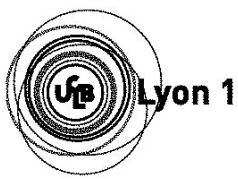
Introduction à OpenMP

Anne Cadiou

Laboratoire de Mécanique des Fluides et d'Acoustique

Ateliers et Séminaires Pour l'Informatique et le Calcul Scientifique
PMCS2I - LMFA
Vendredi 13 mars 2020



Introduction

Définition

- **Open M**ulti-**P**rocessing
- Interface de programmation (API) pour la parallélisation multi-tâches, initialement pour des architectures à mémoire partagée, cible aussi aujourd'hui les accélérateurs, les systèmes temps réel. Elle contient :
  - de directives pour les langages C, C++, Fortran
  - une bibliothèque de routines spécifiques
  - des variables d'environnement
- Supporté par de nombreux OS

OpenMP

- Repose sur l'utilisation de processus système légers (threads)
- Les threads accèdent à un espace mémoire commun, ce qui limite la redondance des données, facilite les communications
- Basé sur le principe du fork and join

Principe

- Un programme OpenMP est composé d'une succession de régions parallèles et de régions séquentielles
- Un programme OpenMP s'exécute sur un seul processus.
  - En entrant dans une zone parallèle, ce processus active plusieurs sous-tâches (ou processus légers, les threads). Les threads peuvent être exécutés sur des coeurs distincts d'un même processeur
  - Une région séquentielle est exécutée par une seule tâche (la tâche maître)
- Le nombre de tâches peut varier d'un région parallèle à une autre
- Rien n'est spécifié au niveau des E/S

- Chaque processus léger exécute sa propre séquence d'instruction
- L'ordre d'exécution des processus est choisit par l'OS
- Chaque thread partage la mémoire avec la tâche initiale, mais dispose aussi d'une mémoire locale (la pile, ou stack). Il est donc possible de définir des variables partagées ou privées (dans la pile)
- Il est parfois nécessaire d'introduire une synchronisation entre les sous-tâches (par exemple lors d'opérations de réduction)

Historique

Architectures SMP (Shared Memory multiProcessor). Dans les années 90, chaque constructeur (CRAY, NEC, IBM, ...) avait son propre jeu de directives (incompatibles entre elles)

- Concept de standard discuté en 1991 pour le parallélisme de boucles en Fortran
- OpenMP 1.0 : 1997 en Fortran, 1998 en C et C++
- OpenMP 2.0 : 2000 en Fortran, 2002 en C et C++
  - Parallélisation de boucles
  - 2.5 combine les directive C/C++/Fortran en 2005
- OpenMP 3.0 : 2008
  - Parallélisme de tâches (tasks construct)
  - Amélioration du parallélisme imbriqué
- OpenMP 4.0 : 2013
  - Sur accélérateurs (host/device)
  - Optimiser le transfert de données (target data)
  - Architectures SIMD
  - Notion de constructeur de groupes de tâches
- OpenMP 5.0 : nov. 2018
  - Ajout d'outils d'analyse de performance et de débogage

Quelle version d'OpenMP j'utilise ?

Pour connaître la version d'OpenMP utilisée, activer la variable d'environnement

```
export OMP_DISPLAY_ENV="TRUE"
```

Compiler son code avec OpenMP et exécuter

```
./a.out
```

En sortie, s'affiche :

```
OPENMP DISPLAY ENVIRONMENT BEGIN
  _OPENMP = '201511'
  OMP_DYNAMIC = 'FALSE'
  OMP_NESTED = 'FALSE'
  OMP_NUM_THREADS = '4'
  OMP_SCHEDULE = 'DYNAMIC'
  OMP_PROC_BIND = 'FALSE'
  OMP_PLACES = ''
  OMP_STACKSIZE = '0'
  OMP_WAIT_POLICY = 'PASSIVE'
  OMP_THREAD_LIMIT = '4294967295'
  OMP_MAX_ACTIVE_LEVELS = '2147483647'
  OMP_CANCELLATION = 'FALSE'
  OMP_DEFAULT_DEVICE = '0'
  OMP_MAX_TASK_PRIORITY = '0'
OPENMP DISPLAY ENVIRONMENT END
```

`_OPENMP` défini la version d'OpenMP utilisée (ici 4.5). Voir la liste sur `https://www.openmp.org/resources/openmp-compilers-tools/`

Alternatives ?

Parallélismes différents

- MPI (mémoire distribuée)
- TBB, Pthreads (mémoire partagée)
- OpenACC, CUDA, OpenCL (sur accélérateurs)
- StarPU (parallélisme de tâches)
- kokkos, Alpaka (cross-platforms)

Structure générale d'un programme OpenMP

```
// debut du programme

// declaration de l'appel a la bibliotheque

// zone sequentielle

// fork : directives de parallelisation

// region parallele

// join : cloture de la region parallele

// zone sequentielle

// fin du programme
```

- La syntaxe des directives dépend du langage utilisé (C, C++ ou Fortran)
- Les directives sont vues comme des commentaires si le code n'est pas compilé avec les bibliothèques OpenMP du langage

Déclarer l'appel à OpenMP

Fortran

```
!$ use OMP_LIB
```

```
gfortran prog.f -fopenmp

ifort prog.f -openmp

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

C

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

```
gcc prog.c -fopenmp

icc prog.c -openmp

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

C++

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

```
gcc prog.cpp -fopenmp

icpc prog.cpp -openmp

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

Exemple

en Fortran

```
PROGRAM hello_world

  !$ use OMP_LIB
  implicit none

  !$OMP PARALLEL
  write (*,*) "hello world!"
  !$OMP END PARALLEL

END PROGRAM hello_world
```

en C

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

int main (int argc, char *argv[]) {
  #pragma omp parallel
  {
    printf("hello world!\n");
  }
  return 0;
}
```

Fortran

```
gfortran prog.f -fopenmp
```

C

```
gcc prog.c -fopenmp
```

Résultat

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

hello world!
hello world!
hello world!
hello world!
```

Directives OpenMP

Paralléliser un code avec OpenMP revient à ajouter des diectives pour indiquer au compilateur

- où sont les parties à exécuter en parallèle
- comment distribuer les instructions et les données entre les threads

Syntaxe en 4 parties :

```
sentinelle directive [clauses] (retour a la ligne)
```

où

- la sentinelle indique l'entrée de la zone à traiter par OpenMP
  - en Fortran

    ```
    !$OMP
    ```

  - en C/C++

    ```
    #pragma omp
    ```

- la directive qui définit l'action parallèle à exécuter
- des options, précisant éventuellement le caractère local d'une variable, etc.
- un retour à la ligne (ou un \ si les directives sont trop longues)

Principales directives OpenMP

- Entrée dans une région parallèle

`parallel` crée une région parallèle (fork-join)

- Partage des tâches

`for` boucles à exécuter en parallèle
`sections` détermine les blocs à exécuter en parallèle
`single` désigne un bloc à exécuter par un seul thread

- Synchronisation

`barrier` attente que tous les threads aient fini
`master` désigne un bloc à exécuter par le thread maître
`critical` bloc à exécuter par un seul thread à la fois
`atomic` instruction dont l'écriture mémoire est atomique

- Gestion des tâches

`tasks` déclaration de tâches filles
`taskswait` attente de la fin des tâches filles

- Portée des variables

`shared`	partagée
`private`	privée
`firstprivate`	initialisation de copie
`lastprivate`	conserve la valeur en fin de boucle
`default`	`(shared|private|none)`
`copyin`	initialisation de variables privées

- Environnement d'exécution

`omp_set_num_threads()`	nombre total de threads
`omp_get_thread_num()`	identifiant du thread
`OMP_NUM_THREADS`	nombre de threads dans le shell
`OMP_DYNAMIC`	booléen
`OMP_STATIC`	booléen
`OMP_SCHEDULE`	permet de spécifier le type et la taille des paquets (chunk)

`parallel` : construction d'une région parallèle

- Le code d'une région parallèle est dupliqué et tous les threads exécutent le même code
- Il y a une barrière implicite à la fin de la région parallèle. Seul le thread 0 continue après cette barrière.
- Par défaut, le statut des variables d'une région parallèle est partagé.

```
!$OMP PARALLEL [clause ...]
                IF (scalar_logical_expression)
                PRIVATE (list)
                SHARED (list)
                DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)
                FIRSTPRIVATE (list)
                REDUCTION (operator: list)
                COPYIN (list)
                NUM_THREADS (scalar-integer-expression)

!$OMP END PARALLEL
```

```
#pragma omp parallel [clause ...] newline
                        if (scalar_expression)
                        private (list)
                        shared (list)
                        default (shared | none)
                        firstprivate (list)
                        reduction (operator: list)
                        copyin (list)
                        num_threads (integer-expression)
```

Clause if

Permet de ne créer une région parallèle que si la condition est vraie, afin de ne pas être pénalisé par le temps de création des threads par rapport au temps d'exécution des instructions dans la région parallèle

```
PROGRAM main

  !$ use OMP_LIB
  implicit none
  logical, parameter :: PARALLEL=.TRUE.

  !$OMP PARALLEL IF(PARALLEL)
  write (*,*) "Hello"
  !$OMP END PARALLEL

END PROGRAM main
```

```
#include <omp.h>
#include <stdio.h>
#define PARALLEL 1

int main() {
    #pragma omp parallel if (PARALLEL)
    printf("Hello\n");
    return 0;
}
```

Clause num_threads

Permet de spécifier le nombre de threads qui exécutera la région parallèle

```
PROGRAM main

  !$ use OMP_LIB
  implicit none

  !$OMP PARALLEL NUM_THREADS(2)
  write (*,*) "Hello"
  !$OMP END PARALLEL

END PROGRAM main
```

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

int main() {
    #pragma omp parallel num_threads(2)
    printf("Hello\n");
    return 0;
}
```

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

Exemple en Fortran

```
      PROGRAM HELLO

!$ use OMP_LIB

      INTEGER NTHREADS, TID

! Fork a team of threads with each thread having a private TID variable
!$OMP PARALLEL PRIVATE(TID)

! Obtain and print thread id
      TID = 0
!$ TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Hello World from thread = ', TID

! Only master thread does this
      IF (TID .EQ. 0) THEN
        NTHREADS = 0
!$ NTHREADS = OMP_GET_NUM_THREADS()
        PRINT *, 'Number of threads = ', NTHREADS
      END IF

! All threads join master thread and disband
!$OMP END PARALLEL

      END
```

En séquentiel

```
gfortran hello.f
```

```
./a.out

 Hello World from thread = 0
 Number of threads = 0
```

En parallèle

```
gfortran hello.f -fopenmp
```

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

 Hello World from thread = 0
 Number of threads = 4
 Hello World from thread = 3
 Hello World from thread = 1
 Hello World from thread = 2
```

Exemple en C

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

int main(int argc, char *argv[]) {

int nthreads, tid;

/* Fork a team of threads with each thread having a private tid variable */
#pragma omp parallel private(tid)
  {
  /* Obtain and print thread id */
  tid = 0;
#ifdef _OPENMP
  tid = omp_get_thread_num();
#endif
  printf("Hello World from thread = %d\n", tid);

  /* Only master thread does this */
  if (tid == 0) {
#ifdef _OPENMP
    nthreads = omp_get_num_threads();
#endif
    printf("Number of threads = %d\n", nthreads);
  }
  } /* All threads join master thread and terminate */
  return 0;
}
```

En séquentiel

```
gcc hello.c
```

```
./a.out

Hello World from thread = 0
Number of threads = 0
```

En parallèle

```
gcc hello.c -fopenmp
```

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

Hello World from thread = 3
Hello World from thread = 2
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 1
```

Portée des variables

La portée des variables dépend des clauses définissant leur statut.
Dans une région parallèle, une variable peut être

- `partagée (shared)` : connue et modifiable par tous les threads
- `privée (private)` : accessible seulement par le thread local

Par défaut, une variable déclarée

- en dehors de la région parallèle est `shared`
- dans la région parallèle est `private`

En dehors de la région parallèle, une variable `private` a la valeur du thread maître.

Attention aux accès concurrents

```
int x=1;
#pragma omp parallel private(x)
  {
    x = omp_get_thread_num()+1;
    // ici x vaut 1 ou 2 ou 3 ... selon le rang du thread
  }
// ici x=1
```

Source d'erreur

```
int x=1;
#pragma omp parallel shared(x)
  {
    x = omp_get_thread_num()+1;
    // ici x vaut 1 ou 2 ou 3 ... selon le rang du thread
  }
// ici x est aleatoire
```

Variables privées

Les variables `private` et `firstprivate` sont dupliquées dans chaque thread.
Elles ne sont pas accessibles en dehors de leur région parallèle.
Les variables `firstprivate` sont initialisées au début de la région.
Seules les variables `threadprivate` conservent leur valeur d'une région parallèle à une autre.
Les variables `threadprivate` peuvent être initialisées par la thread master au travers de la clause `copyin`.

Variables partagées

Par défaut, les variables sont partagées (`shared`). Il est recommandé de définir le statut de chaque variable explicitement avec la clause `default(none)`.
Les variables statiques sont conservées dans toute l'exécution d'un programme. En C, elles sont déclarées par le mot clef `static`, alors qu'en Fortran elles sont soit placées dans un `COMMON`, soit déclarées `save` ou initialisées à la déclaration dans un `data`.
La directive `threadprivate` permet de privatiser une instance statique, et faire qu'elles soient partagées d'une région parallèle à l'autre.

Exemple en Fortran

```
PROGRAM prog
  !$ use OMP_LIB
  implicit none
  integer :: x, y
  x=10
  y=1
  !$OMP PARALLEL DEFAULT(NONE) FIRSTPRIVATE (x) PRIVATE (y)
  print *, "inside parallel regio x= ", x, "y= ",y
  !$OMP END PARALLEL
  print *, "outside x= ", x, "y= ",y
END PROGRAM prog
```

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

 inside parallel regio x= 10 y= 0
 inside parallel regio x= 10 y= 0
 inside parallel regio x= 10 y= 0
 inside parallel regio x= 10 y= 0
 outside x= 10 y= 1
```

Exemple en C

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

int main (int argc, char *argv[]) {

  int x=10;
  int y=1;
  #pragma omp parallel default(none) firstprivate(x) private(y)
  {
    printf("inside parallel regio x= %i y=%i\n",x,y);
  }
  printf("outside x= %i y=%i\n",x,y);

  return 0;
}
```

```
export OMP_NUM_THREADS=4
gcc prog.c -fopenmp
./a.out

inside parallel regio x= 10 y=0
inside parallel regio x= 10 y=0
inside parallel regio x= 10 y=0
inside parallel regio x= 10 y=0
outside x= 10 y=1
```

Directive `threadprivate`

- Les variables `threadprivate` sont privées et persistantes : elles conservent leur valeur d'une région parallèle à une autre.
- Les variables `threadprivate` peuvent être initialisées par la thread master au travers de la clause `copyin`
- Le nombre de threads soit être fixe (`omp_set_dynamic(0)`)

Exemple en Fortran

```
PROGRAM main

  !$ use OMP_LIB

  implicit none
  integer :: tid, tprivate, rprivate
  common /tprv/ tprivate
  !$OMP THREADPRIVATE(/tprv/)

  call OMP_SET_DYNAMIC(.false.)

  print*,"Parallel regio"
  !$OMP PARALLEL PRIVATE(tid, rprivate)
  tid = OMP_GET_THREAD_NUM();
  tprivate = tid;
  rprivate = tid;
  print*,"Thread ",tid,": tprivate=",tprivate," rptivate=",rprivate
  !$OMP END PARALLEL

  print*,"Parallel regio"
  !$OMP PARALLEL PRIVATE(tid, rprivate)
  tid = OMP_GET_THREAD_NUM();
  print*,"Thread ",tid,": tprivate=",tprivate," rptivate=",rprivate
  !$OMP END PARALLEL

END PROGRAM main
```

Exemple en C

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

int tid, tprivate, rprivate;
#pragma omp threadprivate(tprivate)

int main (int argc, char *argv[]) {

  omp_set_dynamic(0);

  printf("Parallel regio \n");
  #pragma omp parallel private(tid, rprivate)
  {
    tid = omp_get_thread_num();
    tprivate = tid;
    rprivate = tid;
    printf("Thread %d: tprivate=%d rptivate=%d\n", tid, tprivate, rprivate);
  }

  printf("Parallel regio \n");
  #pragma omp parallel private(tid, rprivate)
  {
    tid = omp_get_thread_num();
    printf("Thread %d: tprivate=%d rptivate=%d\n", tid, tprivate, rprivate);
  }
  return 0;
}
```

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

Parallel regio
Thread 0: tprivate=0 rptivate=0
Thread 3: tprivate=3 rptivate=3
Thread 1: tprivate=1 rptivate=1
Thread 2: tprivate=2 rptivate=2
Parallel regio
Thread 3: tprivate=3 rptivate=0
Thread 1: tprivate=1 rptivate=0
Thread 0: tprivate=0 rptivate=0
Thread 2: tprivate=2 rptivate=0
```

Clause `reduction`

```
reduction(operateur: )
```

- Effectue une réduction sur les éléments d'une liste
- Les variables dans la liste doivent être partagées
- Une copie privée de chaque variable est créee pour chaque thread, puis l'opérateur de réduction est appliqué aux variables privées et le résultat est copié dans la variable partagée correspondante
- `operateur` peut valoir `+,-,*,min,max` et les opérateurs logiques suivants
  - en C
    `&,&&,|,||,^`
  - en Fortran
    `.and.,.or.,.eqv.,.neqv.,.iand.,.ior.,.ieor.`

Les opérations de réductions peuvent engendrer un surcoût lié à la création des threads.

Exemple de réduction en Fortran

```
PROGRAM dot_product

  !$ use OMP_LIB
  implicit none
  integer, parameter :: N=1000000
  integer :: chunksize, chunk,i
  real, dimension(N) :: a,b
  real :: res

  chunksize = 10
  do i = 1, N
    a(i) = i * 1.0
    b(i) = i * 2.0
  enddo
  res = 0.0
  chunk = chunksize

  !$OMP PARALLEL DO &
  !$OMP DEFAULT(SHARED) PRIVATE(i) SCHEDULE(STATIC,chunk) &
  !$OMP REDUCTION(+:res)
  do i = 1, N
    res = res + a(i)*b(i)
  enddo
  !$OMP END PARALLEL DO

  print *, 'final result= ', res

END PROGRAM dot_product
```

Exemple de réduction en C

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

int main(int argc, char *argv[]) {

  int i, N=1000000, chunk;
  float a[N], b[N], res;

  /* Some initializations */
  chunk = 10;
  res = 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(+:res)

  for (i=0; i < N; i++) {
    res = res + (a[i] * b[i]);
  }

  printf("Final result= %f\n",res);
  return 0;
}
```

Partage du travail

Les directives de partage de travail son incluses dans des régions parallèle. Elles permettent

- la répartition de l'exécution d'une ou plusieurs boucles entre les threads (`DO` en Fortran, `for` en C)
- la répartition de plusieurs sections de code entre les threads, une section de code par thread (`sections`)
- l'exécution d'une portion de code par différents threads, par un seul (`single`)

Elles impliquent un barrière à la fin de la construction (sauf si la clause `nowait`) est spécifiée.

Directive for/DO

Dans des boucles itératives, la syntaxe est

en Fortran

```
!$OMP DO [clause]
      DO (...)
        ...
      END DO
!$OMP END DO
```

en C

```
#pragma omp for [clause]
  for (...) {
    ...
  }
```

La variable d'itération est privée par défaut

Exemple en Fortran

```
program main

  !$ use OMP_LIB
  implicit none
  real*8, dimension(:,:), allocatable :: A
  integer :: i,j,N,M

  N=50000
  M=50000

  allocate (A(M,N))

  !$OMP PARALLEL DEFAULT(NONE), PRIVATE (i,j), &
  !$OMP SHARED(A,N,M)
  do j=1,N
    do i=1,M
      A(i,j)=1.D0
    enddo
  enddo
  !$OMP END PARALLEL

  deallocate (A)

end program main
```

(pas optimal)

En séquentiel

```
gfortran prog.f90
```

```
time ./a.out

real 0m12,900s
user 0m8,230s
sys 0m4,667s
```

En parallèle

```
gfortran prog.f90 -fopenmp
```

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

real 0m6,226s
user 0m15,923s
sys 0m6,538s
```

Exemple en C

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main()
{
  int N,M,len=0;
  int i,j;
  double **A;

  N=50000;
  M=50000;

  A = (double **)malloc(N*sizeof(double *));
  // initialize
  for(i=0; i<N; i++) {
    A[i] = (double *)malloc(M*sizeof(double));
  }
```

...

...

```
  #pragma omp parallel for private(j) firstprivate(A)
  for (i=0; i<N; i++) {
    for (j=0; j<M; j++) {
      A[i][j] = 1.0;
    }
  }

  // deallocate
  for (i=0; i<N; i++) {
    free(A[i]);
  }
  free(A);

  return 0;
}
```

(pas optimal)

En séquentiel

```
gcc prog.c
```

```
./a.out

real 0m13,004s
user 0m8,201s
sys 0m4,788s
```

En parallèle

```
gcc prog.c -fopenmp
```

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

real 0m5,697s
user 0m12,347s
sys 0m7,136s
```

Attention :

Les itérations dans les boucles multithreadées doivent être indépendantes :

Instructions indépendantes :

```
#pragma omp for
for (int i=1; i<N; i++) {
  A[i] = B[i]*A[i];
}
```

Perte d'efficacité :

```
#pragma omp for
for (int i=1; i<N; i++) {
  A[i] = B[i]*A[i-1];
}
```

de manière générale : penser vectoriel

Clause `schedule`

```
schedule(type[, chunk])
```

Spécifie la manière dont sont diviées les itérations entre les threads :

- **static** blocs de `chunk` itérations consécutives (équirépartition par défaut)
- **dynamic** chaque thread demande un bloc de `chunk` itérations consécutives quand il n'a pas de travail (`chunk=1` par défaut)
- **guided** comme `dynamic` mais la taille des blocs décroît exponentiellement
- **runtime** choix reporté au moment de l'exécution et spécifié par la variable d'environnemnt `OMP_SCHEDULE`
- **auto** choix laissé au compilateur et au runtime

Le choix de la répartition de tâches impacte les performances.

Clause collapse

`schedule(#)`

Spécifie le nombre de boucles associées à une directive `for` (1 par défaut) :

- regoupe les # boucles en un seul espace d'itérations
- l'ordre initial est respecté

```
#pragma omp for collapse(2)
for (int i=0; i<10; i++)
  for (int j=0, j<10; j++)
    f(i,j);
```

```
#pragma omp for
for (int n=0; n<100; n++)
  f(n/10,n\%10)
```

Optimisation

Distribution des régions parallèles :

```
!$OMP PARALLEL DO
DO k=1,10
  DO j=1,100
    DO i=1,1000
      a(i,j,k) = ...
    ENDO DO
  ENDO DO
ENDO DO
!$OMP END PARALLEL DO
```

ne sera scalable que si 10 est un multiple du nombre de threads

```
DO k=1,10
  DO j=1,100
!$OMP PARALLEL DO
    DO i=1,1000
      a(i,j,k) = ...
    ENDO DO
!$OMP END PARALLEL DO
  ENDO DO
ENDO DO
```

génèrera de l'overhead (lié à la création des threads dans les boucles)
Penser à passer en séquentiel quand le temps de création des threads est plus grand que le temps d'exécution des instructions dans les boucles.

```
  !$OMP PARALLEL DO IF (N>100000)
DO i=1,N
  a(i) = ...
END DO
!$OMP END PARALLEL DO
```

Clause `nowait`

- retire la barrière implicite en fin de région parallèle
- les threads finissant en avance peuvent exécuter les instructions suivantes sans attendre les autres

Limite le temps d'attente dû au désequilibrage (`load imbalance`)

Exemple

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

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

- Syntaxe pertinente à partir de la version 3.0 d'OpenMP
- Valable si la taille des boucles est la même (`N`)
- Valable si la distribution est la même (`chunk` identique)

Directive sections

```
!$OMP SECTIONS [clause[,clause]]
!$OMP SECTION
  ...
!$OMP SECTION
  ...
!$OMP SECTION
  ...
!$OMP END SECTIONS[nowait]
```

```
#pragma omp secions [clause [clause]]
  {
    #pragma omp section
    ...
    #pragma omp section
    ...
    #pragma omp section
    ...
}
```

- Une section est une portion de code exécutée par un seul thread
- Faire attention entre la corresponance entre les threads et le nombre de sections, pour l'équilibrage de charge

Synchronisation

- Directive `barrier`
  - tous les threads doivent arriver à une barrière avant de continuer
  - existe une barrière implicite à la fin des directive, sauf si la clause `nowait` est utilisée
- Ordonnancement pour forcer un ordre global d'exécution
  - clause `ordered`, peut être utile au dégogage
- Exclusion mutuelle pour que l'exécution soit réalisée par un seul thread à la fois. Directives :
  - `critical`
  - `atomic`, utile e.g. lors de mise à jour de varibles partagées
- Directive `master`
  - section de code séquentiel, traité par le thread master
  - e.g. E/S, initialisation de variables
- Directive `flush`
  - spécifie que l'accès du thread doit être cohérent avec l'état de la mémoire partagée
  - implicite après une région parallèle, un partage de travail (hors `nowait`), une section critique

Concept de tâche

Les tâches désignent une instance de code exécutable (et ses données associées). Elles peuvent être construites par la directive `parallel` ou `task`
Les tâches créées par la directive

- `parallel` sont implicites
- `task` sont explicites

Plusieurs types de synchronisation sont possibles :

- la directive `taskwait` permet d'attendre la terminaison de tous les threads
- le directive `taskgroup/end taskgroup` permet d'attendre la terminaison de tous les descendants d'un groupe de tâches (en arbre)
- les barrières implicites ou explicites permettent d'attendre la terminaison de toutes les tâches déjà créées.

Exemple (repris du cours de l'IDRIS)

Code séquentiel

```
program task_print

  implicit none

  print *,"Un "
  print *,"petit "
  print *,"pas "

end program task_print
```

```
gfortran task_print.f90
./a.out

Un
petit
pas
```

Directive parallel

```
program task_print

  !$ use OMP_LIB
  implicit none

  !$OMP PARALLEL
  print *,"Un "
  print *,"petit "
  print *,"pas "
  !$OMP END PARALLEL

end program task_print
```

```
gfortran task_print.f90 -fopenmp
export OMP_NUM_THREADS=2; ./a.out

Un
petit
pas
Un
petit
pas
```

Directive single

```
program task_print

  !$ use OMP_LIB
  implicit none

  !$OMP PARALLEL
  !$OMP SINGLE
  print *,"Un "
  print *,"petit "
  print *,"pas "
  !$OMP END SINGLE
  !$OMP END PARALLEL

end program task_print
```

```
gfortran task_print.f90 -fopenmp
export OMP_NUM_THREADS=2; ./a.out

Un
petit
pas
```

Directive task

```
program task_print

  !$ use OMP_LIB
  implicit none

  !$OMP PARALLEL
  !$OMP SINGLE
  print *,"Un "
  !$OMP TASK
  print *,"petit "
  !$OMP END TASK
  !$OMP TASK
  print *,"pas "
  !$OMP END TASK
  !$OMP END SINGLE
  !$OMP END PARALLEL

end program task_print
```

```
gfortran task_print.f90 -fopenmp
export OMP_NUM_THREADS=2; ./a.out
```

run 1

```
Un
petit
pas
```

run 2

```
Un
pas
petit
```

Directive taskwait

```
program task_print

  !$ use OMP_LIB
  implicit none

  !$OMP PARALLEL
  !$OMP SINGLE
  print *,"Un "
  !$OMP TASK
  print *,"petit "
  !$OMP END TASK
  !$OMP TASK
  print *,"pas "
  !$OMP END TASK
  !$OMP TASKWAIT
  print *,"pour l'homme"
  !$OMP END SINGLE
  !$OMP END PARALLEL

end program task_print
```

```
gfortran task_print.f90 -fopenmp
export OMP_NUM_THREADS=2; ./a.out
```

```
Un
petit
pas
pour l'homme
```

```
Un
pas
petit
pour l'homme
```

Clause depend

La clause depend permet de gérer les dépendances entre les tâches explicites générées par la même tâche. La liste de dépendance est de type `in`, `out` ou `inout`

```
program task_print

  !$ use OMP_LIB
  implicit none
  integer :: T1

  !$OMP PARALLEL
  !$OMP SINGLE
  print *,"Un "
  !$OMP TASK DEPEND(OUT:T1)
  print *,"petit "
  !$OMP END TASK
  !$OMP TASK DEPEND(IN:T1)
  print *,"pas "
  !$OMP END TASK
  !$OMP TASKWAIT
  print *,"pour l'homme"
  !$OMP END SINGLE
  !$OMP END PARALLEL

end program task_print
```

```
gfortran task_print.f90 -fopenmp
export OMP_NUM_THREADS=2; ./a.out
```

```
Un
petit
pas
pour l'homme
```

```
Un
petit
pas
pour l'homme
```

Suite de Fibonacci

La suite de Fibonacci est définie par

$$f(0) = 0$$
$$f(1) = 1$$
$$f(n) = f(n-1) + f(n-2)$$

Exemple en Fortran

```
module fib_module
  implicit none
contains

  recursive integer function fib(n) &
  result(res)

  integer, intent(in) :: n
  integer :: i,j

  if (n<2) then
    res = n
  else
    !$OMP TASK SHARED(i)
    i=fib(n-1)
    !$OMP END TASK
    !$OMP TASK SHARED(j)
    j=fib(n-2)
    !$OMP END TASK
    !$OMP TASKWAIT
    res=i+j
  endif

  end function fib
end module fib_module
```

```
program fib_main

  !$ use OMP_LIB
  use fib_module

  implicit none
  integer, parameter :: N=10
  integer :: res_fib

  !$OMP PARALLEL
  !$OMP SINGLE
  res_fib=fib(N)
  !$OMP END SINGLE
  !$OMP END PARALLEL
  print *,"res_fib = ",res_fib

end program fib_main
```

Exemple en C

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

int fib(int n)
{
  int i, j;
  if (n<2)
    return n;
  else
    {
       #pragma omp task shared(i) firstprivate(n)
       i=fib(n-1);

       #pragma omp task shared(j) firstprivate(n)
       j=fib(n-2);

       #pragma omp taskwait
       return i+j;
    }
}
```

```
int main()
{
  int n = 10;

  omp_set_dynamic(0);
  omp_set_num_threads(4);

  #pragma omp parallel shared(n)
  {
    #pragma omp single
    printf ("fib(%d) = %d\n", n, fib(n));
  }
}
```

Bibliothèque et variables d'environnement

La bibliothèque regroupe

- les fonctions liées à l'environnement d'exécution
  - pour l'ordonnancement (e.g. `OMP_SCHEDULE`)
  - pour le runtime (e.g. `OMP_NUM_THREADS`)
- les utilitaires
- fonctions de mesure du temps

La variable `_OPENMP` permet d'utiliser les directives du préprocesseur pour écrire un code en version parallèle et séquentielle et définir différentes macros.

```
#include <stdio.h>
#ifdef _OPENMP
  #include <omp.h>
#else
  #define omp_get_thread_num() 0
#endif

int main() {
  #pragma omp parallel
  printf("Hello from thread %d\n", omp_get_thread_num());
  return 0;
}
```

En Fortran

- `call OMP_SET_DYNAMIC(bool)` active l'ajustement automatique du nombre de threads
- `OMP_GET_NUM_THREADS()` nombre de threads dans le pool d'exécution
- `OMP_GET_THREAD_NUM()` numéro du thread courant dans le pool d'exécution
- `OMP_GET_WTIME()` temps écoulé en secondes depuis le temps de référence
- `OMP_GET_WTICK()` temps écoulé en secondes entre deux mesures

En C

- `omp_set_dynamic(int bool);` active l'ajustement automatique du nombre de threads
- `int omp_get_num_threads();` nombre de threads dans le pool d'exécution
- `int omp_get_thread_num();` numéro du thread courant dans le pool d'exécution
- `double omp_get_wtime();` temps écoulé en secondes depuis le temps de référence
- `double omp_get_wtick();` temps écoulé en secondes entre deux mesures

Recommandations

- Utiliser la clause `default(none)`

  pour ne pas déclarer par erreur une variable partagée
- Définir les régions parallèles hors des boucles autant que possible

  pour ne pas avoir d'overhead par rapport au temps d'exécution de la boucle
- Ne pas imposer de synchronisation inutiles

  penser à la clause `nowait`

  séparer la déclaration des région `parallel` et les directives de boucle `do/for`
- Utiliser les méthodes adaptées

  `reduction`, ...
- Équilibrer la charge si possible

  avec la clause `schedule` ad hoc
- Donner suffisamment de travail aux threads

  penser à la clause `if`

Références

- Cours OpenMP, IDRIS
- Tutorial LLNL
- Standard OpenMP 4.5
- Cours de Cédric Bastoul, Unistra