3 OpenMP™

Introduction à la programmation par directives

Plan

Introduction

Le modèle OpenMP

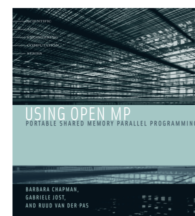
Aspect de base
- Région parallèle
- Distribution du travail
  - Les directives
  - L'ordonnancement du travail
- Attributs de données
- Les directives de synchronisation

Références

Site Web officiel : www.openmp.org
Spécification OpenMP 3.1

Books:

Using OpenMP, Barbara Chapman, Gabriele Jost, Ruud Van Der Pas, Cambridge, MA : The MIT Press 2007, ISBN: 978-0-262-53302-7



Parallel programming in OpenMP, Chandra, Rohit, San Francisco, Calif. : Morgan Kaufmann ; London : Harcourt, 2000, ISBN: 1558606718

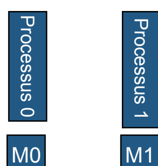
.

Communauté
La communauté des chercheurs et des développeurs d'OpenMP académique et industrielle
- http://www.compunity.org/

Conférence :
- WOMPAT, EWOMP, WOMPEI, IWOMP
- http://www.nic.uoregon.edu/iwomp2005/index.html#program

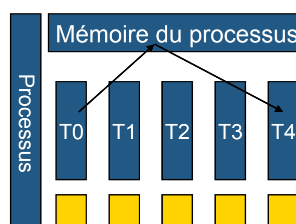
Processus versus threads

Processus



Deux unités indépendantes = deux pids

Threads



Mémoire visible par toutes les threads

Pile pour les variables privées de la thread

Les Threads POSIX

Une autre API de programmation mémoire partagée.

Bas niveau d'abstraction par rapport à OpenMP
- Seulement des fonctions, pas de directives ;
- Plus flexible, mais plus difficile à implémenter et à maintenir ;
- OpenMP peut être implémenté au-dessus des threads POSIX.

Disponibilité
- Pas d'interface en Fortran des threads POSIX

Le modèle OpenMP

Pourquoi OpenMP en 1997

Pas de portabilité pour les applications à mémoires partagées

- Chaque constructeur avait son API
- X3H5 et PCF n'ont pas abouti

Portabilité au travers de MPI

Pas de langages parallèles dominants

Les machines sont là

- Origin2000, SUN, …
- Exemplar, PC, ...

et les applications arrivent ...

Ce qu'apporte OpenMP

Une portabilité

* Fortran 77 et 90 depuis novembre 97
* C et C++ depuis décembre 98
* Unix et NT

Haut niveau de programmation (Directives)

Modèles de programmation

* parallélisme à grains fins (boucle)
* parallélisme à gros grains (SPMD) ← Pour l'extensibilité

Une parallélisation incrémentale

Principaux acteurs

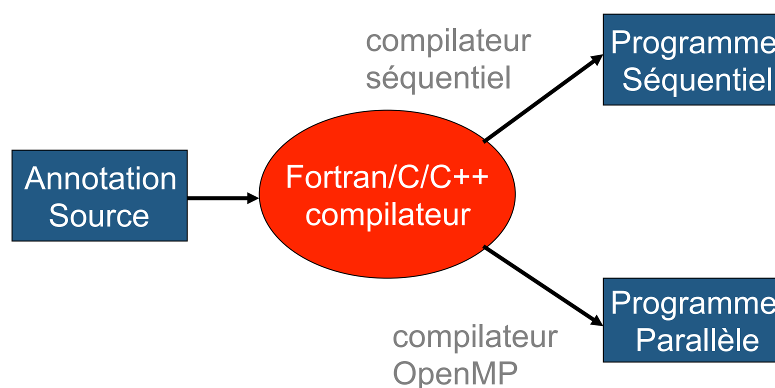
Constructeurs
- *Compacq Computer Corp.*
- *Hewlett-Packard Compagny*
- *International Business Machines*
- *Intel Corp.*
- *Silicon Graphics / Cray Research*
- *Sun Microsystems*

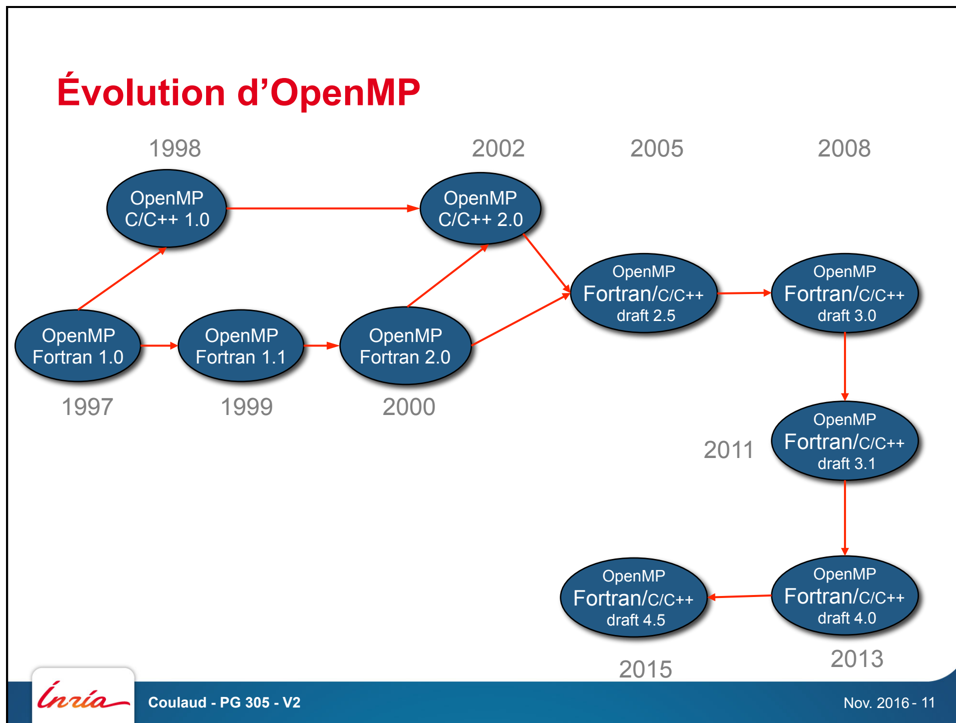
Compilateurs
- *Absoft Corp.*
- *Edinburgh Portable Compilers*
- *Kuck & Associates, Inc.*
- *Myrias Computer Technologies*
- *Numerical Algorithms Group Ltd.*
- *The Portland Group, Inc.*

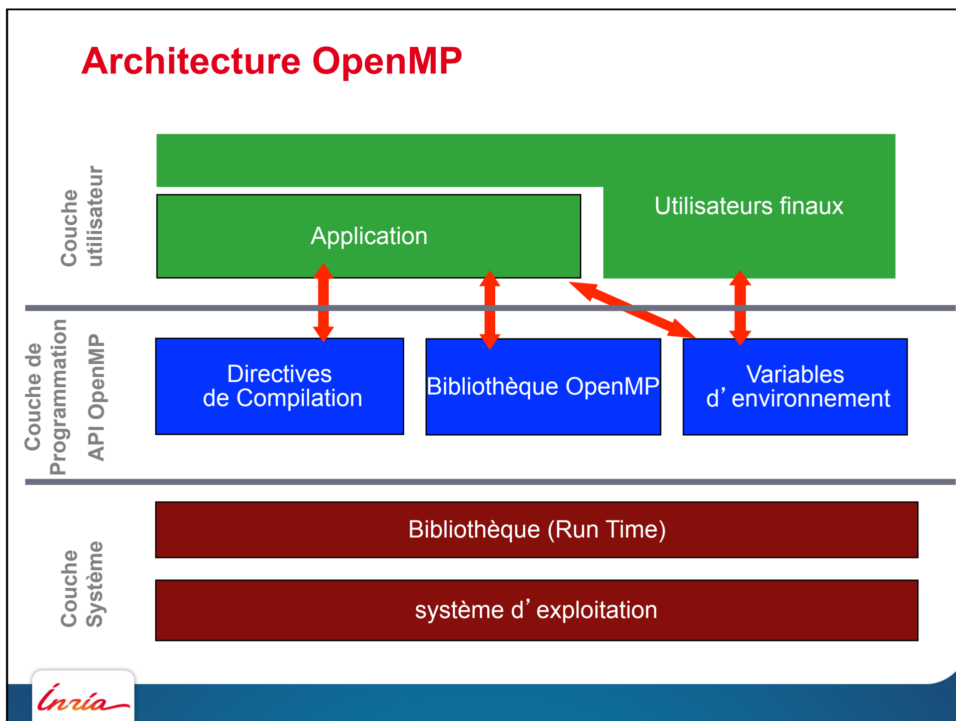
Recherche
- *Perdue University*
- *US Department of Energy ASCI Program*

Applications
- *ADINA R&D, Inc.*
- *ANSYS, Inc.*
- *ILOG CPLEX Division*
- *Fluent, Inc.*
- *LSTC Corp.*
- *MECALOG SARL*
- *Oxford Molecular Group PLC*

Comment utiliser OpenMP

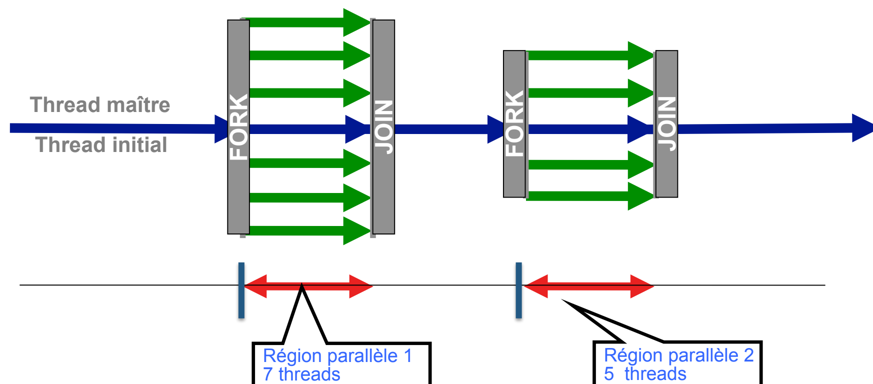



Évolution d'OpenMP
1998
2002
2005
2008
OpenMP C/C++ 1.0
OpenMP C/C++ 2.0
OpenMP Fortran/C/C++ draft 2.5
OpenMP Fortran/C/C++ draft 3.0
OpenMP Fortran 1.0
OpenMP Fortran 1.1
OpenMP Fortran 2.0
1997
1999
2000
2011
OpenMP Fortran/C/C++ draft 3.1
OpenMP Fortran/C/C++ draft 4.5
OpenMP Fortran/C/C++ draft 4.0
2015
2013
Coulaud - PG 305 - V2
Nov. 2016 - 11


Architecture OpenMP
Couche utilisateur
Application
Utilisateurs finaux
Couche de Programmation API OpenMP
Directives de Compilation
Bibliothèque OpenMP
Variables d' environnement
Couche Système
Bibliothèque (Run Time)
système d' exploitation

Modèle d'exécution

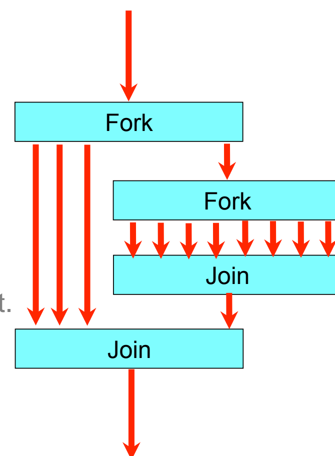
Modèle Fork and join : le maitre lance un ensemble de threads



Modèle d'exécution (suite)

Le modèle Fork/Join peut être emboité

- Géré automatiquement à la compilation
- Indépendant du nombre de threads s'exécutant actuellement.



Modèle mémoire

Les threads communiquent en partageant des variables

Le partage est défini par :
- Toute variable qui est vue par deux ou plusieurs threads est partagée (mémoire)
- Toute variable qui est vue par un seul thread est privée. (sa propre mémoire)

Les situations de concurrence (*race condition*) sont possibles
- Utiliser des synchronisations pour éviter des conflits sur les données;
- Changer le statut de la variable pour minimiser le besoin de synchroniser.

Modèle mémoire

Une variable partagée est visible par tous les threads.

Une variable privée est dupliquée sur les threads.

Une variable interne dans une région parallèle est une variable appartenant à la mémoire propre du thread (pile).

Comment est utilisé classiquement OpenMP ?

OpenMP est classiquement utilisé pour paralléliser des boucles :
- trouver la boucle la plus consommatrice en temps CPU.
- éclater sur plusieurs threads/processeurs.

Éclater la boucle entre plusieurs threads

```
void main()
{
   double Res[1000];

   for(int i=0;i<1000;i++) {
      do_huge_comp(Res[i]);
   }
}
```

Programme séquentiel

```
void main()
{
   double Res[1000];
   #pragma omp parallel for
   for(int i=0;i<1000;i++) {
      do_huge_comp(Res[i]);
   }
}
```

Programme Parallèle

Exemple

Code source

```
#include "omp.h"
….;
#pragma omp parallel
{
      … région parallèle
}
```

Compilation

GNU : gcc -fopenmp filename.cc –o filename
INTEL : icc -openmp filename.cc –o filename

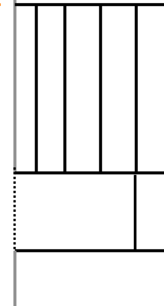
Exécution

```
$ export OMP_NUM_THREADS=4
$ filename
```

Un exemple de programme

```
    ...
    common /setup/ iam,ipiece,npoints,nzone
!$OMP THREADPRIVATE(/setup/)
    dimension field(N), spectrum(NZ)
     npoints = N
     nzone   = NZ
     energy  = 0.0
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(field,spectrum)  &
!$OMP& REDUCTION(+: energy) COPYIN(/setup/)
     np   = omp_get_num_threads()
     iam = omp_get_thread_num()
     call initialise(field, spectrum)
     call calcul (energy, field, spectrum)
!$OMP BARRIER
!$OMP SINGLE
     call affiche(spectrum)
!$OMP END SINGLE
!$OMP END PARALLEL
    print*, energy
```

Master



Un exemple de programme

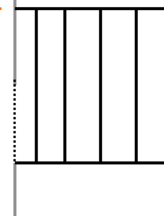
```
#include <omp.h>

main () {

 int var1, var2, var3;

Serial code
 . . .
 // Début de la section parallèle.  Lancement d'un ensemble de threads
//   Précisez la portée des variables
 #pragma omp parallel private(var1, var2) shared(var3)
 {
  //  Section Parallèle exécutée par tous les threads
 . . .
 // Tous les threads rejoingnent le thread maitre et disparaissent
}
Reprise du code de séquentiel
. . .
 }
```

Master



OpenMP: Blocs structurés (C/C++)

La plupart des constructeurs OpenMP s'appliquent à des blocs structurés

- Bloc structuré : un bloc avec un point d'entré au début et un point de sortie à la fin.
- Les seuls branchements autorisés sont le STOP en Fortran et exit() en C/C+

```
#pragma omp parallel
{
        int id = omp_get_thread_num();
more:  res(id) = do_big_job(id);
        if(conv(res(id)) goto more;
}
   printf(" All done \n");
```

Un bloc structuré

```
      if(go_now()) goto more;
#pragma omp parallel
{
        int id = omp_get_thread_num();
more:  res(id) = do_big_job(id);
        if(conv(res(id)) goto done;
         go to more;
}
done:     if(!really_done()) goto more;
```

Un bloc non structuré

OpenMP: blocs structurés

C/C++: un bloc est une déclaration unique ou un groupe d'instructions entre les accolades { }

```
#pragma omp parallel
{
    id = omp_thread_num();
    res(id) = lots_of_work(id);
}
```

```
#pragma omp for
  for(I=0;I<N;I++){
       res[I] = big_calc(I);
       A[I] = B[I] + res[I];
  }
```

Fortran: un bloc est une déclaration unique ou un groupe d'instructions entre la paire de directives.

```
C$OMP PARALLEL
10    wrk(id) = garbage(id)
      res(id) = wrk(id)**2
      if(conv(res(id)) goto 10
C$OMP END PARALLEL
```

```
C$OMP PARALLEL DO
    do I=1,N
        res(I)=bigComp(I)
    end do
C$OMP END PARALLEL DO
```

Syntaxe d'OpenMP

La plupart des fonctionnalités en OpenMP sont des directives de compilations.

Les directives prennent la forme suivante :

Sentinelle directive [clause [clause]…]

FORTRAN	C/C++
Format fixe : **C$OMP !$OMP *$OMP** Format libre : **!$OMP** Module Fortran 95 **OMP_LIB**	**#pragma omp** Fichier d'include **: *omp.h***

Un programme OpenMP par directives peut-être compilé par un compilateur qui ne supporte pas openMP.

Syntaxe d'OpenMP Fortran

Exemple

- Format fixe
  - Début en colonne 1
  - Blanc ou carte suite en colonne 6

```
C6
      USE OMP_LIB
…
!$OMP PARALLEL
!$OMP+ PRIVATE(I)
….
!$OMP END PARALLEL
```

- Format libre

```
USE OMP_LIB
…
!$OMP  PARALLEL &
   !$OMP& PRIVATE(I)
….
!$OMP  END PARALLEL
```

Syntaxe d'OpenMP

Compilation conditionnelle, deux possibilités

- sentinelles
  - Format fixe : C$!$ ou *$
  - Format libre : !$
- Macro du préprocesseur _OPENMP

Exemple

Fortran

```
!$      iam = omp_get_thread_num()
```

C/C++/Fortran

```
#ifdef _OPENMP
    iam = omp_get_thread_num()
#endif
```

Directives OpenMP

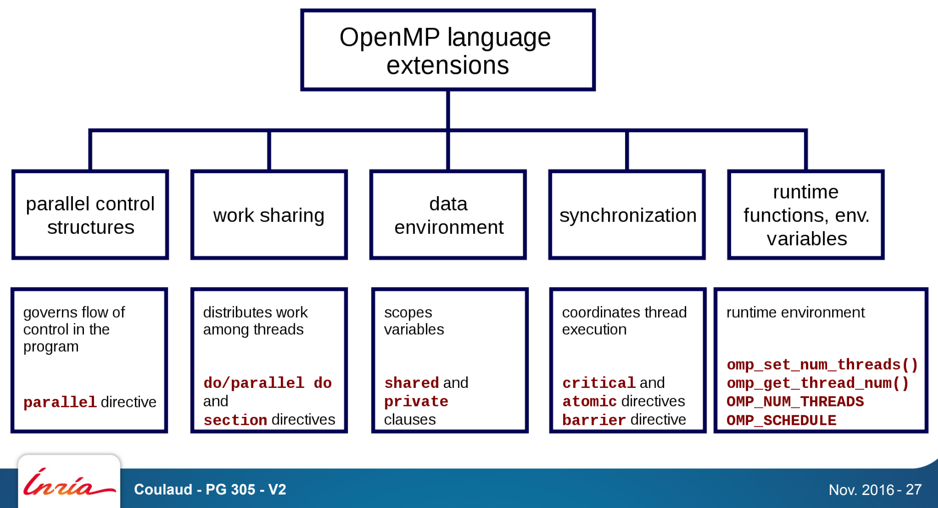
Directives pour créer une zone parallèle :
région parallèle

Directives pour partager du travail :
- Do/for, workshare, sections, task

Directives pour le statut des données :
- shared, private, firstprivate, lastprivate threadprivate
- reduction, ...

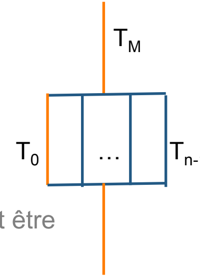
Directives de synchronisation :
- barrier, critical, atomic, flush
- nowait

OpenMP en 1 transparent



RÉGION PARALLÈLE

OpenMP : Région Parallèle (1)



Création uniquement par

#pragma omp parallel [clause(,), clause, ….]
bloc de code à exécuter par chaque thread

Duplication de l'exécution

- chaque thread joue le même code – les données peuvent être différentes
- autorise du travail en fonction du numéro du thread
- Seul le thread maître continue à la fin

Barrière implicite à la fin

Nombre de threads

- Fixé par une fonction, variable ou une clause
- Variable en fonction de l'état du système

OpenMP : Région Parallèle (2)

Clauses

shared (list)
private (list)
firstprivate (list)
default **(shared | none)** en C/C++, pas de restriction en fortran.

reduction (opérateur : list)
copyin (list)

if (expression logique scalaire)
num_threads (expression entière scalaire)
} Contrôler la granularité.

OpenMP : Région Parallèle (3)

Restriction

- Pas de branchement d'entré ou de sortie (non conforme)
  le code doit être un bloc structuré
- Le code ne doit pas dépendre de l'ordre l'exécution des instructions
- une seule condition *if*
- une seule *num_threads* avec une expression positive
- Fortran
  - PARALLEL / END PARALLEL dans la même unité
  - Comportement des I/O asynchrones par différentes threads est non spécifié
- C/C++
  - Un throw exécuté dans un thread est attrapé par le même thread et reste dans la même région parallèle.

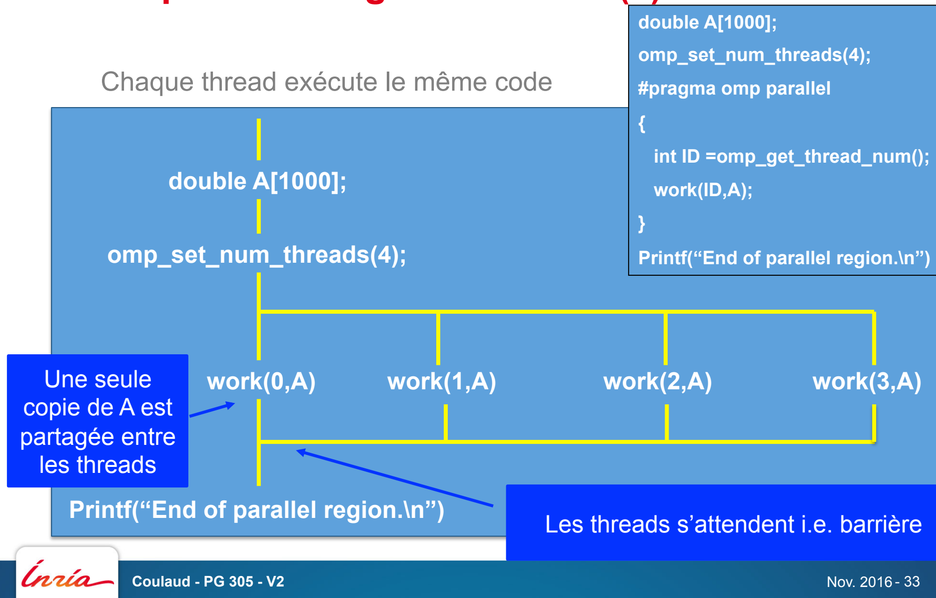
OpenMP : Région Parallèle (4)

Exemple

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
   int ID =omp_get_thread_num();
   work(ID,A);
}
```

Chaque thread exécute le code redondant dans le bloc structuré

Chaque thread appelle work(ID,A) pour `ID = 0 to 3`



OpenMP : Région Parallèle (6)

```
PROGRAM simple

INTEGER :: NTHREADS, TID,  OMP_GET_NUM_THREADS
INTEGER :: OMP_GET_NUM_THREADS, OMP_GET_THREAD_NUM     <-- USE OMP_LIB
                                                           OpenMP V 2.0
! Fork a team of threads giving them their own copies of variables

!$OMP PARALLEL PRIVATE(NTHREADS, TID)
! Obtain thread id
   TID = OMP_GET_THREAD_NUM()
    .....
!
IF (TID .EQ. 0) THEN                          ! Only master thread does this
      NTHREADS = OMP_GET_NUM_THREADS()
      PRINT *, 'Number of threads = ', NTHREADS
END IF
!
!$OMP END PARALLEL   ! All threads join master thread and disband
                                              OpenMP V 1.x
END PROGRAM simple
```

Nombre de threads fixé par une variable : setenv OMP_NUM_THREADS 64

Coulaud - PG 305 - V2 Nov. 2016 - 34

OpenMP: Région Parallèle
La clause IF

Peut être utilisé pour désactiver la parallélisation dans certains cas

```
#pragma omp parallel if (expression)
```

Exemple : quand la taille d'un paramètre est trop petite

```
        integer id, N
!$OMP PARALLEL PRIVATE(id)  IF(N.gt.1000)
            id = omp_get_thread_num()
            res(id) = big_job(id)
!$OMP END PARALLEL
```

La région parallèle est exécutée sur N threads seulement si l' expression logique est .TRUE.

OpenMP: Région Parallèle
la clause num_threads

Contrôler le nombre de threads utilisés dans une région parallèle

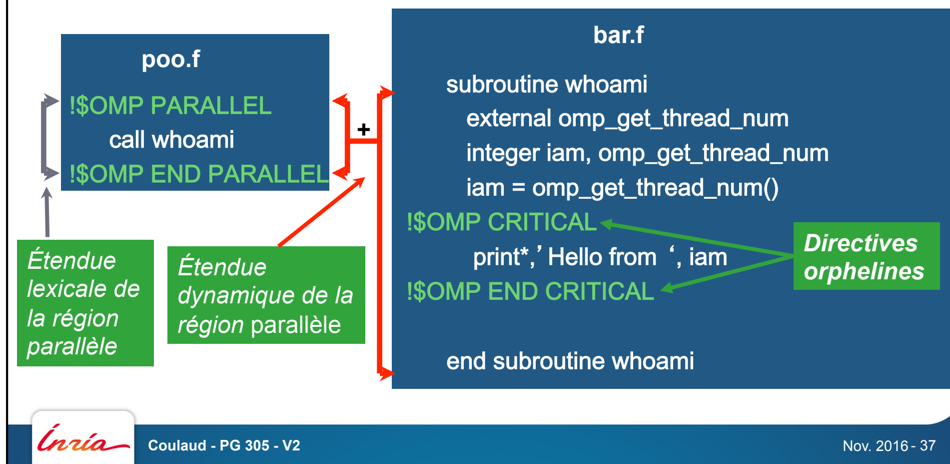
```
#pragma omp num_threads (expression)
```

```
#include <omp.h>
 main() {

#pragma omp parallel num_threads(10)
   {
       ...
       parallel region
      ...
    }
}
```

OpenMP: Portée des directives

Les directives OpenMP peuvent être sur plusieurs fichiers.



Exercice 1
programme "Hello world"

Écrire un programme multithreadé ou chaque thread affiche "hello world" et son numéro de thread.

Le thread 0 affiche en plus dans la région parallèle le nombre de threads

```
omp_get_thread_num()
omp_get_num_threads() / omp_set_num_threads(10)
export OMP_NUM_THREADS=10
```

Exercice 1
programme "Hello world"

icc –openmp hello.c

export OMP_NUM_THREADS=3 ;
./a.out

Hello World from thread = 0
Number of threads = 3
Hello World from thread = 1
Hello World from thread = 2

CONSTRUCTION DE TRAVAIL PARTAGÉ

Construction de travail partagé

Partager le travail parmi les threads
Pas de création de threads
Pas de barrière implicite en entrée mais une en sortie

Les directives :

- la directive DO ou for
- la directive SECTIONS
- la directive SINGLE
- La directive WORKSHARE

Restrictions

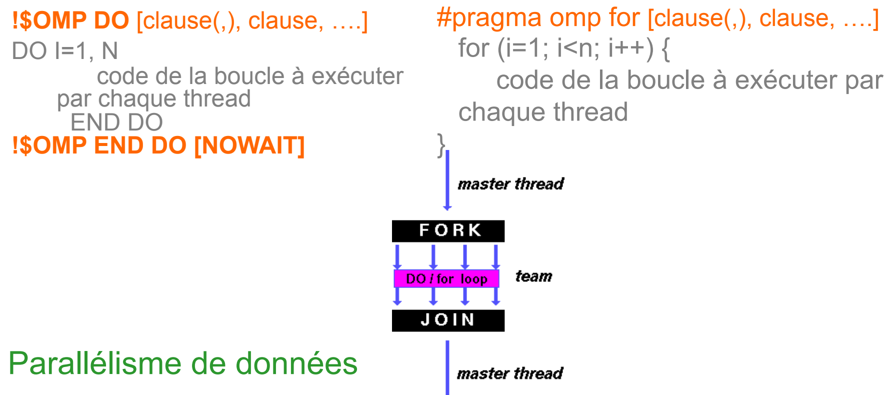
- Chaque région de partage doit être rencontrée par tous les threads ou par aucun
- La séquence de partage et de barrière doit être la même sur chaque thread

Construction de travail partagé
La directive DO ou for (1)

Partager les itérations d' une boucle à travers l' équipe

```
!$OMP DO [clause(,), clause, ….]
DO I=1, N
        code de la boucle à exécuter
   par chaque thread
    END DO
!$OMP END DO [NOWAIT]
```

```
#pragma omp for [clause(,), clause, ….]
  for (i=1; i<n; i++) {
      code de la boucle à exécuter par
  chaque thread
}
```



Parallélisme de données

Construction de travail partagé
La directive DO ou for (2)

Clauses

- private, firstprivate , lastprivate
- reduction (operator : list)
- schedule (type [,chunck])
- collapse(n)
- ordered
- nowait

Restrictions

- boucle avec un indice entier, A(i),
- contrôle de boucle (pas de do while)

C++

Il faut utiliser des itérateurs aléatoires pour accéder aux données en temps constant. Sinon il faut utiliser un parallélisme de tâche.

Construction de travail partagé
La directive DO ou for (2)

```
!$OMP PARALLEL
     call init(a)
!$OMP DO
     do i=1,N
        …
        …
     enddo
!$OMP END DO
     call display(a)
!$OMP END PARALLEL
```

Exécution dupliquée

Travail partagé : exécute différentes itérations

Exécution dupliquée

La clause collapse

collapse(n) : indique le nombre de boucles dans un ensemble de boucles imbriquées qui doivent être regroupées en une seule itération.

```
void bar(float *a, int i, int j, int k);

int kl, ku, ks, jl, ju, js, il,
void sub(float *a) {
    int i, j, k;
     #pragma omp for collapse(2) private(i, k, j)
     for (k=kl; k<=ku; k+=ks)
         for (j=jl; j<=ju; j+=js)
              for (i=il; i<=iu; i+=is)
                   bar(a,i,j,k);
}
                         C/C++
```

Fusionne les boucles k et j

Construction de travail partagé
La directive DO ou for (un exemple)

```
!$OMP PARALLEL SHARED(A,B,C,N) PRIVATE(I)
!$OMP DO SCHEDULE(STATIC,CHUNK)
DO I = 1, N
      C(I) = A(I) + B(I)
END DO
!$OMP END DO NOWAIT
!$OMP END PARALLEL
```

Plus de barrière

OpenMP: différentes approches

Code séquentiel

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

Parallélisation manuelle

```
#pragma omp parallel
{
  int id    = omp_get_thread_num();
  int Nthr = omp_get_num_threads();
  int istart = id*N/Nthr, iend = (id+1)*N/Nthr;
  for (int i=istart; i<iend; i++) { a[i]=b[i]+c[i]; }
}
```

Parallélisation automatique

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

Problème avec les boucles

Equilibrage de charge

- Si toutes les itérations s'exécutent à la même vitesse, les processeurs sont utilisés de manière optimale ;
- Si certaines itérations sont plus rapides que d'autres, certains processeurs seront plus lents pour traiter leurs itérations, réduisant ainsi l'accélération ;
- Si nous ne connaissons pas à priori la répartition du travail, il peut être nécessaire de redistribuer dynamiquement la charge.

Granularité

- La création de threads et la synchronisation prennent du temps ;
- Affectation de travail pour les threads peut prendre plus de temps que l'exécution elle-même! ;
- Besoin de fusionner le travail (grain grossier) pour recouvrir le surcout des threads.

Compromis entre l'équilibrage de charge et de la granularité!

L'ordonnancement du travail La clause SCHEDULE

Format : **schedule (type [,chunk])**

Un exemple :

!$OMP PARALLEL DO SCHEDULE (**type**)

#pragma omp schedule (type)

type est à choisir parmi

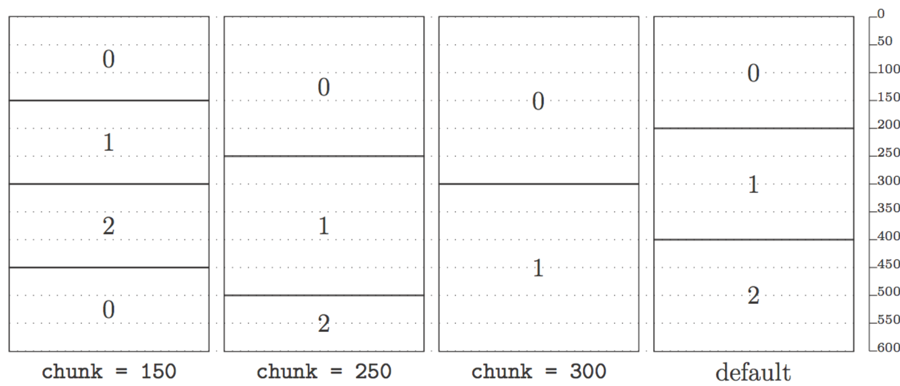
- static (chunk)
- dynamic (chunk)
- guided (chunk)
- auto c'est le compilateur ou le runtime qui décide
- runtime

Décidé à l'exécution et spécifié par une variable OMP_SCHEDULE

setenv OMP_SCHEDULE STATIC,100

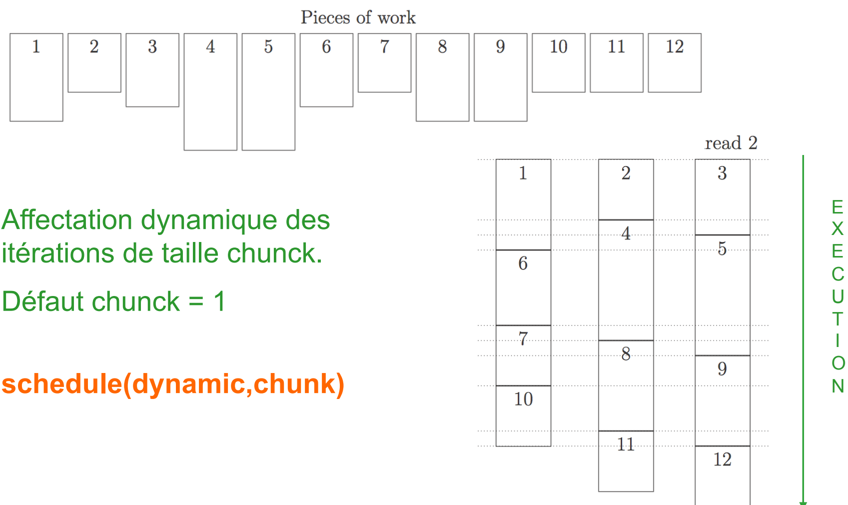
Si pas de cause, l'ordonnancement dépend de l'implémentation !!!

L'ordonnancement du travail
Le type STATIC



Exemple :
600 itérations
SCHEDULE(STATIC,chunk)

L'ordonnancement du travail
Le type DYNAMIC



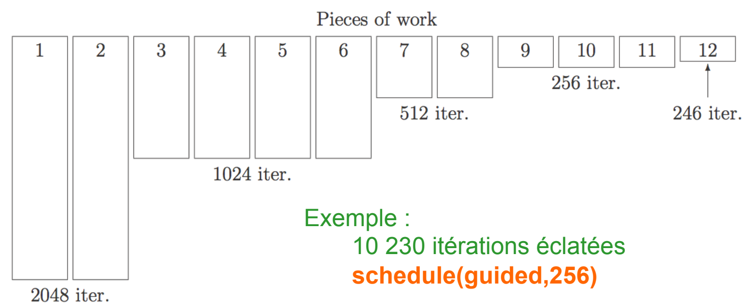
Affectation dynamique des itérations de taille chunck.

Défaut chunck = 1

schedule(dynamic,chunk)

L'ordonnancement du travail
Le type GUIDED

Les itérations sont coupées en morceaux de taille exponentiellement décroissante, chunk est le nombre d'itérations du plus petit morceau. Défaut : chunk = 1
Affectation dynamique entre les threads.



Exemple :
10 230 itérations éclatées
schedule(guided,256)

Comment choisir le type de l'ordonnanceur ?

Ordonnancement	Quand l'utiliser
static	Travail par itération est prédictible et similaire
dynamic	Travail par itération est imprévisible et très variable
guided	Cas spécial du cas dynamique pour réduire le surcoût.
auto	Aucune idée

Construction de travail partagé
La directive sections

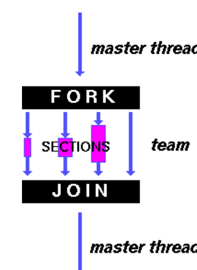
Zone non itérative, chaque section est exécutée par un thread

```
!$OMP SECTIONS clause[[,] clause…]
!$OMP SECTION
    < code bloc 1>
!$OMP SECTION
    < code bloc 2>
!$OMP SECTION
    . . .
!$OMP END SECTIONS [NOWAIT]
```

Clauses :
- private
- firstprivate
- Lastprivate
- reduction

```
#pragma omp sections clause[[,] clause…]
{
#pragma omp section
    < code bloc 1>
#pragma omp section
    < code bloc 2>
#pragma omp section
    . . .
}
```

Clauses :
- private
- firstprivate
- lastprivate
- reduction
- nowait



Parallélisme procédurale

Construction de travail partagé
La directive sections (un exemple)

```
!$OMP PARALLEL
!$OMP SECTIONS

!$OMP SECTION
      CALL XAXIS()
!$OMP SECTION
      CALL YAXIS()
!$OMP SECTION
      CALL ZAXIS()

!$OMP END SECTIONS NOWAIT

!$OMP END PARALLEL
```

```
#pragma omp parallel default(none)\
      shared(n,a,b,c,d) private(i)
{
#pragma omp sections nowait
  {
#pragma omp section
      for (i=0; i<n-1; i++)
         b[i] = (a[i] + a[i+1])/2;
#pragma omp section
      for (i=0; i<n; i++)
         d[i] = 1.0/c[i];
  } /*-- End of sections --*/
} /*-- End of parallel region --*/
```

La directive MASTER

Détermine un section où uniquement la thread maître exécute le code
- Le reste de l'équipe saute la section et continue l'exécution après la fin de la section maître
- Pas de barrière implicite à l'entrée et à la sortie de la section

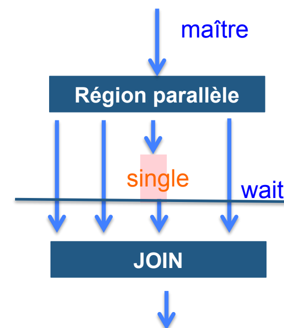
```
!$omp master
    bloc structuré
!$omp end master
```

```
#pragma omp master
    { bloc structuré }
```

```
#pragma omp parallel {
      .... ;
      #pragma omp master
       { printf(" Hello \n"); }

      .....
}
```

La directive SINGLE



un seul thread du groupe va exécuter le code

```
!$OMP SINGLE clause[[,] clause…]
    bloc structuré
!$OMP END SINGLE [end_clause]
```

```
#pragma omp single clause[[,] clause…]
    { bloc structuré }
```

Clauses :
- Début : private, firstprivate
- Fin : copyprivate, nowait

Clauses :
- private, firstprivate
- copyprivate, nowait

Construction de travail partagé
La directive SINGLE (un exemple)

```
!$OMP PARALLEL SHARED(A,B,C,N) PRIVATE(I)

!$OMP SINGLE
     call output(…)
!$OMP SINGLE

!$OMP END PARALLEL
```

Restrictions

- La clause copyprivate ne peut pas être utilisée avec nowait ;
- Une seule clause nowait.

Construction de travail partagé
La directive WORKSHARE

Uniquement en fortran.
Permet de partager les itérations d'une boucle à travers l'équipe

```
!$OMP WORKSHARE  [NOWAIT]
code à exécuter par chaque thread
!$OMP END WORKSHARE [NOWAIT]
```

Code Fortran95

Tableau
ForALL, where
Fonction elemental

- Pas de clause
- Barrière implicite en entré et en sortie

Attention le surcoût peut-être élevé

Construction de travail partagé
La directive WORKSHARE (une exemple)

```
!$OMP PARALLEL SHARED(A,B,C,N) PRIVATE(I)

!$OMP WORKSHARE
      A(:,:) = 1.0
      C(:,:) = A(:,:) + B(:,:)
      where ( D(:,: )  >  0;) E(:,:) = sqrt((D(:,:))
!$OMP END WORKSHARE NO WAIT          <-------- pas de barrière

!$OMP END PARALLEL
```


Construction de travail partagé
Restriction

Les directives doivent se trouver dans une région parallèle

Ces directives doivent être rencontrées

- par tous les threads du groupe
- dans le même ordre

Combiner région parallèle et travail partagé

On peut combiner les directives région parallèle et travail partagé :

- Directive do parallèle

```
!$OMP PARALLEL DO [ … ]
boucle do
!$OMP END PARALLEL DO
```

- Directive worshare parallèle

```
!$OMP PARALLEL WORKSHARE […]
    boucle do
!$OMP END PARALLEL WORKSHARE […]
```

- Directive sections parallèle

```
!$OMP PARALLEL SECTIONS […]
!$OMP SECTION
    Bloc 1
!$OMP SECTION
    Bloc 2
    …
!$OMP END PARALLEL SECTIONS
```

Combiner région parallèle et travail partagé

On peut combiner les directives région parallèle et travail partagé :

- Directive do parallèle

```
#pragma omp parallel for [ … ]
boucle do
```

- Directive sections parallèle

```
#pragma omp parallel sections […]
{
    #pragma omp section
      Bloc 1
    #pragma omp section
      Bloc 2
     …
}
```

ATTRIBUTS DE DONNÉES

Les différents statuts

PRIVATE (list)

SHARED (list)

DEFAULT(PRIVATE | SHARED | NONE)

FIRSTPRIVATE (list)
initialise chaque copie locale par la valeur originale

LASTPRIVATE (list)
le dernier thread met à jour la variable

THREADPRIVATE (list)

Un exemple (1)

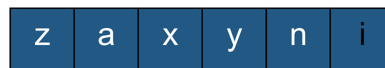
```
void saxpy(z, a, x, y, n){
    int   i, n
    float z[n], a, x[n], y

#pragma omp for
    for(  i = 0 ; i<  n ; ++i) {
        z[i] = a * x[i] + y
    }

}
```

Un exemple (2)

Mémoire partagée Globale



Exécution séquentielle.

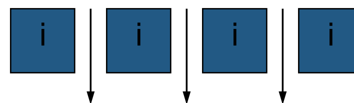
Toutes les données pointent sur la mémoire globale

Mémoire partagée Globale

Exécution parallèle

Chaque thread possède une copie privée de i
Les accès à i vont vers la copie privée

Les accès à z, a, x, y, n pointent vers la mémoire globale.



Un exemple (3)

Division du travail entre les threads

Mémoire partagée globale.

Z(1)		Z(10)	Z(11)		Z(20)	Z(21)		Z(30)	Z(31)		Z(40)	a
X(1)		X(10)	X(11)		X(20)	X(21)		X(30)	X(31)		X(40)	y
												n

```
void saxpy(z, a, x, y, n){
      int   i, n
      float z[n], a, x[n], y
#pragma omp for
      for(  i = 0 ; i<  n ; ++i)
      {
          z[i] = a * x[i] + y
      }
}
```

Mémoire locale

i = 1, 10	i = 11, 20	i = 21, 30	i = 31, 40

n = 40, 4 threads

La clause shared

shared (var)

Tous les threads accèdent à la même zone mémoire de la variable var
- Attention au conflit d'écriture, mise à jour, …

Uniquement possible avec la directive de région parallèle ou des tâches

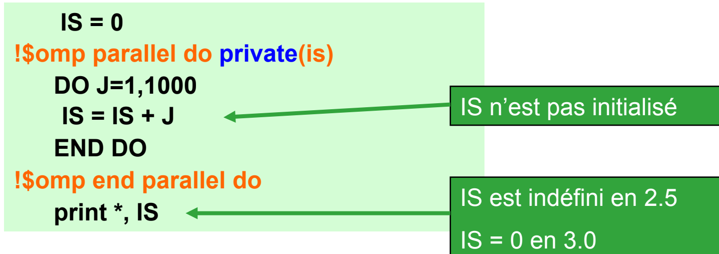
```
Subroutine saxpy(z, a, x, y, n)
integer i, n
real z(n), a, x(n), y
!$omp parallel do shared (z,a,x,y) private(i)
  do i = 1, n
     z(i) = a * x(i) + y
  end do
end
```

La clause private

private (VAR)

Création d’ une copie locale de VAR dans chaque thread :

- La valeur n’est pas initialisée
- Pas de lien entre le stockage de la copie privée et de la variable originale
- En sortie dépend de la version.

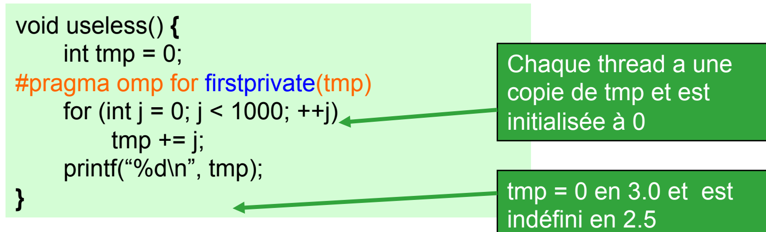


La clause firstprivate

firstprivate (VAR)

Firstprivate est un cas spécial du statut PRIVATE

- Création d’une copie locale dans chaque thread
- Initialisation de chaque copie par la valeur de la variable provenant de la thread maître.



La clause lastprivate

lastprivate (VAR)

LASTPRIVATE est un cas spécial du statut PRIVATE

- En sortie, affecte la valeur de la dernière itération ou section (séquentielle)
- Si NOWAIT; la variable est indéfinie tant qu'une barrière de synchronisation n'a pas été réalisée.

```
void useless() {
    int tmp = 0;
#pragma omp for lastprivate(tmp)
    for (int j = 0; j < 1000; ++j)
        tmp += j;
    printf("%d\n", tmp);
}
```

Chaque thread a une copie de tmp et est initialisée à 0

tmp = valeur pour j = 999

La clause default

default (private | shared | none) Fortran

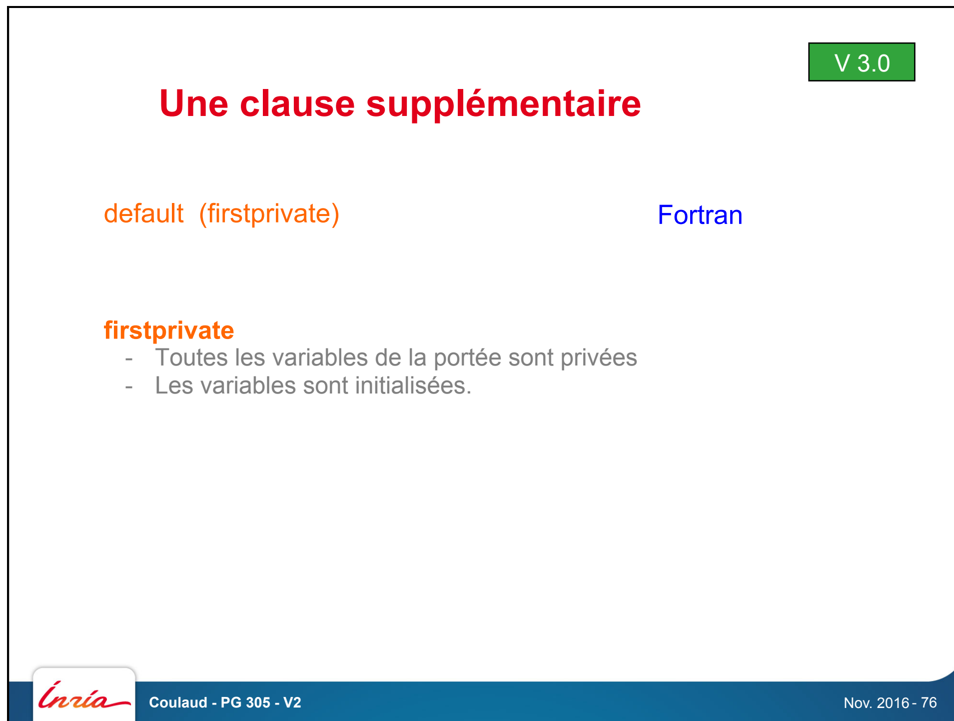
default (shared | none) C/C++

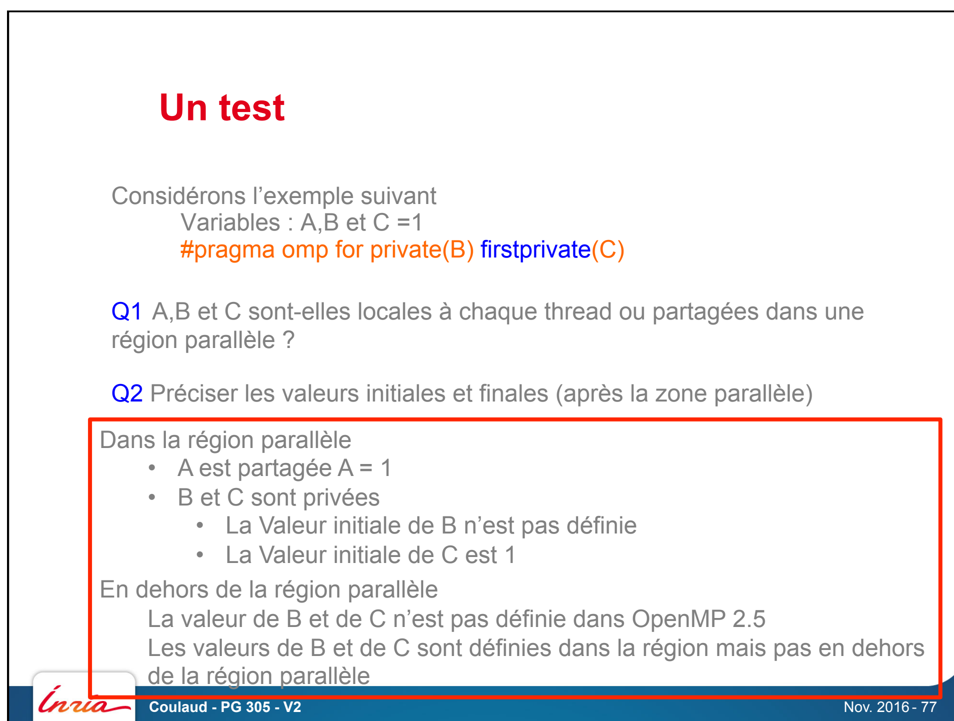
Précise le statut des variables dans la portée.

none
- Pas de statut par défaut
- Il faut préciser le statut de toutes les variables.

shared
- Toutes les variables sont partagées.
- C'est le statut par défaut en l'absence de la clause none.

private
- Toutes les variables de la portée sont privées
- y compris les variables dans les commons (sauf ceux spécifiés dans la clause threadprivate).


V 3.0
Une clause supplémentaire
default (firstprivate)
Fortran
firstprivate
- Toutes les variables de la portée sont privées
- Les variables sont initialisées.
Coulaud - PG 305 - V2
Nov. 2016 - 76


Un test
Considérons l'exemple suivant
Variables : A,B et C =1
#pragma omp for private(B) firstprivate(C)
Q1 A,B et C sont-elles locales à chaque thread ou partagées dans une région parallèle ?
Q2 Préciser les valeurs initiales et finales (après la zone parallèle)
Dans la région parallèle
• A est partagée A = 1
• B et C sont privées
• La Valeur initiale de B n'est pas définie
• La Valeur initiale de C est 1
En dehors de la région parallèle
La valeur de B et de C n'est pas définie dans OpenMP 2.5
Les valeurs de B et de C sont définies dans la région mais pas en dehors de la région parallèle
Coulaud - PG 305 - V2
Nov. 2016 - 77

La clause threadprivate

#pragma omp threadprivate(list)
!$omp threadprivate (list)

Rendre des variables globales privées dans un thread
- Fortran : COMMON blocks, variables des modules
- C/C++ : variable globales, statiques, membre statique d'une classe

Différent de les considérer comme PRIVATE
- Avec PRIVATE les variables globales sont masquées.
- THREADPRIVATE préserve la portée globale à l'intérieur de chaque thread

Les variables threadprivate peuvent être initialisées en utilisant la clause **COPYIN** ou en utilisant l'instruction DATA.

Restrictions
- Le nombre de threads ne doit pas varier (omp_dynamic = false)
- La directive doit suivre la déclaration.

La clause Threadprivate (exemple)

Créer un compteur privé dans chaque thread.

```
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
    counter++;
    return (counter);
}
```

```
#include <omp.h>
int a, b, i, tid;
float x;
#pragma omp threadprivate(a, x)

main () {
     printf("1st Parallel Region:\n");
#pragma omp parallel private(b, tid)
   {  tid = omp_get_thread_num();
   a = tid ;  b = tid;   x = 1.1 * tid +1.0;
    printf("Thread %d: a,b,x= %d %d %f\n",tid,a,b,x);
    } /* end of parallel section */
  printf("************************************\n");
  printf("Master thread doing serial work here\n");
  printf("************************************\n");
  printf("2nd Parallel Region:\n");
#pragma omp parallel private(tid)
  { tid = omp_get_thread_num();
    printf("Thread %d: a,b,x= %d %d %f\n",tid,a,b,x);
  } /* end of parallel section */
}
```

```
% export OMP_NUM_THREADS=4
% ./a.out
1st Parallel Region:
Thread 0: a,b,x= 0 0 1.000000
Thread 1: a,b,x= 1 1 2.100000
Thread 2: a,b,x= 2 2 3.200000
Thread 3: a,b,x= 3 3 4.300000
************************************
Master thread doing serial work here
************************************
2nd Parallel Region:
Thread 0: a,b,x= 0 0 1.000000
Thread 3: a,b,x= 3 0 4.300000
Thread 1: a,b,x= 1 0 2.100000
Thread 2: a,b,x= 2 0 3.200000
```

La clause threadprivate (exemple)

Considérons des procédures appelées chacune dans un thread dans une région parallèle.

```
      subroutine poo
      parameter (N=1000)
      common/buf/A(N),B(N)
C$OMP THREADPRIVATE(/buf/)
      do i=1, N
        B(i)= const* A(i)
      end do
      return
      end
```

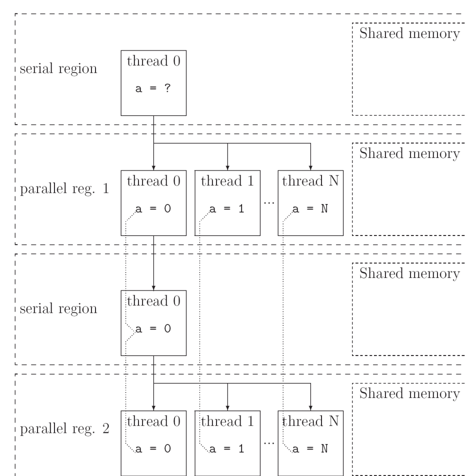
```
      subroutine bar
      parameter (N=1000)
      common/buf/A(N),B(N)
C$OMP THREADPRIVATE(/buf/)
      do i=1, N
        A(i) = sqrt(B(i))
      end do
      return
      end
```

À cause de la directive threadprivate, chaque thread exécutant ces procédures ont leur propre copie du bloc common /buf/.

La clause Threadprivate (exemple 2)

```
integer, save :: a
!$OMP THREADPRIVATE(a)

!$OMP PARALLEL
        a = OMP_get_thread_num()
!$OMP END PARALLEL
….
!$OMP PARALLEL
...
!$OMP END PARALLEL
```



V 3.0

C++: Threadprivate

- création
- Autorise dans les classes C++ un membre static à être privé

```
class T {
    public:
    static int i;
    #pragma omp threadprivate(i)
    ...
};
```

La clause copyin

copyin (list)

donne un moyen d'assigner la même valeur aux variables threadprivate pour tous les threads.

La variable du thread master est utilisée comme valeur à copier

Fortran : la liste contient le nom des variables et des commons

Les commons privés (thread) sont initialisés par la valeur du thread maître.

La clause copyprivate

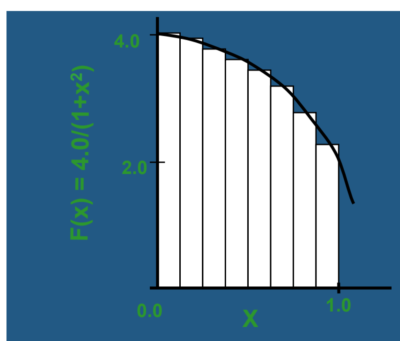
copyprivate(list)

Pour diffuser une variable privée dans une variable globale ou un pointer

- list ne doit pas avoir le statut de PRIVATE, FIRSTPRIVATE
- uniquement après la directive SINGLE

```
float x, y;
#pragma omp threadprivate(x, y)
void init(float a, float b)
{
   #pragma omp single copyprivate(a,b,x,y)
   {
      get_values(a,b,x,y);
   }
}
```


Exercice 2 : programme pi



$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

On approche l'intégrale par la somme des aires des rectangles :

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

où chaque rectangle a une largeur Δx et une hauteur $F(x_i)$ évaluée au milieu de d'intervalle i.

Indication : chaque thread évalue un bout de la somme en parallèle

Double SUM[NUM_THREAD_MAX];

On somme en séquentiel les contributions.

La clause reduction

Une autre clause qui affecte comment une variable est partagée

reduction (op : list)

Les variables dans "list" doivent être partagées dans le région parallèle.
Les opérateurs possibles sont :

FORTRAN : +, *, -, .AND., .EQV., .NEQV., MAX, MIN, IAND, IOR, IEOR
C/C++ : +, *, -, &, |, ^, &&, max, min

Dans une région parallèle :

- Une copie locale des variables de la liste est faite et initialisée en fonction de l'opérateur op de la réduction. (ex.. 0 for "+").
- Les copies locales sont réduites en une seule valeur et combinée avec la valeur de la variable globale (originale).

La clause reduction

Restriction :

- List ne doit pas contenir des tableaux dans la version 1.0 (accepté en 2.0)
- Les copies locales sont réduites en une seule valeur et combinée avec la valeur de la variable globale (originale).

Exemple:

```
!$OMP DO REDUCTION(+: A, Y) REDUCTION(.OR.: AM)
```

Statut des données
La clause reduction (exemple)

```
PROGRAM DOT_PRODUCT

INTEGER N, CHUNK, I
 PARAMETER (N=100)
PARAMETER (CHUNK=10)
REAL A(N), B(N), RESULT
! Some initializations

!$OMP  PARALLEL DO DEFAULT(SHARED) &
!$OMP& PRIVATE(I)                  &
!$OMP& SCHEDULE(STATIC,CHUNK)
DO I = 1, N
    A(I) = I * 1.0
    B(I) = I * 2.0
ENDDO
!$OMP END DO

RESULT= 0.0
```

```
!$OMP PARALLEL DO                          &
!$OMP& DEFAULT(SHARED)  PRIVATE(I)&
!$OMP& SCHEDULE(STATIC,CHUNK)       &
!$OMP& REDUCTION(+:RESULT)
  DO I = 1, N
      RESULT = RESULT + (A(I) * B(I))
  ENDDO
!$OMP END DO

PRINT *, 'Final Result= ', RESULT
END PROGRAM DOT_PRODUCT
```

```
#include <omp.h>
main () {
 int i, n, chunk;
float a[100], b[100], result;
/* Some initializations */
     n = 100; chunk = 10;
     result = 0.0;
     for (i=0; i < n; i++)
        {a[i] = i * 1.0; b[i] = i * 2.0; }
 #pragma omp parallel for default(shared) private(i) schedule(static,chunk) \
     reduction(+:result)
     for (i=0; i < n; i++)
        result = result + (a[i] * b[i]);
printf("Final result= %f\n",result);
 }
```

Résumé des clauses

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	●				●	●
PRIVATE	●	●	●	●	●	●
SHARED	●	●			●	●
DEFAULT	●				●	●
FIRSTPRIVATE	●	●	●	●	●	●
LASTPRIVATE		●	●		●	●
REDUCTION	●	●	●		●	●
COPYIN	●				●	●
COPYPRIVATE				●		
SCHEDULE		●			●	
ORDERED		●			●	
NOWAIT		●	●	●		

L' API OPENMP

Les variables d'environnement

Fixe le nombre de threads à utiliser
OMP_NUM_THREADS *int_literal*

Autorise d'utiliser un nombre de thread différents dans chaque région ?
OMP_DYNAMIC TRUE || FALSE

Précise si l'on souhaite faire du parallélisme emboîté avec une nouvelle équipe de thread ou non
OMP_NESTED TRUE || FALSE

Contrôle comment OpenMP ordonnance pour la clause schedule (RUNTIME) le travail partagé.
OMP_SCHEDULE "schedule[, chunk_size]"

setenv OMP_SCHEDULE "guided,4"

V 3.0

Les variables d'environnement

OMP_PROC_BIND : permet de fixer un thread sur le processeur.
OMP_PROC_BIND true/false

OMP_STACKSIZE : contrôle la taille de la pile pour les threads créés (sauf le master)
OMP_STACKSIZE n [B,K,M,G]

OMP_WAIT_POLICY : permet de préciser la politique d'attente des threads
OMP_WAIT_POLICY active/passive

OMP_MAX_ACTIVE_LEVELS : contrôle le nombre maximal de parallélisme emboitée dans une région parallèle. La valeur est un entier positif.

OMP_THREAD_LIMIT : Fixe le nombre de threads utilisé dans le programme

L'API OpenMP
Les fonctions de la bibliothèque

1. OMP_SET_NUM_THREADS
2. OMP_GET_NUM_THREADS
3. OMP_GET_MAX_THREADS
4. OMP_GET_THREAD_NUM
5. OMP_GET_THREAD_LIMIT
6. OMP_GET_NUM_PROCS
7. OMP_IN_PARALLEL
8. OMP_SET_DYNAMIC
9. OMP_GET_DYNAMIC
10. OMP_SET_NESTED
11. OMP_GET_NESTED
12. OMP_SET_SCHEDULE
13. OMP_GET_SCHEDULE
14. OMP_SET_MAX_ACTIVE_LEVELS
15. OMP_GET_MAX_ACTIVE_LEVELS
16. OMP_GET_LEVEL
17. OMP_GET_ANCESTOR_THREAD_NUM
18. OMP_GET_TEAM_SIZE
19. OMP_GET_ACTIVE_LEVEL
20. OMP_INIT_LOCK
21. OMP_DESTROY_LOCK
22. OMP_SET_LOCK
23. OMP_UNSET_LOCK
24. OMP_TEST_LOCK
25. OMP_INIT_NEST_LOCK
26. OMP_DESTROY_NEST_LOCK
27. OMP_SET_NEST_LOCK
28. OMP_UNSET_NEST_LOCK
29. OMP_TEST_NEST_LOCK
30. OMP_GET_WTIME
31. OMP_GET_WTICK

L'API OpenMP
Les fonctions de la bibliothèque

Fonctions de l'API

- Tester/ modifier le nombre de threads
  omp_set_num_threads(), omp_get_num_threads(),
  omp_get_thread_num(), omp_get_max_threads()
- Modifie le mode dynamique i.e le nombre de threads peut varier entre deux constructions parallèles
  omp_set_dynamic(), omp_get_dynamic()
- Modifie le parallélisme emboité
  omp_set_nested(), omp_get_nested(),
- Sommes nous dans une région parallèle ?
  omp_in_parallel()
- Combien de processeurs dans le système ?
  omp_num_procs()

L' API OpenMP
Les fonctions de la bibliothèque

Fonctions sur les verrous

omp_init_lock(), omp_init_nest_lock(),
omp_destroy_lock(), omp_destroy_nest_lock(),
omp_set_lock(), omp_set_nest_lock(),
omp_unset_lock(), omp_unset_nest_lock(),
omp_test_lock(), omp_test_nest_lock()

Fonction pour mesurer le temps

omp_get_wtime(), omp_get_wtick()

```
double start;
double end;
start = omp_get_wtime();
... work to be timed ...
end = omp_get_wtime();
printf_s("Work took %f sec. time.\n", end-start);
```

SYNCHRONISATION

OpenMP: Synchronization

Différentes directives permettent de synchroniser les

Haut niveau
- Barrier
- critical
- atomic
- ordered

Bas niveau
- flush
- lock

La directive barrier

!$omp barrier **#pragma omp barrier**

Le thread attend jusqu'à ce que toute l'équipe arrive sur la directive

```
#pragma omp parallel shared (A, B, C) private(id)
{
    id=omp_get_thread_num();
    A[id] = big_calc1(id);
#pragma omp barrier
#pragma omp for
    for(i=0;i<N;i++){C[i]=big_calc3(I,A);}
#pragma omp for nowait
    for(i=0;i<N;i++){ B[i]=big_calc2(C,  i); }

    A[id] = big_calc3(id);
}
```

Barrière implicite à la fin de bloc de la directive for

Pas de barrière implicite à cause de **nowait**

Barrière implicite à la fin de la région parallèle

La directive critical

```
!$omp critical [(nom)]
    bloc d'instructions
!$omp end critical [(nom)]
```

```
#pragma omp critical [(name)]
    {bloc structuré}
```

Empêcher l'accès simultané pas les threads à un bloc d'instruction

Un thread à la fois dans le bloc d'instructions parmi toutes les threads du programme (pas de l'équipe!)

La section critique

- On peut la nommer
  - Le nom est une entité globale
  - Ne doit pas entrer en conflit avec le nom d'une procédure, common, …
- Un thread à la fois dans la section

La directive critical (exemple)

Accumulation dans une boucle

```
!$omp parallel do shared(A) private(ALOCAL)
    ……
!$OMP CRITICAL (left)
    A(index(i)) = A(index(i)) + Alocal
!$OMP END CRITICAL (left)
    ……
!$omp end parallel
```

```
#pragma omp for shared(A) private(Alocal)
 for (…) {
    ……
#pragma omp critical (left)
    A[index[i]] =  A[index[i]] + Alocal
……
}
```

La directive atomic

La directive assure qu'une variable partagée est lue et modifiée en mémoire par un seul thread à la fois

Ne s'applique qu'à l'instruction suivant la directive

```
#pragma omp atomic [read | write | update Í capture]
    {bloc structuré}
```

Clauses :

read v = x

write x = v

update (défaut) x = x binop operation (++x, x--, …)

capture fait un atomic update de x + capture la valeur d'origine ou finale

La directive atomic (exemple)

```
#pragma omp parallel
{
    double tmp, B;
    B    = DOIT();
    tmp = other_work(B);

#pragma omp atomic update
     x  +=  tmp ;

}
```

fournit l'exclusion mutuelle mais s'applique uniquement à la mise à jour d'un emplacement mémoire

La directive ordered

Les itérations d'une boucle seront exécutées dans le même ordre qu'en séquentiel.

```
!$omp ordered
    bloc
!$omp end ordered
```

```
#pragma omp ordered
   {  bloc  }
```

Restriction

- Uniquement dans un bloc DO/for
- La directive DO/for doit aussi avoir la clause ordered

```
#pragma omp parallel private (tmp)
#pragma omp for ordered
     for (I=0;I<N;I++){
          tmp = NEAT_STUFF(I);
#pragma ordered
          res += consum(tmp);
     }
```

```
!$omp parallel do shared(A) ordered
do i = 1, 1000
!$omp ordered
     A(i) = 2 * A(i-1)
!$omp end ordered
enddo
!$omp end parallel  do
```

La directive flush

Permet à un thread de construire une vue cohérente de la mémoire

```
!$omp flush (list)
```

```
#pragma omp flush (list)
```

Après cet appel

- Toutes les opérations mémoires sont terminées
- Variables dans les registres, buffers d'écriture doivent être mis à jour en mémoire ;

Autorise une mécanisme de synchronisation point à point

La dernière valeur de la variable est visible pour tous les threads

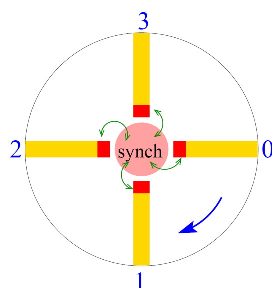
La directive flush (exemple)

Permet de synchroniser deux threads, la variable ISYNC doit être partagée.

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(ISYNC)
IAM      = OMP_GET_THREAD_NUM() ;  ISYNC(IAM) = 0
NEIGH = GET_NEIGHBOR (IAM)
!$OMP BARRIER
CALL WORK()
C I am done with my work, synchronize with my neighbor
ISYNC(IAM) = 1
!$OMP FLUSH(ISYNC)
C Wait until neighbor is done
DO WHILE ( ISYNC(NEIGH) .EQ. 0)
!$OMP FLUSH(ISYNC)
END DO
!$OMP END PARALLEL
```

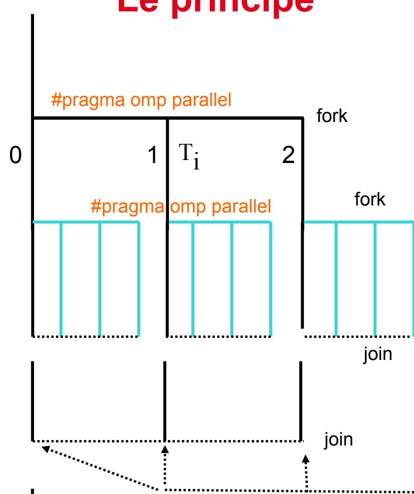

Exercice avancé

Faire circuler un jeton sur un anneau.



LE PARALLÉLISME EMBOÎTÉ

Parallélisme emboîté
Le principe



Chaque thread rencontre une région parallèle

- Autoriser le parallélisme emboîté
  setenv OMP_NESTED TRUE
  Utiliser **omp_set_nested()**
- Sinon la partie est sérialisée (1 thread)

Création des nouveaux threads

- thread T_i devient master
- master + slaves = team
- à la fin les slaves sont en mode sleep ou spin
  - Dépend de **OMP_WAIT_POLICY**

Parallélisme emboîté
Les restrictions

Attention il faut imbriquer des régions parallèles et pas du partage de travail.

```
#pragma omp parallel
{
 #pragma omp parallel
         {  bloc 2}
}
```

```
#pragma omp parallel
#pragma omp for
{
 #pragma omp for
         {  bloc 2}
}
```

Directives interdites :

- barrier, master, single, ordered
- critical avec le même nom

Parallélisme emboîté
Un exemple

```
program parallel
implicit none
integer :: rang, OMP_GET_THREAD_NUM
! $OMP PARALLEL NUM_THREADS(3)  PRIVATE(rang)
    rang= OMP_GET_THREAD_NUM ()
    print *,"Mon rang dans region 1 :",rang
! $OMP PARALLEL NUM_THREADS(2) PRIVATE(rang)
   rang= OMP_GET_THREAD_NUM ()
print *," Mon rang dans region 2 :",rang
!$OMP END PARALLEL
!$OMP END PARALLEL
end program parallel
```

Activé que si OMP_NESTED = TRUE

Exemple

```
#include <omp.h>
int main() {
  int rang = -1, rang2 = -1 ;
  omp_set_nested(1);
#pragma omp parallel default(none) num_threads(2) private(rang,rang2)
  {
    rang = omp_get_thread_num() ;
#pragma omp parallel default(none) num_threads(3) private(rang2) ,shared(rang)
    {
      rang2 = omp_get_thread_num() ;
      printf("Mon rang dans region 1 est : %d  dans la region 2 est %d \n",rang, rang2) ;
    }
  }
  return 0 ;
}
```

```
$ ./nested
Mon rang dans la region 1 est : 0 et  dans la region 2 est 0
Mon rang dans la region 1 est : 0 et  dans la region 2 est 1
Mon rang dans la region 1 est : 0 et  dans la region 2 est 2
Mon rang dans la region 1 est : 1 et  dans la region 2 est 0
Mon rang dans la region 1 est : 1 et  dans la region 2 est 1
Mon rang dans la region 1 est : 1 et  dans la region 2 est 2
```

Autres exemples (1)

```
#pragma omp parallel default(shared)
{
 #pragma omp for
     for (i=0; i<n; i++) {
 #pragma omp parallel shared(i, n)
           {
#pragma omp for
            for (j=0; j<n; j++)
                 work(i, j);
           }
     }
 }
```

Autres exemples (2)

```
#pragma omp parallel default(none) num_threads(2) private(rang)
 {
   rang = omp_get_thread_num() ;
    printf("Mon rang dans region 1 est : %d \n",rang) ;
#pragma omp parallel default(none) num_threads(3) private(rang)
   {
    rang = omp_get_thread_num() ;
    printf("Mon rang dans la region 2 est %d \n",rang) ;
   }
 }
```

$ ./nested_1

Mon rang dans region 1 est : 0
Mon rang dans la region 2 est 0
Mon rang dans region 1 est : 1
Mon rang dans la region 2 est 1
Mon rang dans la region 2 est 2
Mon rang dans la region 2 est 0
Mon rang dans la region 2 est 1
Mon rang dans la region 2 est 2

Quelques méthodes utiles (1)

Fixer le nombre de threads par niveau

- Variable d'environnement : OMP_NUM_THREADS
- fonction: omp_set_num_threads() dans une région parallèle
- Clause : num_threads(10)

Fixer/obtenir le nombre de threads dans le programme

- Variable d'environnement : OMP_THREAD_LIMIT
- fonction: omp_get_thread_imit() pour connaître le nombre de threads disponible

Quelques méthodes utiles (2)

Set/Get le nombre maximal de niveau de régions parallèles emboitées
- Variable d'environnement : OMP_MAX_ACTIVE_LEVELS
- Fonctions omp_set_max_active_levels(), omp_get_max_active_levels()

Fonctions de la bibliothèques pour déterminer
- La profondeur : omp_get_active_level() (1, 2, …, $level_{max}$)
- Id du père : omp_get_ancestor_thread_num(level)
- La taille de l'équipe d'un niveau level: omp_get_team_size(level)

CONSTRUCTION DE TÂCHES

Les tâches OpenMP

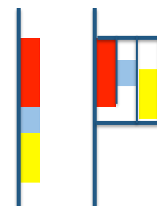
Principale nouveauté avec OpenMP 3.0

Offre un modèle flexible pour les problèmes irréguliers
- Boucles non bornées (boucle while) ;
- Algorithmes récursifs ;
- Schémas producteurs/consommateurs

Les tâches OpenMP

Les tâches OpenMP sont des unités de travail indépendantes
- Les threads sont affectés pour effectuer le travail des tâches ;
- L'exécution des tâches peut être différée ;
- L'exécution des tâches peut être immédiate ;

le système d'exécution décide si l'exécution est différée ou immédiate.



Une tâche est constituée
- D'un code à exécuter ;
- D'un environnement de données initialisées à la création ;
- De variables de contrôle interne (ICV).

Construction de tâches

C/C++:

```
#pragma omp task [clause [[,]clause] ...]
    structured-block
```

Fortran:

```
!$omp task [clause [[,]clause] ...]
    structured-block
!$omp end task
```

Permet de construire une tâche qui sera exécutée par un thread du pool de threads.

On peut imbriquer le constructeur (parallélisme emboité)

Construction de tâches

C/C++:

```
#pragma omp task [clause [[,]clause] ...]
    structured-block
```

Clauses :

- if (expression scalaire)
- final (expression scalaire)
- untied
- default (shared | none)
- mergeable
- firstprivate (list)
- private (list)
- shared (list)

Fortran:

```
!$omp task [clause [[,]clause] ...]
    structured-block
!$omp end task
```

Clauses :

- if (expression scalaire)
- final (expression scalaire)
- untied
- default (shared | private | firstprivate | none)
- mergeable
- firstprivate (list)
- private (list)
- shared (list)

Exemple : Hello world (1)

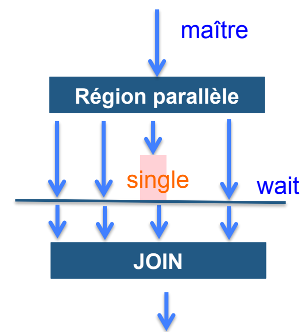
```
int main() {
#pragma omp parallel
 {
#pragma omp single
  {

     {  printf("Hello "); }

     { printf("World "); }

    printf("\nThank You ");
  } // End of single region
 } // End of parallel region
printf("\n");
 return(0);
}
```

```
$ export OMP_NUM_THREADS=2
$ ./task_hello
Hello World
Thank You
```



Exemple : Hello world (2)

```
int main() {
#pragma omp parallel
```

```
$./task_hello_1

Thank You World Hello
$./task_hello_1

Thank You World Hello
$ ./task_hello_1

Thank You Hello World
```

```
printf("\n");
 return(0);
}
```

```
int main() {
#pragma omp parallel
 {
#pragma omp single
  {
#pragma omp task
     {  printf("Hello "); }
#pragma omp task
     { printf("World "); }

    printf("\nThank You ");
  } // End of single region
 } // End of parallel region
printf("\n");
 return(0);
}
```

Exemple : Hello world (3)

```
int main() {
#pragma omp parallel
 {
#pragma omp single
  {
#pragma omp task
     {  printf("Hello "); }
#pragma omp task
     { printf("World "); }
#pragma omp taskwait
    printf("\nThank You ");
  } // End of single region
 } // End of parallel region
printf("\n");
 return(0);
}
```

```
 $./task_hello_2
World Hello
Thank You
```

Exemple : liste chainée

```
node *p  = listhead ;
while (p) {
      do_independent_work(p) ;
      p = p->next() ;
}
```

Difficile de le faire avant OpenMP 3.0
- Compter le nombre d'itérations
- Transformer en une boucle finie *for*

Exemple : liste chainée

```
node *p   = listhead ;
#pragma omp parallel
 {
#pragma omp single
  {
    while (p) {
    #pragma omp task  firstprivate (p)
          {
          do_independent_work(p) ;
          }
          p = p->next()
    }
  } // END SINGLE
}   // END PARALLEL
```

- #pragma omp parallel ← Création des threads
- #pragma omp single ← Un thread exécute la boucle while
- #pragma omp task firstprivate (p) ← Création des tâches et exécution en parallèle
- do_independent_work(p) ; ← Chaque tâche s'exécute dans un thread
- } // END SINGLE ← Quand la tache se termine, le thread associé attend sur la barrière implicite de la construction ***single***

Terminaison

Où et quand les tâches sont elles terminées ?

- sur les barrières implicites (fin de région parallèle, …)
- sur les barrières explicites
  #pragma omp barrier

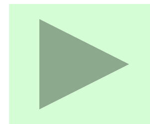
  S'applique à toutes les tâches générées dans la région parallèle
- Sur les barrières des tâches
  #pragma omp taskwait
  Attend jusqu'à ce que toutes les tâches définies par le constructeur soient terminées.
  Ne s'applique pas aux descendants

Exercice : le tri rapide récursif

Paralléliser l'algorithme du Quicksort écrit de manière récursive.

Code séquentiel disponible dans la section OpenMP

http://people.bordeaux.inria.fr/coulaud/Enseignement/PG305



Bilan

Simple et facile à mettre en œuvre

Pour avoir de la performance

- Ne pas créer trop de threads
- Temps de calcul grand > temps de création
- Equilibrer le travail dans les threads

→ Implique de revoir l'algorithme.

Exercice : Les nombres de Fibonacci

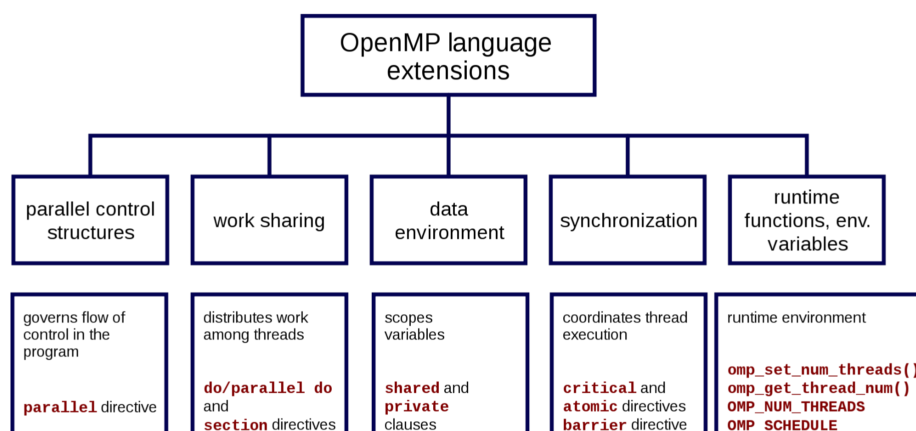
Les nombres de Fibonacci sont définis comme suit

F(0) = 1
F(1) = 1

F(n) = F(n-1) + F(n-2) (n=2,3, …)

Suite :
1, 1, 2, 3, 5, 8, 13, 21, 34

OpenMP en 1 slide



Conclusion (1)

Il est facile d'insérer des directives OpenMP

Toutefois, pour avoir de bonnes performances
- Le coût des synchronisations doit être réduit
- La localité des données doit être optimisée à tous les niveaux

Le style SPMD style conduit à de bonnes performances
- Mais demande beaucoup d'effort à programmer

OpenMP a la flexibilité pour permettre les deux sortes de programmation

Conclusion (2)

- Les plus
  - Facilité de programmation
  - Parallélisation incrémentale
  - Haut niveau d'abstraction
  - Bonne performance (modèle SPMD)
- Les moins
  - Des manques (aspect NUMA thread ou data affinity)
  - Manque de performances sur les tâches
  - Pas d'information sur ce que fait le runtime
- Pour plus de détails : http://www.openmp.org

Quelques nouveautés

OPENMP 4.0

Les changements

1. Prise en compte des accélérateurs
2. Extensions SIMD
3. Placement et affinité
4. Taskgroup et dépendances
5. Gestion des erreurs
6. Réduction utilisateur
7. ….

SIMD

Utile pour faire de la vectorisation automatique

Compilateur ne peut pas vectoriser
- une boucle si
  - Boucle complexe
  - Possède des dépendances
- une fonction « inlinée »

→Jeu d'instructions SIMD pour aller plus vite

Boucle SIMD

Permet de transformer une boucle en boucle SIMD
→Jeu d'instructions SIMD

#pragma omp simd [clause [[,]clause] ...]
for-loops

Les clauses

safelen(length) **linear**(list[:linear-step])
aligned(list[:alignment]) (8,16,32 ou 64)
aligne les objets de la liste au nombre de bytes précisé. Si absent dépend de l'implémentation et de la machine
collapse(n)
private(list) **lastprivate**(list)
reduction(reduction-identifier:list)

Exemple

```
void star( double *a, double *b, double *c, int n, int *ioff )
{
      #pragma omp simd
      for ( int i = 0; i < n; i++) {
            a[i] *= b[i] * c[i+ *ioff];
      }
}
```

A la compilation :

- ioff n'est pas connu → compilateur suppose que ioff peut être ≤0 ou >0
- a, b et c sont peut être des alias

Remarque si a et c sont des alias et ioff = 2 → dépendance arrière
→ Pas de vectorisation

Pragma force la vectorisation

Clause safelen(N)

Précise au compilateur qu'il n'y a pas de dépendance pour un vecteur de taille N ou en dessous.

Si la clause **safelen** n'est pas précisée N = le nombre d'itérations

Exemple

```
void work( float *b, int n, int m ) {
   int i;
   #pragma omp simd safelen(16)
   for (i = m; i < n; i++) {
      b[i] = b[i-m] - 1.0;
   }
}
```

Précise que la boucle est sûre sur une longueur de 16 (incluse)

```
b[m]    = b[0] - 1.0
b[m+1] = b[1] - 1.0
…
b[m+n] = b[n-m] - 1.0
```

si m ≥ 16 code correct
si m < 16 comportement non défini

Fonction SIMD

Autorise la création d'une ou plusieurs versions de la fonction qui peut contenir des arguments différents avec des instructions SIMD pour être appelée dans une boucle SIMD

#pragma omp declare simd [clause[[,] clause] ...]
Définition ou déclaration de la fonction

Les clauses

simdlen(length) aligned(argument-list])
linear(argument-list[:constant-linear-step])
uniform(argument-list)
inbranch notinbranch

Clause uniform et linear

Clause **uniform(val)** précise que val est constant dans tous les appels concurrents

Clause **linear(var:step)** précise que pour chaque itération de la boucle scalaire var est augmenté de step. Défaut **linear(var) ←→linear(var:1)**

```
int main(int argc, char *argv[])
{
int i, k;
float a[1024], b[1024];
…
float op2 = cst
    #pragma omp simd
    for (k=0; k<N; k++) {
        b[k] = fSqrtMul(&a[k], op2);
    }
}
```

```
#pragam omp declare simd linear(op1)uniform(op2)
float fSqrtMul(float *op1, float op2) {
    return sqrt(*op1)*sqrt(op2);
}
```

a | a[0] | a[1] | a[2] | | | | | |

Construction de Boucle parallèle SIMD

Permet de spécifier qu'une boucle peut être exécutée en parallèle avec des instructions SIMD

```
#pragma omp for simd [clause [[,]clause] ...]
    for-loops
```

Les clauses sont les mêmes que pour les constructions for et simd.

Étapes

1. Distribution des itérations sur les threads (tâches implicites)
2. Paquets d'instructions sont convertis en instruction SIMD

Nouveautés dans les tâches

C/C++:

```
#pragma omp task [clause [[,]clause] ...]
    structured-block
```

Clauses :
- if (expression scalaire)
- final (expression scalaire)
- untied
- default (shared | private | firstprivate | none)
- mergeable
- private (list)
- firstprivate (list)
- shared (list)
- depend (list)

Quelques remarques

Une tâche est attachée à un thread.
- la tâche génératrice peut être détachée via untied
- Peut aussi limiter le problème de deadlock

final (expr) : précise que la tâche générée est finale et « included tasks » et que toutes les tâches crées par cette tâche le sont aussi.

- *included tasks* = une tâche qui est séquentielle
- Très utile dans une fonction récursive pour limiter la profondeur de la récursion

La clause depend

Permet de préciser les dépendances entre les tâches
- → conduit à des contraintes sur l'ordonnancement des tâches

Permet de spécifier l'ordre d'exécution des tâches

depend (dependence-type **:** list**)**

avec dependence-type = in, out, inout
- in : la tâche dépend de toutes les autres tâches de même parent ayant la variable en out ou inout
- inout, out : toutes les autres tâches de même parent précédemment produites qui font référence à au moins une des variables avec une dépendance de type **in**, **out**, or **inout**

Exemple (1)

```
#include <stdio.h>
int main() {
    int x;
    #pragma omp parallel
     #pragma omp single
    {
    #pragma omp task shared(x) depend(out: x)
    x = 1;
    #pragma omp task shared(x) depend(out: x)
    x = 2;
    #pragma omp taskwait
    printf("x = %d.\n ", x);
    }
return 0;
}
```

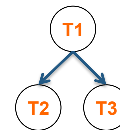
Le programme affiche toujours 2.

Force l'ordre d'exécution des tâches

Si pas de clause ***depend*** l'affichage est soit 1 soit 2

Exemple (2)

```
#include <stdio.h>
int main() { i
    ntx= 1;
    #pragma omp parallel
     #pragma omp single
    {
    #pragma omp task shared(x) depend(out: x)
    x = 2;
    #pragma omp task shared(x) depend(in: x)
    printf("x + 1 = %d. ", x+1);
    #pragma omp task shared(x) depend(in: x)
    printf("x + 2 = %d\n", x+2);
    }
return 0;
}
```



Pas d'ordre sur T2 et T3

% ex2

x + 1 = 3. x + 2 = 4

% ex2

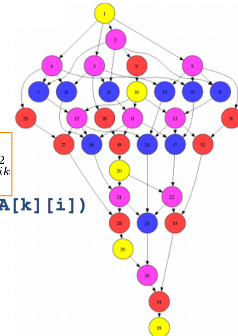
x + 2 = 4

x + 1 = 3.

Cholesky

```
void blocked_cholesky( int NB, float A[NB][NB] ) {
   int i, j, k;
   for (k=0; k<NB; k++) {
     #pragma omp task depend(inout:A[k][k])
        spotrf (A[k][k]) ;
     for (i=k+1; i<NT; i++)
       #pragma omp task depend(in:A[k][k]) depend(inout:A[k][i])
          strsm (A[k][k], A[k][i]);
       // update trailing submatrix
       for (i=k+1; i<NT; i++) {
         for (j=k+1; j<i; j++)
           #pragma omp task depend(in:A[k][i],A[k][j])
                            depend(inout:A[j][i])
              sgemm( A[k][i], A[k][j], A[j][i]);
         #pragma omp task depend(in:A[k][i]) depend(inout:A[i][i])
            ssyrk (A[k][i], A[i][i]);
       }
   }
}
```

$$L_{ii} = \sqrt{A_{ii} - \sum_{k=1}^{i-1} L_{ik}^2}$$



$$L_{ji} = \frac{A_{ij} - \sum_{k=1}^{i-1} L_{ik} L_{jk}}{L_{ii}}$$

Terminaison

#pragma omp taskwait

- spécifie une attente sur la terminaison des tâches produites dans la tâche actuelle, pas aux descendants

#pragma omp barrier

- spécifie une attente à toutes les tâches générées dans la région parallèle actuelle jusqu'à la barrière

#pragma omp taskgroup

- spécifie une attente sur la terminaison des tâches filles de la tâche actuelle et leurs tâches de descendantes

Construction taskyield

taskyield : spécifie que la tâche peut être suspendue au profit d'une autre tâche

```
void foo ( omp_lock_t * lock, int n ) {
  int i;
  for ( i = 0; i < n; i++ )
    #pragma omp task
    {
     something_useful();
     while ( !omp_test_lock(lock) ) {
           #pragma omp taskyield }
     something_critical();
     omp_unset_lock(lock); }
}
```

On test si le verrou est libre si oui on le prend pour faire la section critique

On suspend la tâche au profit d'une autre tâche

Le placement (1)

Fixer un thread sur un cœur (in OpenMP 3.1)

export OMP_PROC_BIND=true/false

Nouvelles extensions pour le placement des threads

1. Plus de possibilités pour OMP_PROC_BIND

   true, false, master, close ou spread

   Pour spécifier comment les tâches (implicites) sont assignées

   - master : affecte les threads de l'équipe à la même place que le thread master
   - close : affecte les threads de l'équipe proche de la place du thread parent
   - spread : étale les threads sur les emplacements disponibles

Le placement (2)

2. Variable d'environnement OMP_PLACES pour placer les threads
   - par un nom abstrait : threads, cores et sockets
   - par un ordre explicit

3. Ajout d'une nouvelle close pour la construction d'une région parallèle
   proc_bind (master | close | spread)

```
#pragma omp parallel proc_bind(spread) num_threads(4)
{
	work();
}
```


TP openMP

Ecrire la version parallèle OpenMP du produit matrice – vecteur dont la version séquentielle est :

```
For (i=0 ;i<n ;i++) {
	C[i]=0 ;
	For (j=0 ;j<n ;j++)
		C[i]+=A[i][j]*B[j] ;
}
```

1. Ecrire le produit par ligne
2. Ecrire le produit par bloc
3. La directive collapse peut-elle fonctionner ? Expliquer pourquoi.

Remerciements

Documents utilisés pour ces transparents

Cours de l'IDRIS
Rudolf Eigenmann, Tim Mattson : tutorial OpenMP à SC'2001
Miguel Hermanns : Parallel Programming in Fortran 95 using OpenMP
Tim Mattson et Larry Meadows : A "Hands-on" Introduction to OpenMP
SC'2008

....