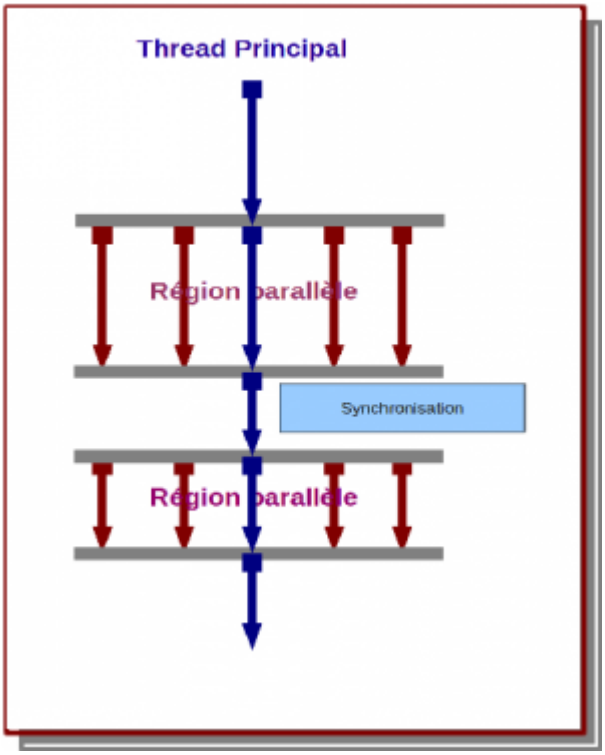
Introduction à OpenMP

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



Fonctionnement

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

Le nombre de threads peut être contrôlé à partir du programme ou en utilisant la variable d'environnement `OMP_NUM_THREADS` par exemple :

```
export OMP_NUM_THREADS=4
```

Compilation :

C/C++	Fortran
`$ icc -openmp omp_pgm.c -o pgm` `$ gcc -fopenmp omp_pgm.c -o pgm`	`$ ifort -openmp omp_pgm.f -o pgm` `$ gfortran -fopenmp omp_pgm.f -o pgm`

Région parallèle

Un bloc de code exécuté par plusieurs threads en parallèle.

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

Fortran :

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

C/C++ :

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

Les clauses possibles :

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

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

Exemples

hello.f

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

```
end program hello
```

hello.c

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

```
export OMP_NUM_THREADS=4
$./hello
hello world from 1
hello world from
hello world from 3
hello world from 2
```

Portée de variables

- **private (list)** déclare une liste de variables privées à chaque thread.
- **firstprivate(list)** comme private, force l'initialisation des variables privées à leur dernière valeur avant l'entrée dans la région parallèle
- **Default(none|private|shared)** change le statut par défaut des variables dans une région parallèle

Boucles parallèles

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

C/C++	Fortran
`#pragma omp for[clause [clause]…] for loop`	`!$omp do[clause [clause]…] do loop [!$omp end do [nowait]]`

Les clauses possibles :

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

La clause `schedule(type,[chunk])` permet de spécifier le mode de répartition des itérations sur l'ensemble de threads : « `chunk` » est la taille de chaque bloc à traiter (nombre d'itérations), « `type` » mode de distribution :

- **static** : les itérations sont réparties de manière équitable entre les threads
- **dynamic** : les itérations sont divisées en paquets de taille donnée dès qu'un thread termine ses itérations un autre paquet lui est attribué
- **guided** : comme le mode dynamic, mais «chunk» décroît exponentiellement
- **runtime** : le mode de répartition est déterminé à l'exécution,se base sur la valeur de la variable d'environnement `OMP_SCHEDULE`

Exemples

C/C++	C/C++

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

Fortran	**C/C++**

`!$omp do shared(x) private(i) schedule(runtime)`
`do i = 1, 12000`
`x(i)=a`
`end do`
	`#pragma omp parallel for private(i), shared(x) schedule`
	`(guided,10)`
	`for (i=;i<1000;i++)`
	`x(i)=f(i);`

Réduction

La réduction est une opération associative appliquée à une variable partagée.

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

Les variable doivent être partagées

- Fortran

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

- C/C++

opérateurs : `+, *, -, &, ^, |, &&, ||`

Les pointeurs et les variables de référence ne sont pas autorisés

Exemple

reduction.f

```
program parallel
 implicit none
 integer, parameter :: n=5
 integer :: i, s=, p=1, r=1

 !$OMP PARALLEL
   !$OMP DO REDUCTION(+:s) REDUCTION(*:p,r)
    do i = 1, n
      s = s + 1
      p = p * 2
      r = r * 3
    end do
   !$OMP END DO
 !$OMP END PARALLEL
 print *,"s =",s, "; p =",p, "; r =",r
end program parallel
```

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

Sections parallèles

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

Fortran	C/C++

`!$omp sections[clause[,clause]...]`
`!$omp section`
`code block`
`[!$omp section`
`another code block`
`[!$omp section`
`…]]`
`!$omp end sections[nowait]`
	`#pragma omp sections[clause [clause...]]`
	`{`
	`#pragma omp section`
	`structured block`
	`[#pragma omp section`
	`structured block`
	`…]`
	`}`

Les clauses possibles :

```
private(list)
```

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

Une manière plus simple de créer une section parallèle est de combiner les directives de région parallèle et de section parallèle Dans ce cas :

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

Exemples

Fortran	C/C++

`!$OMP PARALLEL SECTIONS`
`!$OMP SECTION`
`CALL INIT(A)`
`$OMP SECTION`
`CALL INIT(B)`
`!$OMP END PARALLEL SECTIONS`
	`#pragma omp parallel sections`
	`{`
	`#pragma omp section`
	`init(A);`
	`#pragma omp section`
	`init(B);`
	`}`

Dépendance de données

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

Observons le code suivant :

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

Pas d'erreurs de compilation, mais le résultat est FAUX ! Il y a une dépendance de donnée causée par les accès en lecture/écriture à la variable a.

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

avec Intel compiler

Existe-il une dépendance de donnée ?

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

Exemple d'application

Calcul de PI

Calcul de PI par intégration numérique (méthode des trapèzes).

$$\int_0^1 1/(1+x^2)dx \qquad (A)$$

Voici la version séquentielle :

Pi.f90

```
PROGRAM pi
    IMPLICIT NONE
        INTEGER, PARAMETER :: nb_iter=100000000
        DOUBLE PRECISION :: pas, x, pi_approchee = 0.0D0, temps
        INTEGER :: i, t1, t2, ir

        pas  = 1.0D0 / nb_iter
        CALL SYSTEM_CLOCK(count=t1)

        DO i=1, nb_iter
                x = pas * (i - 0.5D0)
                pi_approchee = pi_approchee + 4.0D0 * pas / (1.0D0 + x
* x)
        END DO

        CALL SYSTEM_CLOCK(count=t2, count_rate=ir)
        temps = REAL(t2 - t1) / ir
        PRINT '(" Valeur approchee de PI = ", F16.14)', pi_approchee
        PRINT '(" Temps d''execution : ", F7.3, " sec.")', temps
END PROGRAM pi
```

Pour la version parallèle, il suffit de paralléliser la boucle avec une réduction :

```
!$OMP PARALLEL DO PRIVATE (i,x) REDUCTION(+:pi_approchee)
```

Exécution : On utilise 8 threads pour l'exécution.

```
export OMP_NUM_THREADS=8
```

Temps d'exécution en séquentiel :

$$Ts = 1.861sec$$

Temps d'exécution en parallèle :

$$Tp = 0.239sec$$

L'accélération :

$$S = Ts/Tp = 7.78$$

Théoriquement (la loi d'amdahl) la valeur de l'accélération est définie :

$$1 <= S <= p$$

p est le nombre de threads ou processeurs.

Liens

- Introduction à OpenMP (formation mésocentre de Franche-Comte) openmp.pdf
- https://computing.llnl.gov/tutorials/openMP