i)=a
end do
```



schedule(static,chunk)

- Les itérations sont divisées en paquets de taille «**chunk**»
- Les paquets sont attribués aux threads d'une manière cyclique

```
!$omp do shared(x) private(i)&
!$omp schedule(static,1000)
 do i = 1, 12000
   x(i)=a
end do
```



schedule(dynamic,chunk)

- Les itérations sont divisées en paquets de taille donnée
- Dès qu'un thread termine ses itérations un autre paquet lui est attribué
- La valeur par défaut de «**chunk**» est 1
- **Technique coûteuse**
- **Meilleure équilibrage de charge**

```
!$omp do shared(x) &
!$omp private(i)&
!$omp do schedule(dynamic)
 do i = 1, 12000
   x(i)=a
$omp end do
```

schedule(guided,chunk)

- Comme le mode `dynamic`, mais «**`chunk`**» décroît exponentiellement
- **Moindre surcoût**
- **Meilleure équilibrage de charge**
- **Assure que tous les threads exécutent un travail à un instant donné**

```
#pragma omp parallel \
 private(i), shared(x)
 #pragma for schedule
   (guided,10)
  for (i=0;i<1000;i++)
   x(i)=f(i);
```

schedule (runtime)

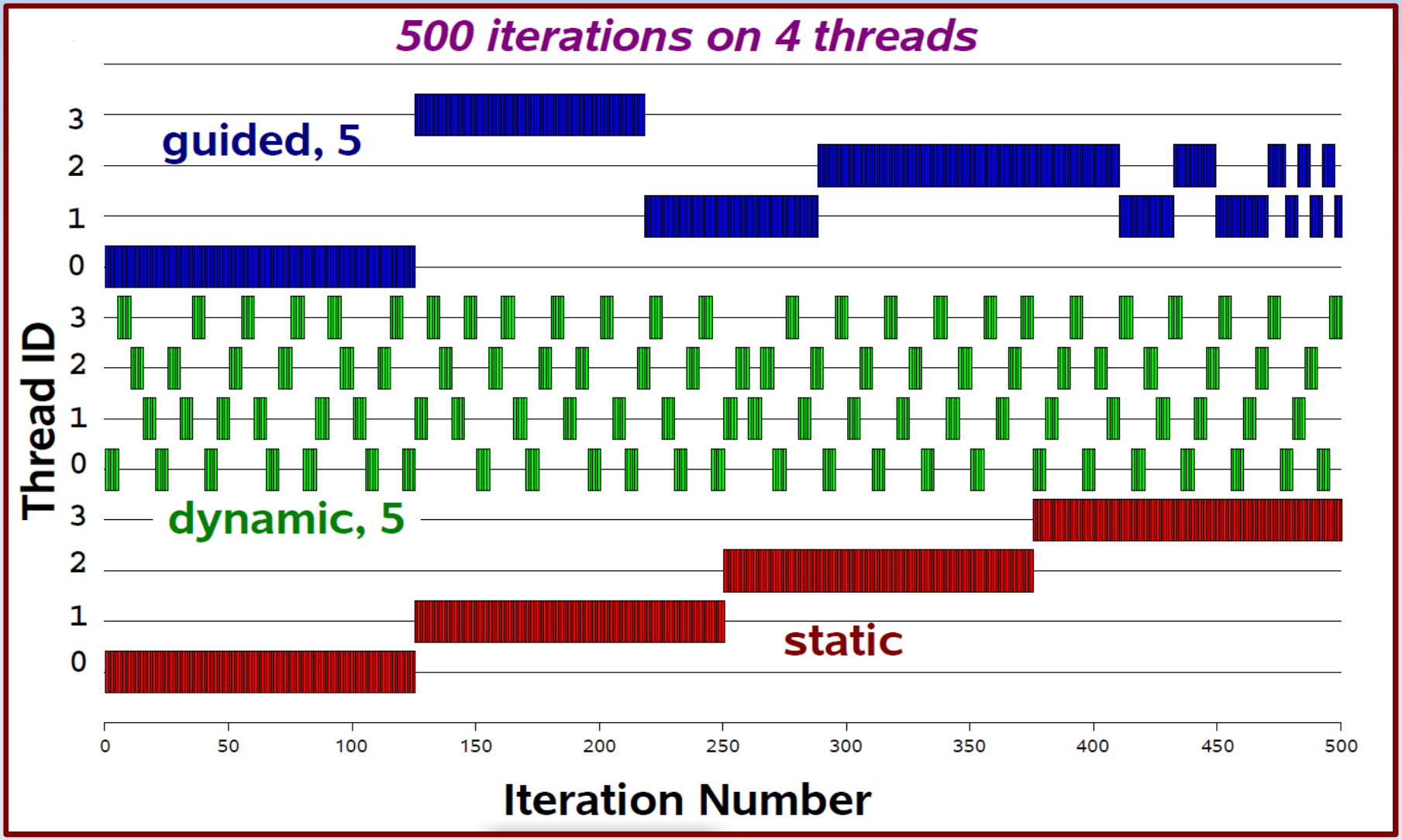
- Le mode de répartition est déterminé à l'exécution
- Se base sur la valeur de

  la variable d'environnement **OMP_SCHEDULE**
- Le mode par défaut est **static**
- Utile pour tester les applications sans recompilation

```
$ export OMP_SCHEDULE static,1000
$ export OMP_SCHEDULE runtime
$ export OMP_SCHEDULE guided,55
```

```
!$omp do shared(x) &
!$omp private(i)&
!$omp schedule(runtime)
 do i = 1, 12000
   x(i)=a
end do
```

Exemple


500 iterations on 4 threads
guided, 5
dynamic, 5
static
Thread ID
Iteration Number
0
50
100
150
200
250
300
350
400
450
500

La clause lastprivate

- Comme **`private`** pour les régions parallèle : chaque thread possède une copie de la variable
- Après la boucle, la valeur de la variable contient la valeur du dernier calcul de l'itération

```
i=200
a=10;
b=50;
!$OMP PARALLEL PRIVATE(i,b)&
!$OMP SHARED(a)

      $OMP DO LASTPRIVAE(b)&
      $ PRIVATE (i)
         DO i = 1, 12000
          b=b*i;
         END DO

a=b;
!$OMP END PARALLEL
```

« a » contient la dernière valeur calculée de « b »

Réduction

```
int some=0;
int i;
#pragma for parallel \
 private(i) shared(some)
    for (i=0;i<1000;i++)
     some=some+i;
```

```
int some=0;
int i;
#pragma for parallel \
 private(i) reduction(+:some)
    for (i=0;i<1000;i++)
     some=some+i;
```

Réduction

- Opération associative appliquée à une variable partagée

```
reduction(operator|intrinsic:var1[,var2])
```

- Les variable doivent être partagées

Fortran

- opérateurs : **+, *, -, .and., .or., .eqv., .neqv.**
- intrinsic : **max, min,iand,ior, ieor**

C/C++

- opérateurs : **+, *, -, &, ^, |, &&, ||**
- Les pointeurs et les variables de référence ne sont pas autorisés

Réduction : exemple

```
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
   !$OMP END DO
 !$OMP END PARALLEL
 print *,"s =",s, "; p =",p, "; r =",r
end program parallel
```

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

Boucle parallèle : complément

- Une manière plus simple de créer une boucle parallèle est de combiner directives de région parallèle et de boucle parallèle

  Dans ce cas :

  - une synchronisation implicite est ajoutée à la fin de la boucle
  - la boucle n'admet pas la clause « **nowait** »

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

```
#pragma omp parallel for\
 lastprivate(temp)
  for (i=0; i<N; i++)
    temp =  i;
```

Boucle parallèle : complément

- La clause «**ORDERED**» permet d'exécuter une boucle de manière ordonnée (utile pour déboguer)
- L'ordre des itérations est identique à celui d'une exécution séquentielle

```
id : 0 ; iteration : 0
id : 1 ; iteration : 1
id : 2 ; iteration : 2
```

```
int N=3;
int id,i;
#pragma omp parallel \
default(none) private(id,i)
  {
    rang=omp_get_thread_num();
    #pragma omp for \
      schedule(runtime) ordered nowait
    for (i=0; i<N; i++) {
      #pragma omp ordered
      {
        printf("Id : %d ; \
         iteration : %d\n",id,i);
      }
    }
```

Partage de travail : WORKSHARE

- Valable uniquement pour Fortran (95)
- Utile pour répartir le travail lié à certaines constructions telles que :
  - Instructions ou blocs **FORALL** et **WHERE**
  - Instructions sur des variables scalaires
  - Instructions sur des variables de tableaux
  - Fonctions dites **ELEMENTAL** définies par l'utilisateur

```
!$OMP WORKSHARE
    structured block
!$OMP END WORKSHARE [NOWAIT]
```

```
REAL AA(N,N), BB(N,N), CC(N,N),&
 DD(N,N), FIRST, LAST

!$OMP WORKSHARE
      CC = AA * BB
      DD = AA + BB
      FIRST = CC(1,1) + DD(1,1)
      LAST = CC(N,N) + DD(N,N)
!$OMP END WORKSHARE NOWAIT
```

Sections parallèles

Construction : SECTIONS

- Une section est une portion de code exécutée par un et un seul thread
- Permet une décomposition fonctionnelle
- Une barrière implicite est ajoutée à la fin de la construction
- Accepte la clause «**nowait**»

```
!$omp parallel
!$omp sections

!$omp section
 call computeXpart()

!$omp section
 call computeYpart()

!$omp section
 call computeZpart()

!$omp end sections
!$omp end parallel

call sum()
```

Sections parallèles : syntaxe

- Fortran

```
!$omp sections[clause[,clause]...]
!$omp section
code block
[!$omp section
another code block
[!$omp section
…]]
!$omp end sections[nowait]
```

```
    private(list)
    firstprivate(list)
    lastprivate(list)
    reduction(operator|intrinsic:list)
```

Clauses possibles

Sections parallèles : syntaxe

- C/C++

```
#pragma omp sections[clause [clause...]]
{
 #pragma omp section
  structured block
 [#pragma omp section
  structured block
…]
}
```

```
    private(list)
    firstprivate(list)
    lastprivate(list)
    reduction(operator|intrinsic:list)
    nowait
```

Clauses possibles

Sections parallèles : complément

- Une manière plus simple de créer une section parallèle est de combiner les directives de région parallèle et de section parallèle
- Dans ce cas :
  - une synchronisation implicite est ajoutée à la fin de la directive **`sections`**
  - La directive **`sections`** n'admet pas la clause « **`nowait`** »

```
!$OMP PARALLEL SECTIONS
!$OMP SECTION
  CALL INIT(A)
 $OMP SECTION
  CALL INIT(B)
!$OMP END PARALLEL SECTIONS
```

```
#pragma omp parallel sections
{
  #pragma omp section
   init(A);
  #pragma omp section
    init(B);
  }
```

Exécution exclusive

Exécution exclusive

- **Objectif :** Exclure toutes les tâches à l'exception d'une seule pour exécuter certaines portions de code incluses dans une région parallèle
- **Exemple :** traiter le résultat d'une boucle parallèle
- Possible grâce aux constructions **`SINGLE`** et **`MASTER`**

La construction SINGLE

- Permet d'exécuter une portion de code par un et un seul thread (en général le premier thread qui arrive à la construction **SINGLE**)

  Les autres threads attendent en fin de région **SINGLE** à moins d'avoir spécifié la clause **nowait**

- Une clause supplémentaire **copyprivate** est autorisée pour diffuser les variables privées aux autres threads

```
 a = 1000
!$OMP PARALLEL DEFAULT(PRIVATE)
 a = 9000

!$OMP SINGLE
  a = -9000
!$OMP END SINGLE

 id = OMP_GET_THREAD_NUM()
 print *,"id :",id,"; A :",a

 !$OMP END PARALLEL
```

```
id : 0 ; A : 9000
id : 1 ; A : -9000
id : 2 ; A : 9000
```

SINGLE : syntaxe

Fortran

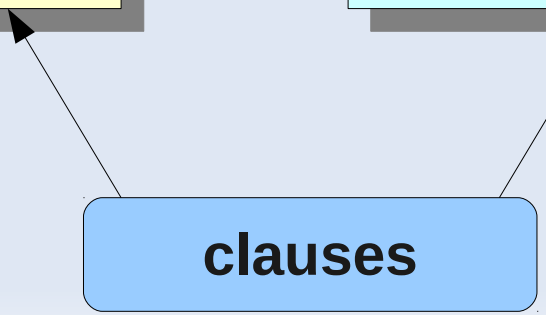
```
!$omp single[clause [clause…]]
 structured block
!$omp end single[nowait]
```

C/C++

```
pragma omp single[clause
  [clause…]]
structured block
```

```
private(list)
copyprivate(list)
firstprivate(list)
```

```
private(list)
copyprivate(list)
firstprivate(list)
nowait
```



La construction SINGLE : la clause copyprivate

```
 a = 1000;
#pragma omp parallel private(a)
 {
 a = 9000;
  #pragma omp single copyprivate(a)
  a = -9000;

 id = omp_get_thread_num();
 printf("id :%d , A :%d\n",id,a);

 }
```

```
id : 0 ; A : -9000
id : 1 ; A : -9000
id : 2 ; A : -9000
```

La construction MASTER

- Permet d'exécuter une portion de code par le thread master uniquement
- La construction n'admet aucune clause
- Pas de barrière de synchronisation ni en début ni en fin de la construction

```
 a = 1000
!$OMP PARALLEL DEFAULT(PRIVATE)
 a = 9000

!$OMP MASTER
  a = -9000
!$OMP END MASTER

 id = OMP_GET_THREAD_NUM()
 print *,"id :",id,"; A :",a

 !$OMP END PARALLEL
```

```
id : 0 ; A : -9000
id : 1 ; A : 9000
id : 2 ; A : 9000
```

Synchronisation

Synchronisation : Barrière

Supposons deux boucles s'exécutant à l'intérieur d'une région parallèle :

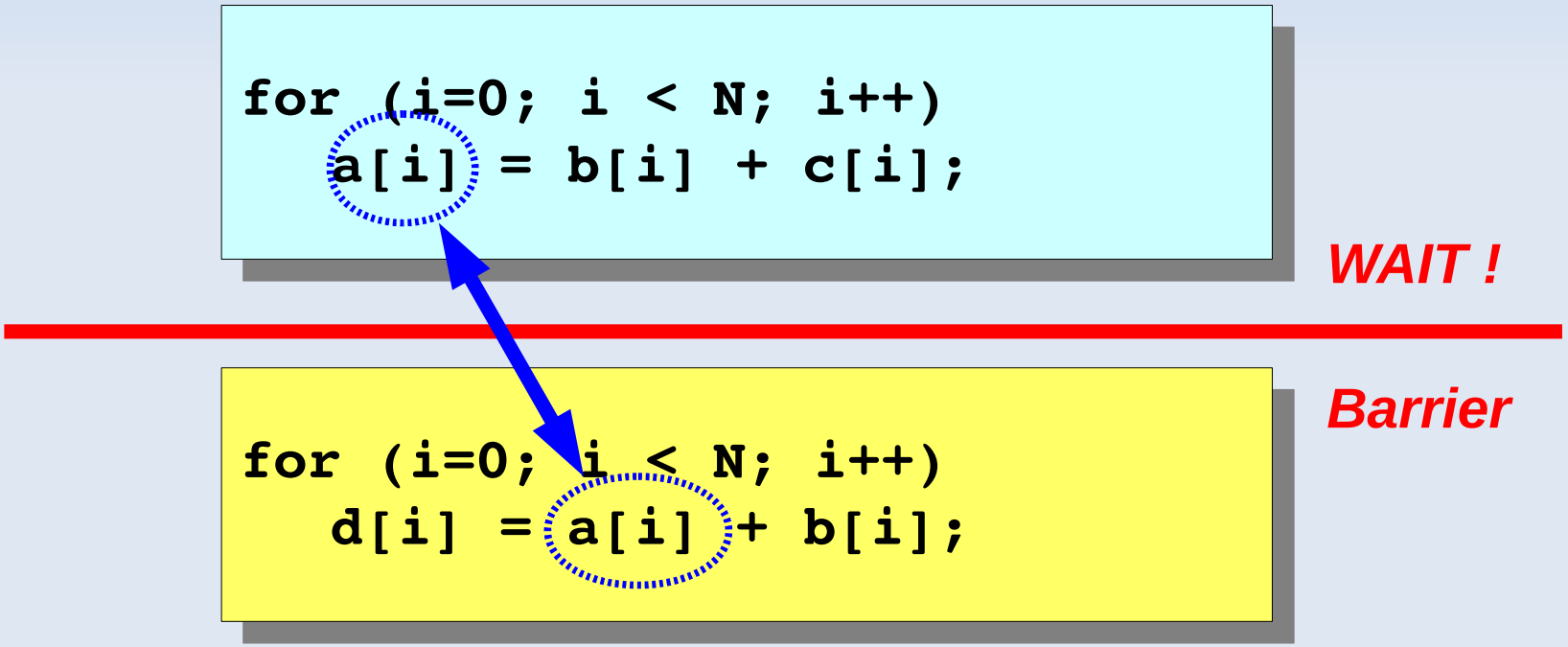
```
for (i=0; i < N; i++)
   a[i] = b[i] + c[i];
```

```
for (i=0; i < N; i++)
   d[i] = a[i] + b[i];
```

Résultats corrects ? Pourquoi ?

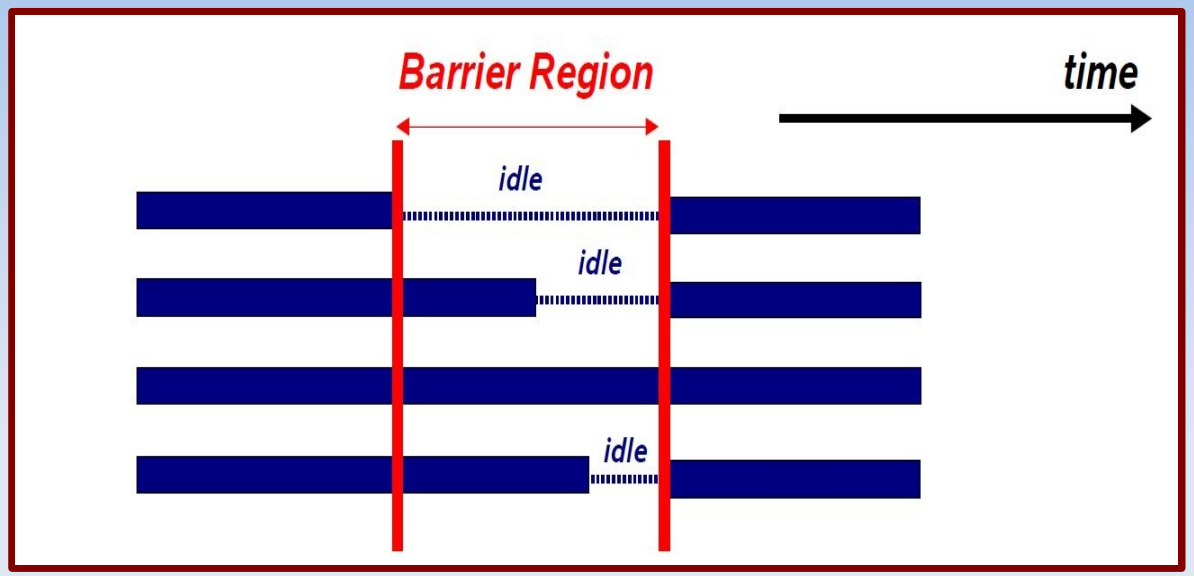
Synchronisation : Barrière (2)

Toutes les valeurs de **a[ ]** doivent être calculées avant qu'elles ne soient utilisées pour calculer **d[ ]**



Chaque thread attend que tous les autres threads soient arrivés à ce point de synchronisation pour reprendre son exécution

Synchronisation : Barrière (2)




```
#pragma omp barrier
```

```
!$OMP BARRIER
```

```
#pragma omp parallel
 {
 for (i=0; i < N; i++)
   a[i] = b[i] + c[i];
 #pragma omp barrier
 for (i=0; i < N; i++)
   d[i] = a[i] + b[i];
 }
```

Les barrières sont coûteuses ! Utilisez les avec précaution.

Synchronisation : section critique

Si ***some et prod*** sont «**shared**» on ne peut pas exécuter cette boucle en parallèle !
Problème d'accès concurrent

```
for (i=0;i<1000;i++)
      some=some+i;
      prod=prod*a[i];
```

On peut utiliser une région critique «**critical**»

```
for (i=0;i<1000;i++)
 #pragma omp critical
  {
   some=some+i;
   prod=prod*a[i];
  }
```

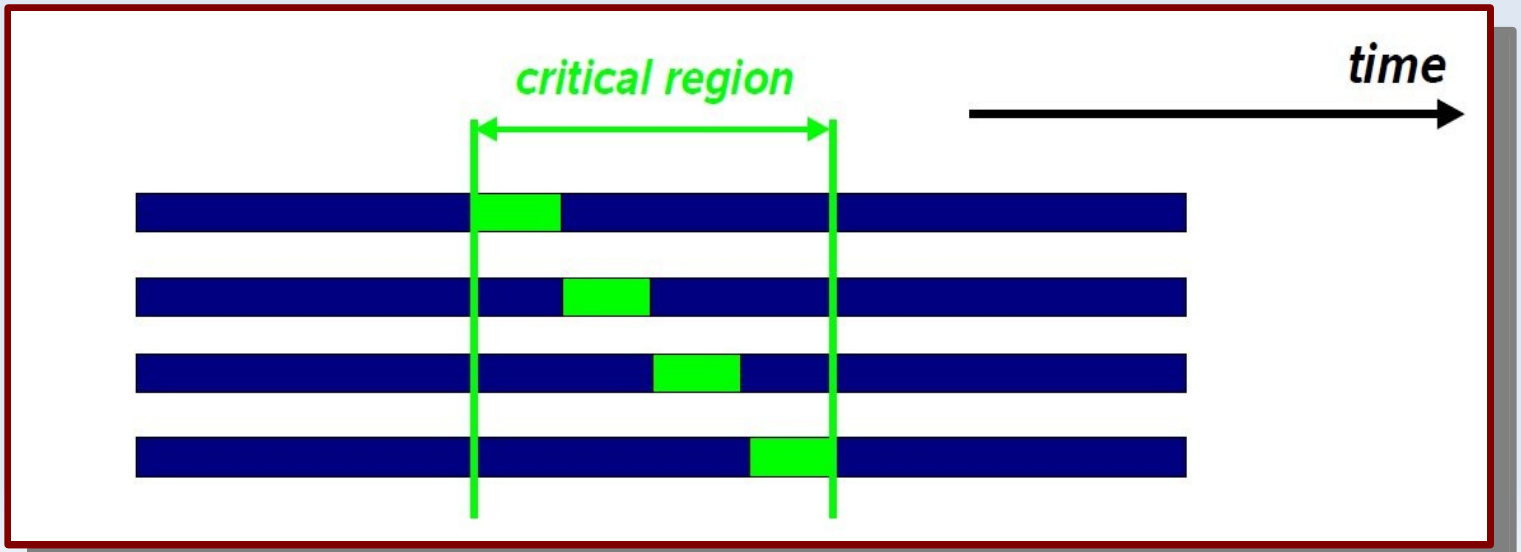
Les threads exécutent cette région tour à tour dans un ordre non déterministe.

Synchronisation : section critique (2)

```
#pragma omp critical [(name)]
 {<code block>}
```

```
!$omp critical [(name)]
  <code-block>
!$omp end critical [(name)]
```

Pas de barrière implicite ni à l'entrée ni à la sortie de la région



Synchronisation : section critique

Cas Particulier de Critical

La directive **«ATOMIC»** garantit qu'une variable partagée ne peut être lue et modifiée en mémoire que par un seul thread à la fois

```
#pragma omp atomic
 <statement>
```

```
!$omp atomic
 <statement>
```

```
program parallel
  implicit none
  integer :: id, compteur
  compteur=2010;
 !$OMP PARALLEL PRIVATE(id)
   id=OMP_GET_THREAD_NUM();
   !$OMP ATOMIC
    compteur=compteur+1;
   print *, "id:",id,&
    "compt = ",compteur
  !$OMP END PARALLEL

end program parallel
```

```
Id : 1, compt = 2010
Id : 0, compt = 2011
Id : 2, compt = 2012
Id : 2, compt = 2013
```

Synchronisation : directive FLUSH

- Utile pour rafraîchir la valeur d'une variable partagée en mémoire globale
- Peut servir à mettre en place un mécanisme de synchronisation entre les threads
- Directive implicite dans les cas :
  - Entrée et sortie **`omp critical`**
  - Sortie de : **`omp parallel`**, **`omp for`**, **`omp sections`**, **`omp single`**

```
#pragma omp flush(var1[,var2]…)
```

```
!$omp flush(var1[,var2]…)
```

Récapitulatif

	default	shared	private	firstprivate	lastprivate	copyprivate	if	reduction	shedule	ordred	copyin	nowait
parallel	●	●	●	●			●	●			●	
do			●	●	●			●	●	●		●
sections			●	●	●			●				●
workshare												●
single			●	●								
master												
threadprivate												
atomic												
critical												
flush												

Récapitulatif

Fonctions openMP

Name	Functionality
omp_set_num_threads	*Set number of threads*
omp_get_num_threads	Return number of threads in team
omp_get_max_threads	Return maximum number of threads
omp_get_thread_num	Get thread ID
omp_get_num_procs	Return maximum number of processors
omp_in_parallel	Check whether in parallel region
omp_set_dynamic	Activate dynamic thread adjustment
omp_get_dynamic	Check for dynamic thread adjustment
omp_set_nested	Activate nested parallelism
omp_get_nested	Check for nested parallelism
omp_get_wtime	Returns wall clock time
omp_get_wtick	Number of seconds between clock ticks

Récapitulatif

Variables d'environnement

OMP_NUM_THREADS	n (number of processors)
OMP_SCHEDULE “schedule,[chunk]”	static, “N/P” (1)
OMP_DYNAMIC { TRUE | FALSE }	TRUE (2)
OMP_NESTED { TRUE | FALSE }	FALSE (3)

(1) la taille des paquets correspond au nombre d'itération divisé par le nombre de threads
(2) le nombre de threads est limité au nombre de processeurs dans le système
(3) par défaut les régions parallèle sont désactivées

Notions avancées

Procédure orphéline

- Une procédure 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, FO,...)`** dans le corps de la procédure
- Ces directives sont dites orphelines : On parle de « procédures orphelines » (***`orphaning`***)

```
!$OMP PARALLEL IF (n.gt.256)
Call prod_mat_vect(a,x,y,n)
!$OMP END PARALLEL
```

```
subroutine prod_mat_vect&
                    (a,x,y,n)
..
..
!$OMP DO
 do i=1,n
   y(i)=SUM(a(i,:)*(x(:))
 end do
$OMP END DO

end subrotine ....
```

Données globales

```
....
include "global.h"
....
!$omp parallel private(j)
do j = 1, n
 call suba(j)
end do
!$omp end parallel
....
```

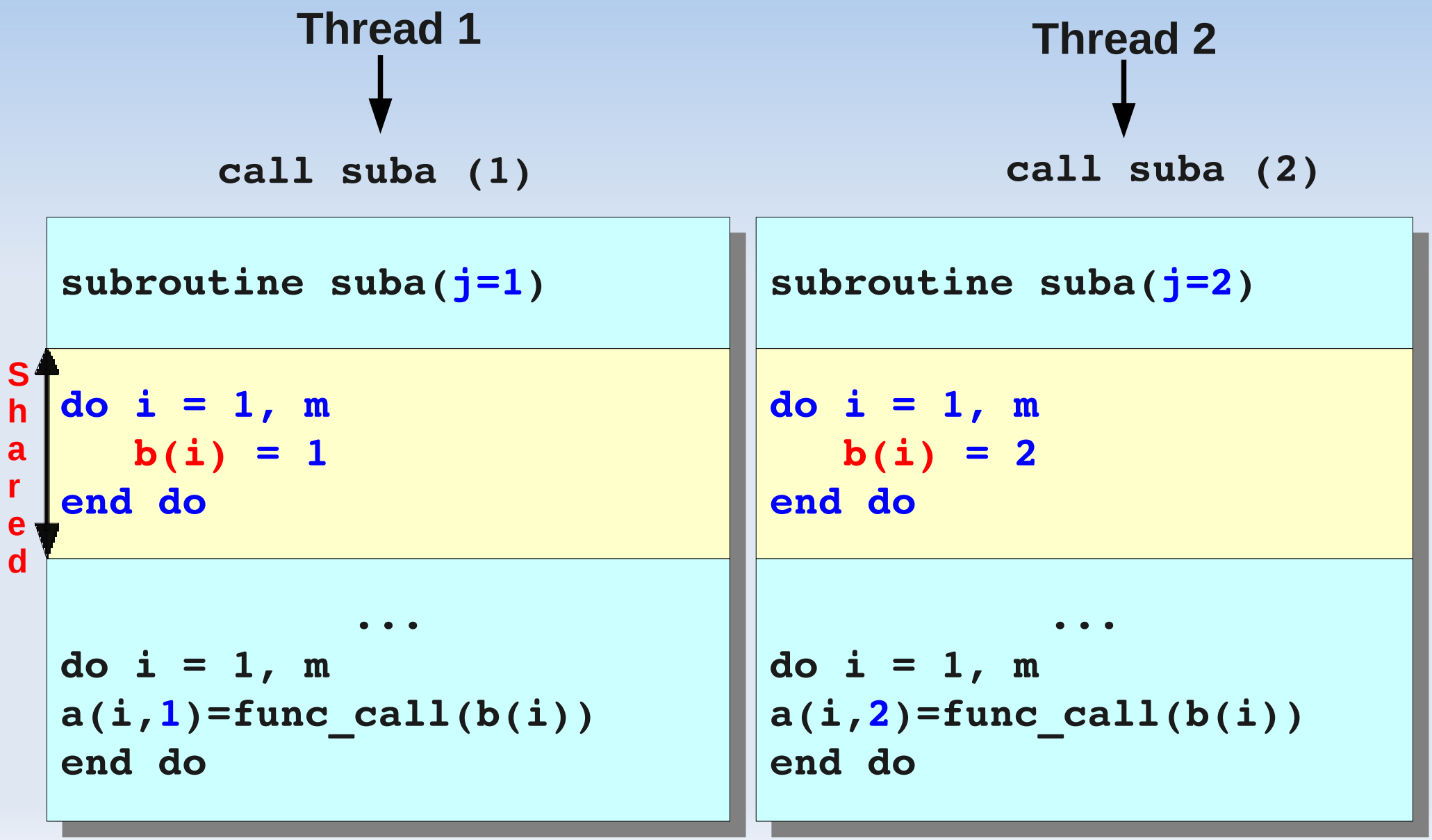
```
common /work/a(m,n),b(m)
```

```
subroutine suba(j)
.....
include "global.h"
.....
do i = 1, m
 b(i) = j
end do

do i = 1, m
a(i,j) = func_call(b(i))
end do
return
end
```

Race condition !

Données globales



Solution

```
....
include "global.h"
....
!$omp parallel private(j)
do j = 1, n
 call suba(j)
end do
!$omp end parallel
....
```

```
common /work/a(m,n)
common /tprivate/b(m)
!$omp threadprivate(/tprivate/))
```

```
subroutine suba(j)
.....
include "global.h"
.....
do i = 1, m
 b(i) = j
end do

do i = 1, m
a(i,j) = func_call(b(i))
end do
return
end
```

Le compilateur créera une copie privée du tableau B pour chaque thread, pour éviter les conflits d'accès :

Le nombre de copies sera automatiquement adapté au nombre de threads.

ThreadPrivate

Un thread peut-il conserver ses propres variables privées tout au long des sections parallèles dans un programme ? **Oui !**

```
!$omp threadprivate (/cb/ [,/cb/] ...)
```

```
#pragma omp threadprivate (list)
```

ThreadPrivate (copyin)

Objectif : *Initialisation des variables* `threadprivate`

La valeur de la variable therad master est copiée sur toutes les autres variables.

```
integer :: x,tid
common/mine/x
!$omp threadprivate(/mine/)
x=33
callomp_set_num_threads(4)
!$omp parallel private(tid)copyin(/mine/)
tid=omp_get_thread_num()
print *,'T:',tid,' x=',x
$omp end parallel
```

Note : la clause **copyin** est utilisée aussi avec les directives de **boucles** et de **sections**

```
T:1 i=33
T:2 i=33
T:0 i=33
T:3 i=33
```

Dépendance de données

Dépendance de données

- Pour bien paralléliser une boucle, le travail effectué dans une itération de la boucle ne doit pas dépendre du travail effectué dans une autre itération
- En d'autres termes, l'ordre d'exécution des itérations de la boucle doit être pertinent
- Certaines dépendances de données peuvent être évitées en modifiant le code

```
!$OMP PARALLEL DO PRIVATE(id)
   do i = 1, n
    a[i] = f(a[i-1]);
  end do
!$OMP END PARALLEL
```

Pas d'erreurs de compilation

Résultat FAUX !!

Dépendance de données

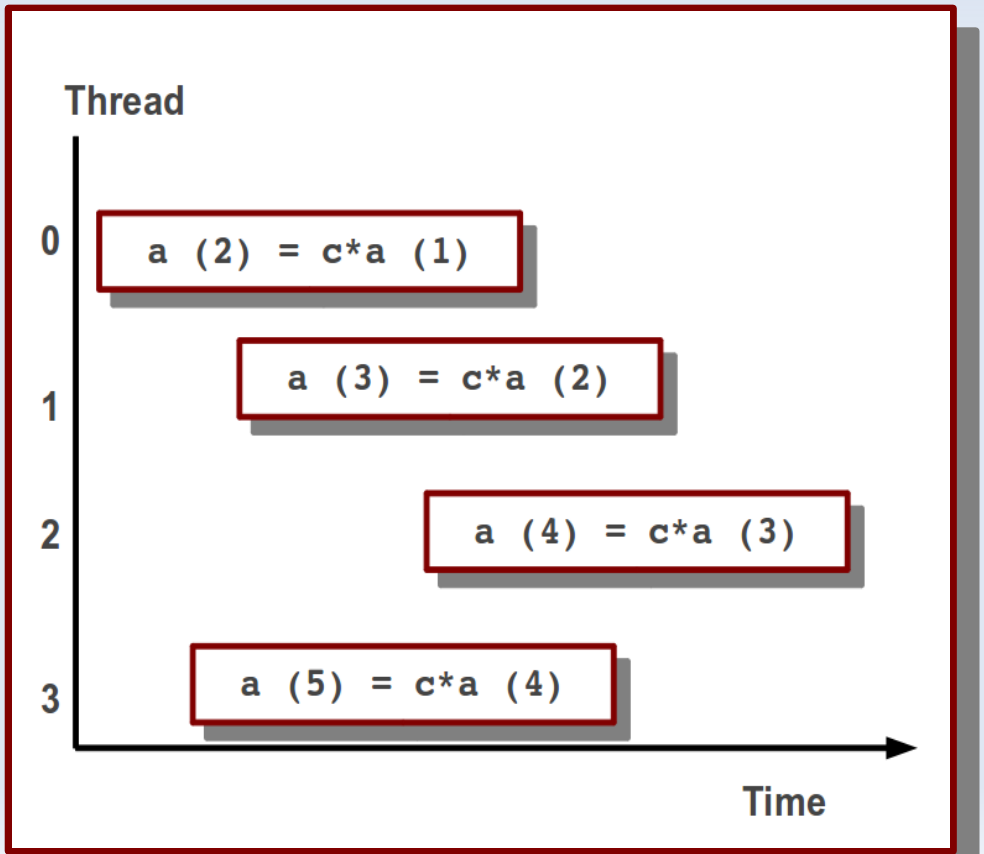
- Seules les variables (partagées) **modifiées** dans une itération et **lues** dans une autre itération peuvent causer des **dépendances** de données
- Les dépendances de données sont souvent difficiles à identifier
- Les outils de compilation peuvent assister cette identification

```
do i = 2,n,2
   a(i) = c*a(i-1)
end do
```

des dépendances ?

Récurrence

```
do i = 2,5
   a(i) = c*a(i-1)
end do
```



Dépendance de données

- **La dépendance de réduction** est éliminée en rajoutant la clause «**`reduction`**» à la boucle parallèle.

```
do i = 1,n
 xsum=xsum+ a(i)
 xmul = xmul * a(i)
 xmax= max(xmax,a(i))
 xmin= min(xmin,a(i))
end do
```

- Il y a une dépendance si :
  - **`idx(i)`** est différent de **`i`** à chaque itération ou
  - **`ndx(i)`** ne se répète jamais

```
do i = 1,n
 a(i) = c * a(idx(i))
 enddo

 do i = 1,n
 a(ndx(i)) = b(i)+c (i)
end do
```

Dépendance de données

- Si la **k-boucle** est parallélisable, alors il existe une dépendance liée à **a(i,j)**
- Peut être corrigé en rentrant la **k-boucle** dans toutes les autres boucles

```
do k = 1, n
 do j = 1, n
  do i = 1, n
    a(i,j)=a(i,j)+
                b(i,k)*c(k,j)
  end do
 end do
end do
```

```
do i = 1,n
callmyroutine(a,b,c,i)
enddo
subroutinemyroutine(a,b,c,i)
…
a(i) = 0.3 * (a(i-1)+b(i)+c)…
return
```

Dépendance de données : exemple

```
for (i=0; i <NumElements; i++)
    {

    array[i] = StartVal;

    StartVal++;
    }
```

```
#pragma omp parallel for

 for (i=0; i < NumElements; i++)

     {
       array[i] = StartVal + i;
     }

    StartVal += NumElements;
```

Dépendance de données

Ojectif

Minimiser le surcoût des récurrences

- Déplacer la dépendance dans une boucle séparée
- Paralléliser la boucle sans la dépendance

```
do i = 1, NHUGE
  a(i) = ...lots of math...
& + a(i-1)
end do
```

```
!$omp parallel do&
!$omp shared(junk) private(i)
 do i = 1, NHUGE
 junk(i) = ...lots of math..
end do

!Serial Loop
 do i = 1, NHUGE
    a(i) = junk(i) + a(i-1)
end do
```

Performances

Performances

- En général, les performances dépendent de l'architecture (processeur, mémoire) et de l'implémentation OpenMP utilisée

- Il existe, néanmoins, quelques règles de bonne conduite indépendantes de l'architecture

Règles de bonnes performances

- Si possible, utiliser l'auto-parallélisation du compilateur comme première étape
- Utiliser le **«profiling»** pour identifier les sections de code qui consomment du CPU
- Utiliser OpenMP pour paralléliser les boucles les plus importantes
- Si la boucle parallélisée ne fonctionne pas bien, vérifier :
  - le coût de démarrage de threads
  - si la boucle est de taille suffisante pour justifier le coût
  - les éventuels déséquilibres de charge
  - le nombre excessif de références à des variables partagées
  - les synchronisation inutiles

Performances : exemple

- Tout d'abord, paralléliser la boucle
  - *de préférence la boucle externe*
- c2 n'est jamais vrai, sauf si c1 est vrai aussi, **r** peut être **private**
- Mais, la boucle est «triangulaire» la durée des itérations peut être déséquilibrée entre les threads
  - *Utiliser **schedule dynamic** pour forcer un meilleur équilibrage de charge*
- Paralléliser la réduction

Code original

```
c1 = x(1)>0
c2 = x(1:10)>0
DO i=1,n
 DO j=i,n
 if (c1) then r(1:100) = …
 …
 if (c2) then … = r(1:100)
 sum(j) = sum(j) + …
 ENDDO
ENDDO
```

Performances : exemple(2)

```
c1 = x(1)>0
c2 = x(1:10)>0
ALLOCATE(xsum(1:nprocs,n))
c$omp parallel do private(i,j,r,myid)
c$omp& schedule(dynamic)
DO i=1,n
myid = omp_get_thread_num()
DO j=i,n
if (c1) then r(1:100) = …
…
if (c2) then … = r(1:100)
xsum(myid,j) = sum(myid,j) + …
END DO
END DO
c$omp parallel do
DO i=1,n
sum(i) = sum(i) + xsum(1:nprocs,i)
END DO
```

Performances : mesures du temps

- **OpenMP** offre deux fonctions :
  - **OMP_GET_WTIME** pour mesurer le temps d'exécution en seconde
  - **OMP_GET_WTICK** pour connaître la précision des mesures en secondes
- On peut mesurer le temps écoulé depuis un point de référence du code
- Cette mesure peut varier d'une exécution à l'autre selon la charge de la machine

```
#pragma omp parallel \
private(rang,t_ref,t_final)
 {
  rang  = omp_get_thread_num();
  t_ref = omp_get_wtime();
  prod_mat_vect(a,x,y);
  t_final = omp_get_wtime();
  printf("Rang : %d ;\
  Temps : %f\n",\
  rang,t_final-t_ref);
}
```

Exemple : MatMul

Application : Multiplication matricielle
N=10000
Thread Schedule : default (static)

Nombre de threads	1	2	3	4	5	6	7	8
Temps (s)	91	46.2	30.80	23.31	18.70	15.75	13.6	11.97

Conclusion

- OpenMP nécessite une machine multi-processeurs à mémoire partagée
- Permet la parallélisation progressive d'un programme séquentiel
- Partage du travail grâce aux boucles et sections parallèles
- Des synchronisations explicites sont parfois nécessaires

Questions ?