Formation en Calcul Scientifique - LEM2I

Introduction à OpenMP

Loïc Gouarin, Violaine Louvet, Laurent Series

Groupe Calcul CNRS

9-13 avril 2012

Plan

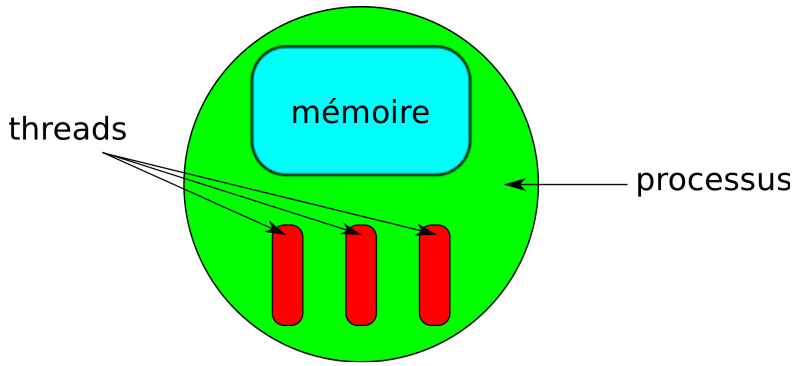
Plan

Qu'est-ce qu'OpenMP

OpenMP (*Open Multi-Processing*) est une **interface de programmation** (API) pour générer un **programme multithreading** sur **architecture à mémoire partagée**.

Modèle de programmation multithreading

- Un programme multithreading s'exécute dans un processus unique.
- Ce processus active plusieurs processus légers (appelés également *threads*) capables de s'exécuter de manière concurrente.
- L'exécution de ces processus légers se réalise dans l'espace mémoire du processus d'origine.

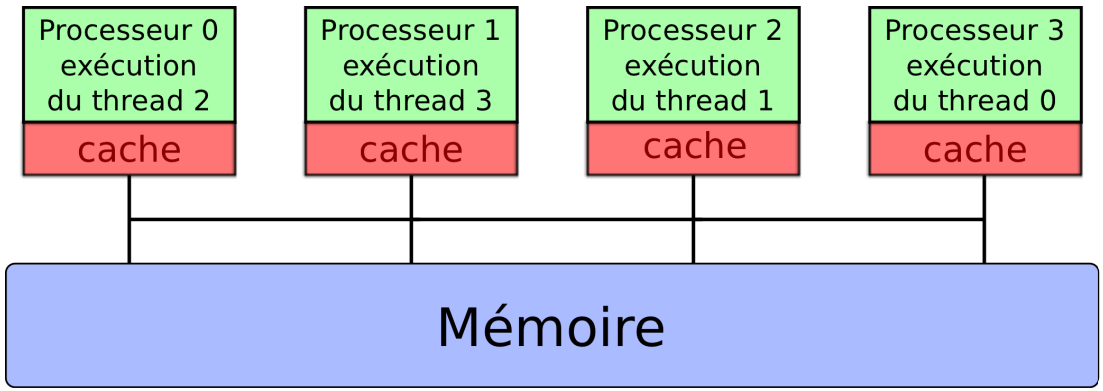


$\Longrightarrow$ architecture cible : machine à mémoire partagée.

Modèle de programmation multithreading

- C'est l'exécution concurrente des threads sur plusieurs processeurs ou cœurs qui permet l'exécution parallèle du programmme.
- Le système d'exploitation distribue les threads sur les différents processeurs ou cœurs de la machine à mémoire partagée.

Exemple : distribution de 4 threads sur une machine à mémoire partagée de 4 processeurs de type UMA.

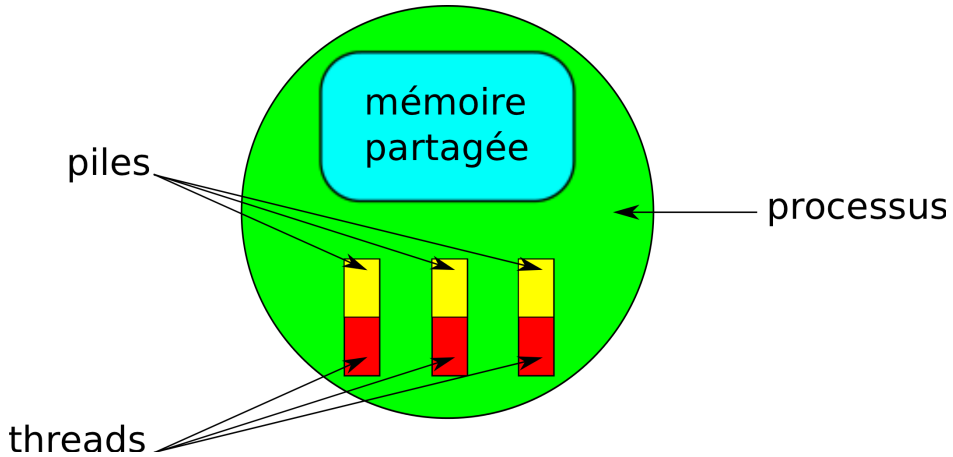


Modèle de programmation multithreading

- Les threads d'un programme multithreading partagent leurs ressources, notamment la mémoire, excepté la **pile** (*stack*).
- La **pile** est un espace mémoire local à chaque thread, invisible des autres threads.

Modèle de programmation multithreading

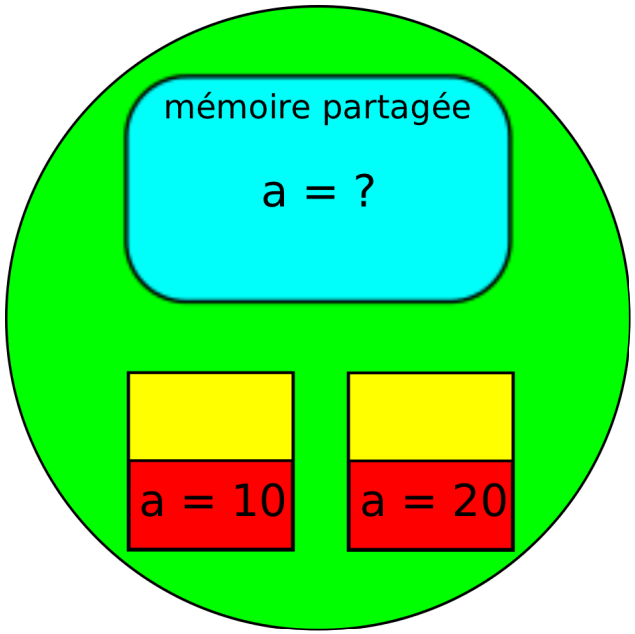
- Un thread dispose donc d'un accès :
  - à un espace mémoire partagé par tous les threads, c'est le lieu d'adressage des **variables partagées**
  - à un espace mémoire privé, c'est le lieu d'adressage des **variables privées**



Modèle de programmation multithreading

- A cause du partage d'un espace mémoire entre les threads, une **synchronisation** entre les threads concurrents est parfois nécessaire.

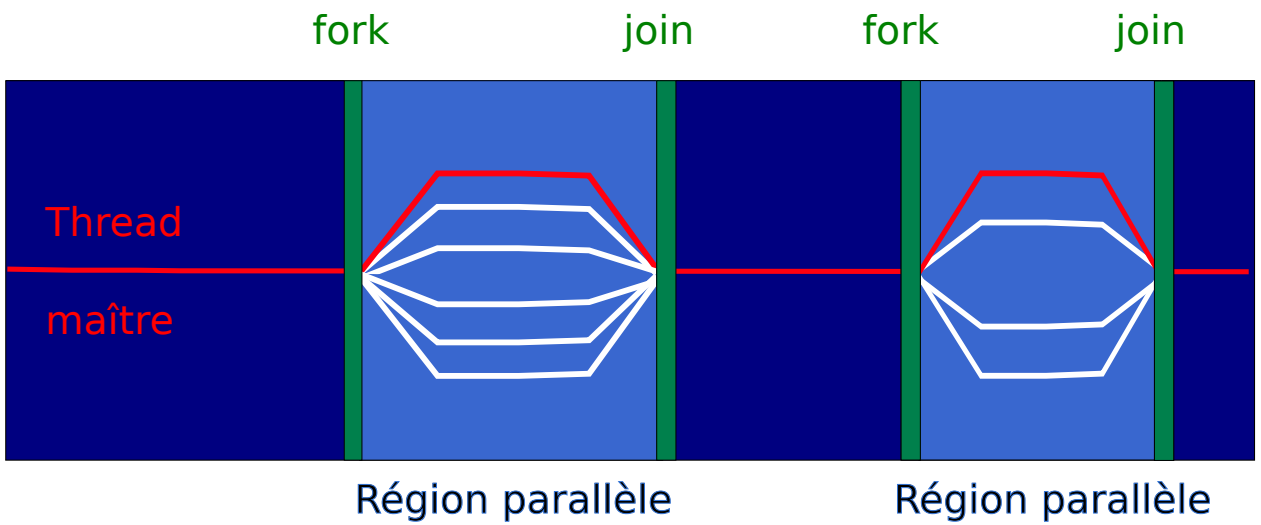
  Par exemple, il peut être nécessaire d'imposer l'ordre de modification d'une variable partagée pour assurer la cohérence en mémoire.



Modèle de programmation multithreading

Un programme OpenMP est une succession de régions séquentielles et parallèles.

Le thread maître crée (*fork*) des processus légers (*threads*) à l'entrée de régions dans lesquelles le code est exécuté en parallèle puis les désactive en fin de région (*join*).



Interface de programmation OpenMP

- L'interface de programmation fournit :
  - des directives de compilation ;
  - des sous-programmes ;
  - des variables d'environnement.
- Cette API est :
  - disponible pour le Fortran, le C et le C++ ;
  - supportée par de nombreux systèmes et compilateurs (gnu, intel...).

Interface de programmation OpenMP

- The OpenMP Architecture Review Board (http ://www.openmp.org) est en charge des spécifications.
- OpenMP-2 date de 2000, OpenMP-3 de 2008. OpenMP-4 est prévu pour 2012.
- OpenMP est un standard "industriel".

Directives

- Les directives OpenMP permettent de
  - débuter et terminer une région parallèle ;
  - contrôler la répartition du travail ;
  - synchroniser les threads ;
  - ...
- Ce sont des lignes ayant une syntaxe particulière et prises en compte par le compilateur uniquement si l'option permettant leur interprétation est spécifiée (sinon elles sont interprétées comme des commentaires ce qui permet de préserver le code séquentiel).

Syntaxe :

```
sentinelle directive [clause[ clause]...]
```

Directives

Premier exemple

Fortran format libre

Sentinelle : `!$omp`

```
program parallel
  implicit none

!$omp parallel

    print *,"Hello World !"

!$omp end parallel

end program parallel
```

Directives

- Les directives les plus utilisées sont :
  - `parallel`
  - `do` en Fortran
  - `sections`
  - `single`
  - `atomic`
  - `barrier`
  - ...

Directives

- Chaque directive accepte un nombre variable de clauses (`default`, `shared`, `reduction`...) : 8 pour `parallel`, 0 pour `atomic`.
- Une directive peut s'écrire sur plusieurs lignes.
- Les clauses permettent notamment de changer le statut des variables (partagée ou privée).
- Certaines directives peuvent être fusionnées avec la directive `parallel`.
  Exemple : `parallel do`.

Compilation et variables d'environnement

- L'interprétation des directives est activée par une option :
  - Pour les compilateurs GNU (gcc, gfortran), il s'agit de `-fopenmp`.
  - Pour les compilateurs Intel (icc, ifort), il s'agit de `-openmp`.
- Les principales variables d'environnement sont les suivantes :
  - `OMP_NUM_THREADS` : définit le nombre de threads à lancer ;
  - `OMP_SCHEDULE` : définit la répartition des itérations ;
  - `OMP_STACKSIZE` : définit la taille de la pile.

Compilation et variables d'environnement

Exemple de compilation avec GNU et exécution sur 2 threads.

```
program parallel
  implicit none

!$omp parallel

    print *,"Hello World !"

!$omp end parallel
end program parallel
```

Compilation sans -openmp

```
> gfortran prog.f90
> a.out
 Hello World !
```

Compilation avec -openmp

```
> gfortran -fopenmp prog.f90
> export OMP_NUM_THREADS=2
> a.out
 Hello World !
 Hello World !
```

Sous-programmes

- Les principaux sous-programmes spécifiques de OpenMP sont
  - `omp_get_num_threads()` : retourne le nombre de threads dans la région parallèle ;
  - `omp_get_thread_num()` : retourne le rang du thread ;
  - `omp_set_num_threads(NumThreads)` : permet de spécifier le nombre de threads pour la prochaine région parallèle ;
  - `omp_get_wtime()` : permet de mesurer un temps d'exécution ;
  - `omp_in_parallel()` : vaut 1 (ou `T`) dans une région parallèle.
- Les prototypes sont à définir avant utilisation en incluant le module `omp_lib`.

Sous-programmes

```
program parallel
use omp_lib
  implicit none

  integer :: NumThreads

  NumThreads = omp_get_num_threads()
  print *,"Region sequentielle, &
           Nb de threads :", NumThreads

!$omp parallel private(NumThreads)
  NumThreads = omp_get_num_threads()
  print *,"Region parallele, Nb de threads :", NumThreads
 !$omp end parallel

end program parallel
```

```
> gfortran -fopenmp prog.f90
> export OMP_NUM_THREADS=2
> a.out
Region sequentielle, Nb de threads :  1
Region parallele, Nb de threads :  2
Region parallele, Nb de threads :  2
```

Compilation conditionnelle

- Elle permet la compilation des programmes contenant des fonctions OpenMP sans l'option de compilation interprétant les directives OpenMP.

Fortran (format libre), utiliser `!$`

```
program parallel
!$ use omp_lib
 implicit none

 integer :: NumThreads

 NumThreads=1

!$omp parallel private(NumThreads)
 !$ NumThreads = omp_get_num_threads()
  print *,"Region parallele, Nb de threads :", NumThreads
!$omp end parallel

end program parallel
```

```
> gfortran prog.f90
> a.out
Region parallele, Nb de threads : 1
```

Alternative à OpenMP

- Il existe d'autres bibliothèques pour écrire des programmes multithreading mais elles nécessitent l'écriture de lignes spécifiques (déclaration des structures de threads, créer les threads...).
- La plus connue est la bibliothèque Pthreads (*i.e.* POSIX Threads).
- Cependant, OpenMP est plus simple pour le programmeur comme le montre la comparaison suivante sur "Hello World !"...

Pthreads vs OpenMP

hello.c en Pthreads

```
#include <stdio.h>
#include <pthread.h>
#define NTHR 4

int nt = NTHR, tidNTHR;
pthread_attr_t attr;

void *hello(void *id)
{
  printf("From thread %d out of %d: \
          hello, world\n", *((int *) id), nt);
  pthread_exit(0);
}

int main()
{
  int i, arg1;
  pthread_t threadNTHR;

  /* system threads */
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, \
                        PTHREAD_SCOPE_SYSTEM);
```

```
  /* suite du programme */

 /* create threads */
  for (i = 0; i < nt; i++) {
    tidi = i;
    pthread_create(&threadi, &attr, \
                   hello, (void *) &tidi);
  }

  /* wait for threads to complete */
  for (i = 0; i < nt; i++)
    pthread_join(threadi, NULL);

  return 0;
}
```

```
> gcc -pthread -o hello hello.c
> ./hello
From thread 0 out of 4: hello, world
From thread 2 out of 4: hello, world
From thread 1 out of 4: hello, world
From thread 3 out of 4: hello, world
```

Pthreads vs OpenMP (suite)

hello.c en OpenMP

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

int main()
{
  int nthreads, tid;

#pragma omp parallel private(nthreads, tid)
{
    nthreads = omp_get_num_threads();
    tid = omp_get_thread_num();
    printf("From thread %d out of %d, \
     hello, world\n", tid, nthreads);
}
  return 0;
}
```

```
> gcc -fopenmp -o hello hello.c
> export omp_NUM_THREADS=4
> ./hello
From thread 1 out of 4, hello, world
From thread 2 out of 4, hello, world
From thread 0 out of 4, hello, world
From thread 3 out of 4, hello, world
```

Région parallèle

- La directive `parallel` est la directive OpenMP la plus importante.
- Elle permet la création d'une région parallèle, c'est-à-dire qu'elle permet l'activation (*fork*) de $N-1$ threads par le thread maître (thread de rang 0).
- Dans une région parallèle, chaque thread exécute le code qui s'y trouve inclus.
- A la fin de la région parallèle, les $N-1$ threads créés par le thread maître sont désactivés et seul le thread maître continue à s'exécuter (*join*).

Région parallèle

```
program parallel

! Partie du code exécutée en séquentiel

  print *, "Avant region parallele"
!$omp parallel
! Partie exécutée en parallèle
    print *, "Pendant region parallele"
!$omp end parallel

! Partie du code exécutée en séquentiel
  print *, "Apres region parallele"

end program parallel
```

```
> gfortran -fopenmp prog.f90
> export OMP_NUM_THREADS=4
> a.out
Avant region parallele
Pendant region parallele
Pendant region parallele
Pendant region parallele
Pendant region parallele
Apres region parallele
```

Région parallèle

- Le nombre de threads exécutant une région parallèle peut être défini de plusieurs façons :
  - par la variable d'environnement `OMP_NUM_THREADS` ;
  - en utilisant le sous-programme `omp_set_num_threads(NumThreads)` ;
  - grâce à la clause `num_threads` de la directive `parallel` ;
- Le nombre de threads peut être choisi indépendamment du nombre de cœurs, leur répartition sur ceux-ci est à la charge de l'O.S.

Région parallèle : clause `if`

- La clause `if` permet de mettre en place une parallélisation conditionnelle :

```
program parallel

  integer, parameter :: n = 10000

!$omp parallel if(n > 1000)
! Partie de code exécutée sur
! l'ensemble des threads si n > 1000
!$omp end parallel



end program parallel
```

Exercice 1

1. Ecrire un programme écrivant une chaîne de caractère dans une région parallèle.
2. Exécutez le programme. Sur combien de threads s'exécute la partie parallèle ?
3. Renseignez `OMP_NUM_THREADS` : `export OMP_NUM_THREADS=2` et exécutez le programme.

Variables partagées et privées

- Au sein d'une région parallèle, une variable est soit **partagée** soit **privée**.
- Une variable partagée se trouve dans la mémoire globale : tous les threads accèdent à la même instance de cette variable.
- Une variable privée
  - se trouve dans la pile de chaque thread (duplication) ;
  - est déclarée par une clause `private` ou `firstprivate` de la directive `parallel`.

Variables partagées et privées

- Par défaut, sauf exceptions (indices de boucles des directives `dor...`), toute variable d'une région parallèle est partagée.
- La clause `default(none|private|shared)` en Fortran permet de modifier le statut par défaut dans une région parallèle.

```
program parallel
 use omp_lib
 implicit none
 integer :: a, b
 a = 100
 !$omp parallel default(none) private(a) shared(b)
 ...
 !$omp end parallel

end program parallel
```

Exercice 2

1. Ecrire un programme initialisant et affichant dans une région parallèle
   - la variable `nb_threads` contenant le nombre de threads en utilisant la fonction `omp_get_num_threads`
   - la variable `thread_nb` contenant le rang du thread en utilisant la fonction `omp_get_thread_num`
2. Quel statut doivent avoir les variables `nb_threads` et `thread_nb` ?
3. Tester les fonctions `omp_get_thread_num()` et `omp_get_thread_num()` en dehors de la région parallèle

Variables partagées et privées

Une variable rendue privée par la clause `private` a une valeur indéterminée en entrée de région parallèle :

```
program parallel
 use omp_lib
 implicit none
 integer :: a, rang

 a = 100

 !$omp parallel default(none) private(a,rang)
   rang = omp_get_thread_num()
   a = a + rang
   print *,"a vaut : ",a
 !$omp end parallel

end program parallel
```

```
> gfortran -fopenmp prog.f90
> export OMP_NUM_THREADS=2
> a.out
 a vaut :            1
 a vaut :        32667
```

Variables partagées et privées

La clause `firstprivate` force l'initialisation à la valeur de la variable avant l'entrée dans la région parallèle :

```
program parallel
 use omp_lib
 implicit none
 integer :: a, rang

 a = 100

!$omp parallel default(none) private(rang) firstprivate(a)
   rang = omp_get_thread_num()
   a = a + rang
   print *,"a vaut : ",a
!$omp end parallel

end program parallel
```

```
> gfortran -fopenmp prog.f90
> export OMP_NUM_THREADS=2
> a.out
 a vaut :           100
 a vaut :           101
```

Cas de l'allocation dynamique

- L'allocation et la désallocation dynamiques sont possibles dans une région parallèle.
- Une opération d'allocation ou de désallocation mémoire sur une variable privée sera locale à chaque thread.
- Si la variable allouée dynamiquement a un statut `shared`, il faut l'allouer hors de la région parallèle ou dans la région parallèle mais par un seul thread (clauses `master` ou `single`).

Portée d'une région parallèle

Elle s'étend :

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

L'ensemble des 2 constitue l'étendue dynamique de la région parallèle.

Fortran format libre

```
program parallel
 implicit none
!$omp parallel
 call sub()
!$omp end parallel
end program parallel
```

```
subroutine sub()
!$ use omp_lib
 implicit none
 logical :: p
 p=.false.
!$  p = omp_in_parallel()
  print *,"in_parallel dans sub vaut : ", p
end subroutine sub
```

```
> gfortran -fopenmp prog.f90 sub.f90
> export OMP_NUM_THREADS=2
> a.out
in_parallel dans sub vaut :  T
in_parallel dans sub vaut :  T
```

Portée d'une région parallèle

Statut des variables d'un sous-programme appelé dans une région parallèle :

- Les variables locales sont implicitement privées à chaque thread.
- Les variables transmises par argument héritent du statut défini dans l'étendue statique de la région parallèle.

Exemple avec des variables locales (privées) :

```
program parallel
 implicit none

!$omp parallel default(none)
   call sub()
!$omp end parallel

end program parallel
```

```
subroutine sub()
 use omp_lib
 implicit none
 integer :: a, rang
 a = 100
 rang=omp_get_thread_num()
 a = a + rang
 print *,"a vaut : ",a

end subroutine sub
```

```
> gfortran -fopenmp prog.f90 sub.f90
> export OMP_NUM_THREADS=2
> a.out
 a vaut :           101
 a vaut :           100
```

Exemple avec avec des variables passées par arguments :

```
program parallel
 implicit none

 integer :: a, b
 a = 100; b=0

!$omp parallel shared(a), private(b)
   call sub(a,b)
   print *,"b vaut : ",b
!$omp end parallel

end program parallel
```

```
subroutine sub(x,y)
 use omp_lib
 implicit none
 integer :: x, y
 y= x + omp_get_thread_num()
end subroutine sub
```

```
> gfortran -fopenmp prog.f90 sub.f90
> export OMP_NUM_THREADS=2
> a.out
 b vaut :           101
 b vaut :           100
```

Réduction

- La clause `reduction` permet de réaliser une opération associative sur une variable partagée.
- En Fortran, la syntaxe est :
  - `reduction(operateur : liste_variables)`
  - `reduction(fonction : liste_variables)`

  où
  - `operateur` est un des opérateurs suivants :
    `+ , * , - , .and. , .or. , .eqv. , .neqv.`
  - `fonction` est une des fontions intrinsèques suivantes :
    `max, min, iand, ior, ieor`
  - `liste_variables` est la liste des variables de réduction

Réduction

- Chaque thread réalise le résultat partiel de l'opération de réduction de manière indépendante. Le résultat final est alors obtenu lors de la synchronisation des threads.
- Les variables de réduction doivent être partagées dans la section parallèle associée.

Réduction

```
program parallel
 use omp_lib
 implicit none
 integer :: s

 s = 0

!$omp parallel reduction(+:s)
 s = s + omp_get_thread_num()
!$omp end parallel


 print *, "Somme des rangs des threads =", s


end program parallel
```

```
> gfortran -fopenmp prog.f90
> export OMP_NUM_THREADS=4
> a.out
Somme des rangs des threads =           6
```

Quelques conseils

En général,

- on attribue le statut `shared` aux variables accédées uniquement en lecture (R.H.S dans des opérations) ;
- on attribue le statut `private` pour les variables accédées en écriture (L.H.S) sauf :
  - pour les variables de réduction ;
  - pour les variables accédées par un seul thread (clauses `master` et `single`) ou par un thread à la fois (clauses `atomic` et `critical`).

Exercice 3

Dans cet exercice, nous allons utiliser OpenMP pour paralléliser un programme calculant π.

π peut être calculé simplement par intégration :

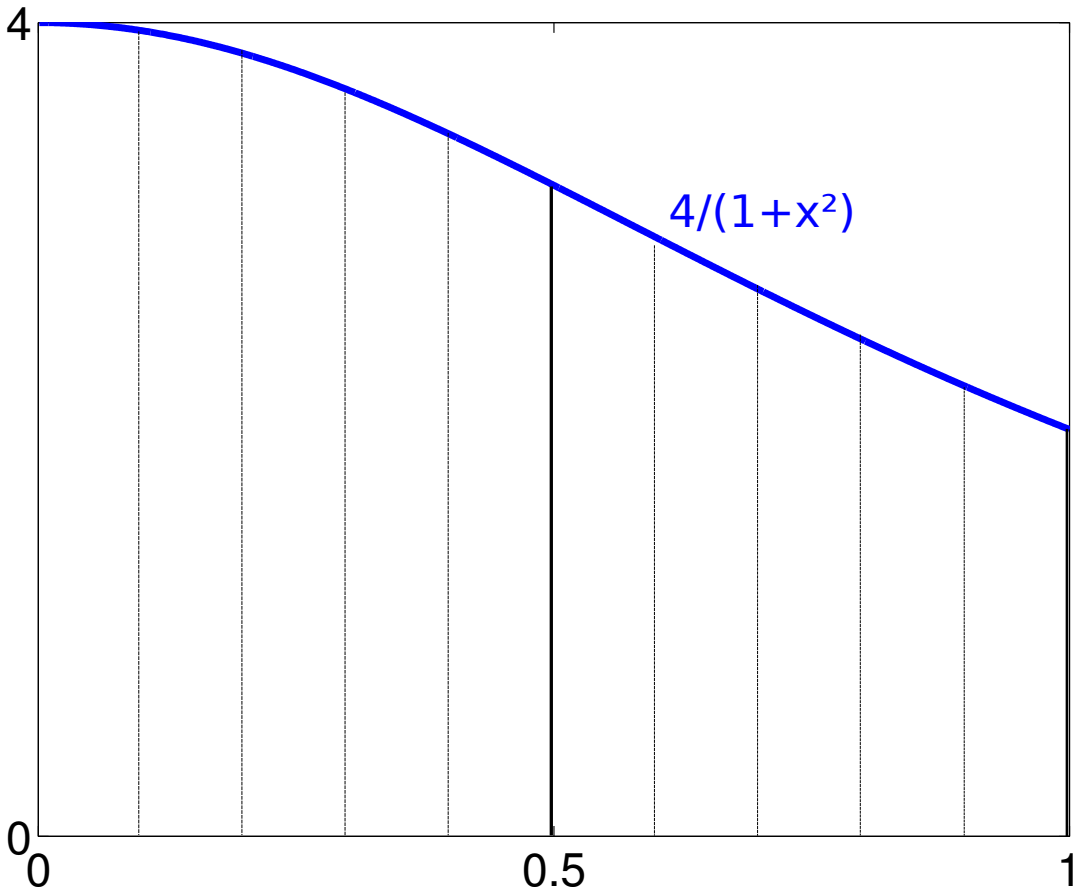
$$\pi = \int_0^1 f(x)dx \text{ avec } f(x) = \frac{4}{1+x^2}$$

Une approximation de cette relation est :

$$\pi = h\sum_{i=1}^{n} f(x_{i-1/2}) \text{ avec } h = \frac{1}{n} \text{ et } x_{i-1/2} = \frac{i-1/2}{n}$$

Exercice 3

Cette approximation correspond à l'air sous la courbe $f(x) = \frac{4}{1+x^2}$ entre 0 et 1.



Exercice 3

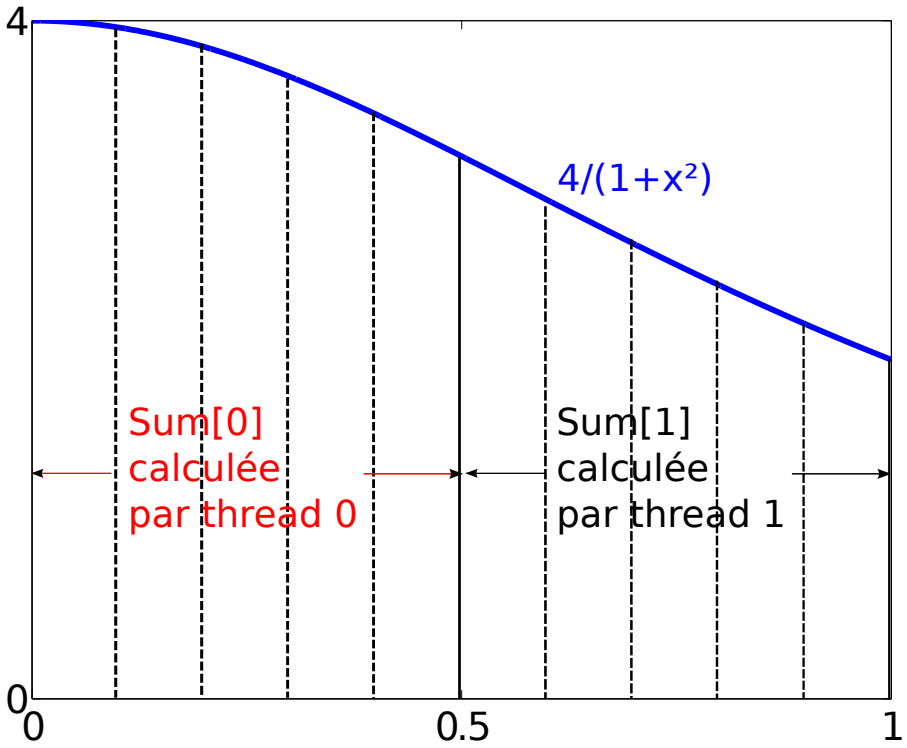
Ecrire le programme séquentiel calculant la valeur de π.

Programme séquentiel pi.c

```
! Programme Pi sequentiel
program pi_seq

  implicit none
  integer, parameter :: dp = kind(1.0d0)
  integer, parameter :: n  = 3000000
  integer :: i
  real(dp) :: h, x, pi, summ

  summ = 0.d0
  h = 1.d0 / (real(n, dp))

  do i = 1, n
     x = (real(i, dp) - 0.5d0) * h
     summ = summ + (4.d0 / (1.d0 + (x*x)))
  end do

  pi = h * summ

  print*, 'Pi = ', pi

end program pi_seq
```

Exercice 3 : parallélisation

- Parallélisation en utilisant la directive `parallel`. La répartition du travail entre les threads est réalisée par le programmeur (*cf.* figure ci-dessous, pour 2 threads)



Exercice 3

Ecrire le programme parallèle du calcul de π.

- Chaque thread calcule une partie de la somme qu'il stocke dans un tableau partagé à l'indice correspondant à son rang.
- La somme totale est calculée par le thread maître après la région parallèle.

Boucle parallèle

- Une boucle parallèle est une boucle dont les instructions ne présentent pas de dépendances entre les itérations.
- La directive `do` en Fortran permet de répartir les itérations d'une boucle entre les threads.
- Cette répartition ne s'applique qu'à la boucle suivant immédiatement la directive `do`.
- Les indices d'une boucle suivant la directive `do` sont privés par défaut.
- La clause `schedule` permet de définir le mode de répartition des itérations.
- Par défaut, une synchronisation globale est effectuée en fin de construction (sauf si la clause `nowait` a été spécifiée).

Boucle parallèle

Fortran

```
program parallel
 use omp_lib
 implicit none
 integer, parameter :: n = 10
 integer :: rang, i

 !$omp parallel private(rang)
   rang = omp_get_thread_num()
   !$omp do schedule(runtime)
   do i = 1, n
     print *, "L'iteration ", i, "s'execute &
                 sur le thread", rang
    end do
    !$omp end do
  !$omp end parallel

end program parallel
```

```
> gfortran -fopenmp boucle.f90
> export OMP_NUM_THREADS=4
> export OMP_SCHEDULE="static,2"
> a.out
L'iteration   0 s'execute sur le thread :   0
L'iteration   1 s'execute sur le thread :   0
L'iteration   2 s'execute sur le thread :   1
L'iteration   3 s'execute sur le thread :   1
L'iteration   4 s'execute sur le thread :   2
L'iteration   5 s'execute sur le thread :   2
L'iteration   6 s'execute sur le thread :   3
L'iteration   7 s'execute sur le thread :   3
L'iteration   8 s'execute sur le thread :   0
L'iteration   9 s'execute sur le thread :   0
```

Boucle parallèle : répartition des itérations

- Clause `schedule(static,chunk)` :
  les itérations sont attribuées aux threads de manière cyclique par bloc de taille `chunk` (à l'exception du dernier bloc dont la taille peut être inférieure).

```
> gfortran -fopenmp boucle.c
> export OMP_NUM_THREADS=4
> export OMP_SCHEDULE="static,3"
> a.out
L'iteration  0 s'execute sur le thread : 0
L'iteration  1 s'execute sur le thread : 0
L'iteration  2 s'execute sur le thread : 0
L'iteration  3 s'execute sur le thread : 1
L'iteration  4 s'execute sur le thread : 1
L'iteration  5 s'execute sur le thread : 1
L'iteration  6 s'execute sur le thread : 2
L'iteration  7 s'execute sur le thread : 2
L'iteration  8 s'execute sur le thread : 2
L'iteration  9 s'execute sur le thread : 3
L'iteration 10 s'execute sur le thread : 3
L'iteration 11 s'execute sur le thread : 3
L'iteration 12 s'execute sur le thread : 0
```

```
L'iteration 13 s'execute sur le thread : 0
L'iteration 14 s'execute sur le thread : 0
L'iteration 15 s'execute sur le thread : 1
L'iteration 16 s'execute sur le thread : 1
L'iteration 17 s'execute sur le thread : 1
L'iteration 18 s'execute sur le thread : 2
L'iteration 19 s'execute sur le thread : 2
```

Boucle parallèle : répartition des itérations

- Clause `schedule(dynamic,chunk)` :
  les itérations sont attribuées aux threads par bloc de taille `chunk`.
  Dès qu'un thread a fini de traiter ses itérations, un nouveau bloc lui est attribué.

- Clause `schedule(guided,chunk)` :
  les itérations sont attribuées aux threads par bloc de taille exponentionellement décroissante.
  La taille des blocs ne peut être inférieure à `chunk`. Dès qu'un thread a fini de traiter ses itérations, un nouveau bloc lui est attribué.

Boucle parallèle : répartition des itérations et compléments

- Clause `schedule(runtime)` : la répartition des itérations est décidée à l'exécution en fonction de la valeur de la variable d'environnement `OMP_SCHEDULE`.
- Compléments :
  - Il est possible de fusionner les directives `parallel` et `for` ou `do` grâce à la directive `parallel for` ou `parallel do`. Cette directive ne peut admettre la clause `nowait`.
  - La directive `for` ou `do` admet entre autres les clauses `private`, `firstprivate`, `reduction` et `lastprivate`.

Boucle parallèle : exécution ordonnée

- La clause `ordered` permet d'exécuter une boucle d'une façon ordonnée (utile pour déboguer).
- L'ordre des itérations sera alors identique à celui correspondant à une exécution séquentielle.

```
program parallel
 use omp_lib
 implicit none
 integer, parameter :: n = 10
 integer :: rang, i

 !$omp parallel private(rang)
   rang = omp_get_thread_num()
   !$omp do schedule(runtime) ordered
   do i = 1, n
     print *, "L'iteration ", i, "s'execute &
                sur le thread", rang
    end do
    !$omp end do
  !$omp end parallel

end program parallel
```

```
> gfortran -fopenmp boucle.f90
> export OMP_NUM_THREADS=4
> export OMP_SCHEDULE="static,2"
> a.out
L'iteration   0 s'execute sur le thread :   0
L'iteration   1 s'execute sur le thread :   0
L'iteration   2 s'execute sur le thread :   1
L'iteration   3 s'execute sur le thread :   1
L'iteration   4 s'execute sur le thread :   2
L'iteration   5 s'execute sur le thread :   2
L'iteration   6 s'execute sur le thread :   3
L'iteration   7 s'execute sur le thread :   3
L'iteration   8 s'execute sur le thread :   0
L'iteration   9 s'execute sur le thread :   0
```

Exercice 4

- Paralléliser le calcul de π en utilisant la directive `do` avec la clause `reduction`.

Sections parallèles

- La directive `sections` permet de répartir l'exécution de parties de code indépendantes sur différents threads.
- Les parties de code à repartir sont repérées par la directive `section` au sein de la construction `sections`.
- Par défaut, une synchronisation globale est effectuée en fin de construction (sauf si la clause `nowait` a été spécifiée).

```
$omp parallel
$omp sections

$omp section
//partie du code réalisée dans un thread

$omp section
//partie du code réalisée dans un autre thread

$omp end sections
$omp end parallel
```

Exercice 5

- Ecrire un programme parallélisant le produit scalaire de deux vecteurs.
- Chaque vecteur devra être initialisé dans un thread différent.
- Les composantes i des vecteurs devront valoir la racine carré de i.

Construction workshare

La directive `workshare` permet de répartir l'exécution de certaines construction fortran telles que les instructions sur les tableaux. de parties de code indépendantes sur différents threads.

```
program prodMatVec
  implicite none
  integer, parameter :: n
  real(dp), dimension(n,n) :: a
  real(dp), dimension(n) :: x, y

  call random_number(a)
  call random_number(x)

  !$omp parallel
  !$omp workshare
  y = matmul(a,x)
  !$omp end workshare
  !$omp end parallel

end program prodMatVec
```

Fonctions orphelines

- Une fonction ou une procédure appelée dans une région parallèle dans laquelle il existe des directives OpenMP est appelée fonction ou procédure orpheline.

```
subroutine sumVec(n, x, y, z)
implicit none
integer :: n
real(kind=kind(1.d0)), dimension(n) :: x, y, z
integer :: i

do i=1, n
  z(i) = x(i) + y(i)
end do

end subroutine sumVec
```

```
program orphan
use omp_lib
implicit none
integer, parameter :: n = 20
real(kind=kind(1.d0)), dimension(n) :: x, y, z
integer :: i

do i = 1, n
  x(i) = sqrt(dfloat(i))
  y(i) = sqrt(dfloat(i))
end do

!$omp parallel
call sumVec(n, x, y, z)
!$omp end parallel

end program orphan
```

```
> gfortran -fopenmp orphan.f90 sumVec.f90
```

Fonctions orphelines

Trois contextes d'exécution sont possibles :

L'option de compilation permettant l'interprétation des directives OpenMP :

- a été activée pour l'unité appelante et l'unité appelée : l'exécution peut être parallèle et la charge de travail de la fonction peut être distribuée aux threads ;
- a été activée pour l'unité appelante mais pas pour l'unité appelée : l'exécution peut être parallèle mais la charge de travail de la fonction est répliquée sur les threads ;
- n'a pas été activée pour l'unité appelante : l'exécution est entièrement séquentielle quelque soit le mode de compilation de l'unité appelée.

Exécution exclusive

- Les directives `master` et `single` permettent de faire exécuter une partie de code uniquement par un thread.
- La directive `master` impose qu'il s'agit du thread maître.
- En fin de construction, la directive :
  - `master` ne contient pas de synchronisation implicite.
  - `single` contient une synchronisation implicite.

Exécution exclusive

```
program exclusive
use omp_lib
implicit none
integer :: a, rang

!$omp parallel private(rang)
rang = omp_get_thread_num()
!$omp single
print *, "Entrer un entier :"
read *, a
!$omp end single

print *, "Thread", rang, " : l'entier &
                entré est", a
!$omp end parallel

end program exclusive
```

```
> gfortran -fopenmp single.c
> export OMP_NUM_THREADS=4
> a.out
 Entrer un entier:
 10
 Thread        0  : l'entier entré est     10
 Thread        2  : l'entier entré est     10
 Thread        3  : l'entier entré est     10
 Thread        1  : l'entier entré est     10
```

Synchronisations

- La directive `barrier` permet de synchroniser l'ensemble des threads dans une région parallèle.
- La directive `critical` permet d'imposer qu'une partie de programme soit exécutée par un seul thread à la fois.

Synchronisations

```
program barrier
use omp_lib
implicit none
integer :: a, rang

!$omp parallel private(rang)
rang = omp_get_thread_num()
!$omp master
print *, "Entrer un entier :"
read *, a
!$omp end master

!$omp barrier
print *, "Thread", rang, " : l'entier &
                    entré est", a
!$omp end parallel

end program barrier
```

```
> gfortran -fopenmp master.c
> export OMP_NUM_THREADS=4
> a.out
 Entrer un entier :
 10
 Thread       0  : l'entier entré est      10
 Thread       1  : l'entier entré est      10
 Thread       3  : l'entier entré est      10
 Thread       2  : l'entier entré est      10
```

Synchronisations

- La directive `atomic` permet d'imposer qu'une instruction soit exécutée par un seul thread à la fois.
- Son effet est valable uniquement sur l'instruction suivant la directive.

Synchronisations

- En fortran, l'instruction peut avoir les formes suivantes :
  - `x = x operateur expression`
  - `x = expression operateur x`
  - `x = f(expression, x)`
  - `x = f(x,expression)`

  où
  - `x` est une variable
  - `expression` est une expression scalaire indépendante de `x`
  - `operateur` est un des opérateurs suivants :
    `+, -, *, /, .and., .or. , .eqv. , .neqv.`
  - `f` est une des fonctions suivantes :
    `max, min, iand, ior, ieor`

Exercice 6

1. Ecrire un programme parallélisant le produit matrice vecteur en utilisant la directive `do` avec une clause `reduction`.
2. Ecrire un programme parallélisant le produit matrice vecteur en utilisant la directive `do` et `atomic`.
   Les éléments de la matrice seront parcouru afin de favoriser la localité spatiale.

Coût des directives

- Exemple* pour 4 threads (calculateur ECP, Intel WSM 2.66*Ghz*)

	ifort (11.1)	gfortran (4.3.4)		ifort	gfortran
`parallel`	1.54	19.67	`single`	0.44	5.06
`do`	0.47	10.30	`critical`	0.32	0.84
`barrier`	0.45	4.30	`reduction`	1.65	20.84
			`atomic`	0.06	0.08

Table: overheads (μs) à comparer au temps de cycle (0.38*ns*)

- Les compilateurs Intel sont mieux adaptés aux processeurs Intel que GNU.
- Si possible, ne pas émuler une clause `atomic` avec une clause `critical`

*EPCC OpenMP Microbenchmarks, http ://www2.epcc.ed.ac.uk/

Conseils

- Minimiser le nombre de régions parallèles en utilisant les directives `master` et `single`.
- Choisir le nombre de threads de telle sorte que le rapport entre le nombre d'instructions et le nombre de directives soit suffisamment grand pour compenser le coût des directives.
- Pour les boucles imbriquées, préférer paralléliser la boucle la plus externe.
- Eviter de paralléliser une boucle trop petite en utilisant la clause `if`.
- Ajouter des clauses `nowait` lorsque c'est possible pour éviter des synchronisations inutiles.

Références

- Site officiel : http ://openmp.org/
- Site dédié aux utilisateurs d'OpenMP : http ://www.compunity.org/
- *OpenMP. Parallélisation multitâches pour machines à mémoire partagée.* J. Chergui, P.-F. Lavallée, Cours Idris, 2008.
- *OpenMP*, F. Roch, ANGD "Calcul parallèle et application aux plasmas froids", 2011.
- *Modèles de programmation, Introduction au calcul parallèle*, G. Moebs, ANGD "Choix, installation et exploitation d'un calculateur", 2011