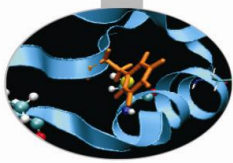
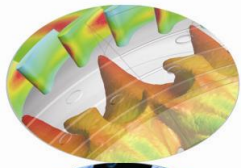
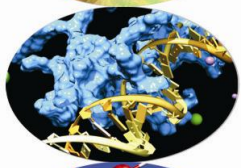
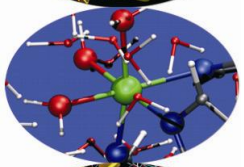
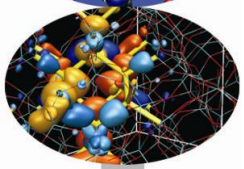
CINECA SCAI
SuperComputing Applications and Innovation



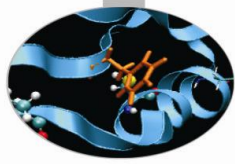
OpenMP

Introduzione al calcolo parallelo

CINECA

Parallelizzazione a memoria condivisa



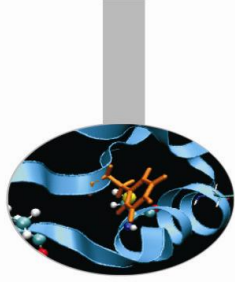
I programmi paralleli a **memoria condivisa** devono il parallelismo all'uso di direttive o alla capacità del compilatore di parallelizzare i **blocchi DO**.

In questo tipo di codici la comunicazione tra i processori avviene automaticamente, e non deve essere esplicitata come nel caso di architetture a memoria distribuita (con il message-passing).

La parallelizzazione viene ottenuta generando dei sottoprocessi (**thread**), ai quali viene assegnata parte del lavoro parallelizzabile. All'inizio solo il thread 0 è attivo, quello che esegue anche la parte sequenziale del codice. Non appena l'esecuzione diventa parallela il thread attivo si preoccupa di risvegliare tutti o parte degli altri thread.

Questo tipo di parallelizzazione è detto a ***memoria condivisa*** (shared memory) perché la memoria è accessibile da tutti i threads.

Parallelizzazione a memoria condivisa



I cicli di un loop parallelo vengono distribuiti tra i vari sottoprocessi, tuttavia l'ordine in cui le singole operazioni vengono effettuate non è determinato; ciò comporta che alcuni cicli non possano essere parallelizzati.

Il seguente loop non dà problemi:

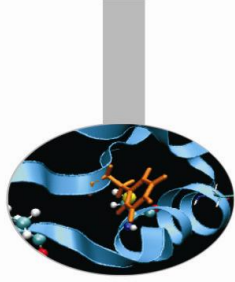
fortran
```
DO I = 1, N
            A(I) = A(I) + B(I) * C(I)
END DO
```

c/c++
```
for(i=1;i<n;i++){
        a[i] = a[i] + b[i] * c[i]
}
```

Invece nel caso seguente è fondamentale l'ordine in cui vengono calcolate le varie iterazioni, pertanto non può essere parallelizzato:

fortran
```
DO I = 1, N
           A(I) = A(I-1) + K * B(I)
END DO
```

c/c++
```
for(i=1;i<n;i++){
        a[i] = a[i-1] + k * b[i]
}
```



Data dependence

Solo il loop esterno può essere parallelizzato:

fortran
```
DO I = 1, N
         DO J = 1, N
            A(J,I) = A(J-1,I) + B(J,I)
         END DO
END DO
```

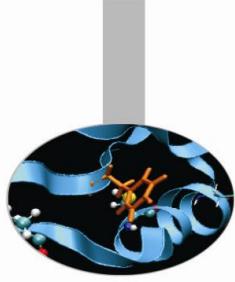
c/c++
```
for(i=1;i<n;i++){
    for(j=1;j<n;j++){
        a[j][i] = a[j-1][i] + b[j][i];
        }
    }
```

La parallelizzabilità dipende da K :

fortran
```
DO I = M, N
             A(I) = A(I-K) + B(I)/C(I)
END DO
```

c/c++
```
for(i=m;i<n;i++){
    a[i] = a[i-k] + b[i]/c[i];
    }
```

Se `K > N-M` oppure `K < M-N` il ciclo può essere parallelizzato.

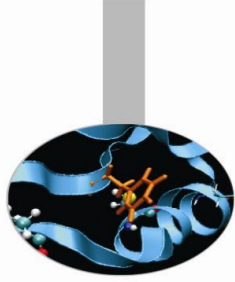


Direttive OpenMP

Per la parallelizzazione a memoria condivisa uno degli strumenti usati è **OpenMP**: è un insieme di direttive adatte ad esplicitare il calcolo parallelo in programmi Fortran e C/C++.

È stato ideato come uno standard per rendere portabile la parallelizzazione dei programmi mediante direttive ed è supportato da varie piattaforme di calcolo parallelo a memoria condivisa sia Unix che Windows.

Una volta che un codice è stato parallelizzato con le direttive **OpenMP**, può comunque essere compilato normalmente e eseguito in seriale senza modifiche perché comandi e direttive **OpenMP** vengono interpretati come commenti.



Formato delle direttive

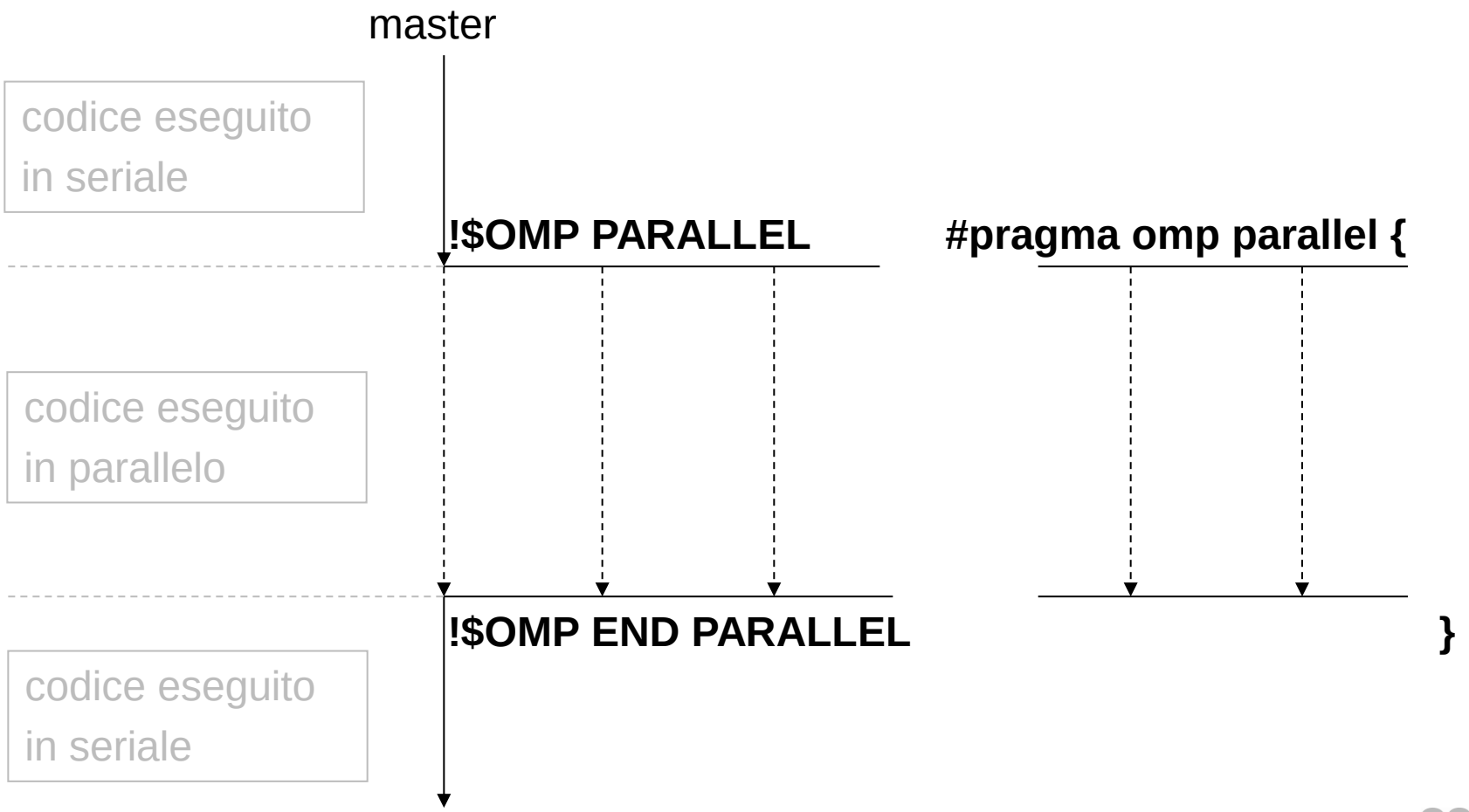
Nei programmi Fortran tutte le direttive OpenMP devono iniziare con la sentinella `!$OMP`, mentre in C/C++ le direttive devono essere inserite dopo le istruzioni `#pragma omp`.

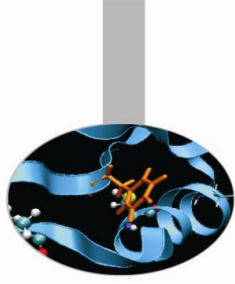
fortran
```
!$OMP
```

c/c++
```
#pragma omp
```

L'esecuzione inizia come un programma seriale: con un solo processore (il master thread) fino a che non incontra una regione parallela delimitata dalle direttive `!$OMP PARALLEL`, `!$OMP END PARALLEL` **oppure** `#pragma omp parallel { .... }` in cui anche gli altri processori diventano attivi ed eseguono la stessa istruzione su dati che possono essere diversi, distribuendosi il lavoro. In uscita dalla sezione parallela il master continua l'esecuzione seriale del codice

Modello di esecuzione





Parallel

Per definire lo spazio della regione parallela in cui tutti i threads eseguono le stesse istruzioni si usa la direttiva `parallel`:

fortran
```
!$OMP PARALLEL
…
!$OMP END PARALLEL
```

c/c++
```
#pragma omp parallel{
      istruzioni c/c++
     }
```

Questa direttiva può essere seguita da diverse opzioni:

if(scalar-expression**)**

num_threads(integer-expression**)**

default(shared | none)

private(list**)**

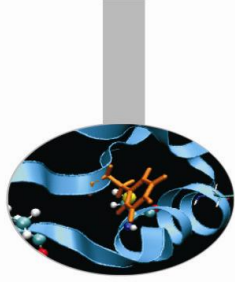
firstprivate(list**)**

shared(list**)**

copyin(list**)**

reduction(operator**:** list**)**

Do - for



Per fare in modo che il loop successivo sia distribuito tra i thread si usa la direttiva `do - for`

fortran

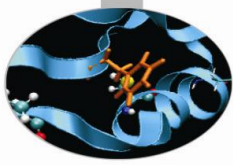
```
!$OMP DO
  DO
    istruzioni
  END DO
!$OMP END DO
```

c/c++

```
#pragma omp for
  for{
      istruzioni c/c++
    }
```

Se vengono incontrati quando non è attiva la regione parallela, allora la coppia di direttive non ha effetto, e il ciclo sarà eseguito dal master thread.

I processi che hanno finito l'esecuzione del ciclo si fermano ad aspettare che anche gli altri abbiano terminato, a meno che non venga specificata la clausola **nowait** che permette ad essi di continuare ad eseguire il codice.



Do - for

fortran

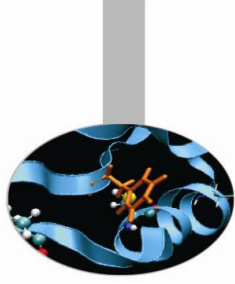
```
!$OMP PARALLEL
        .
        .
        .
 !$OMP DO
     DO I = 1, N
        A(I) = A(I) + B(I) * C(I)
     END DO
 !$OMP END DO [NOWAIT]
        .
        .
        .
 !$OMP END PARALLEL
```

c/c++

```
#pragma omp parallel
{
    .
    .
    .
    #pragma omp for
    for(i=1;i<n;i++){
          a[i] = a[i] + b[i] * c[i]
          }
    .
    .
    .
}
```

Nell'esempio il loop viene distribuito in parti uguali fra i processori ma a volte tale soluzione potrebbe non essere la più performante. Nel caso in cui risulti utile decidere a priori come suddividere il carico computazionale fra i threads, è possibile utilizzare la clausola **schedule (type [, chunk])**.

Schedule



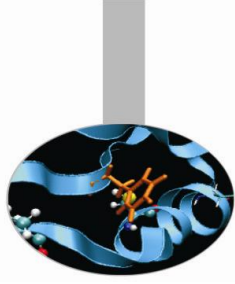
`schedule (type [, chunk])`

La variabile opzionale `chunk` deve essere un intero o un espressione scalare intera calcolata al di fuori del costrutto DO.

`Type` può essere una delle seguenti opzioni:

- **static** : le iterazioni vengono suddivise in sezioni delle dimensioni di chunk. Le sezioni sono staticamente assegnate ai threads seguendo la strategia round-robin secondo l'ordine che li identifica.
- **dynamic** : le iterazioni sono suddivise in parti di dimensioni pari a chunk. Non appena un thread termina un set di iterazioni ottiene dinamicamente il set successivo.
- **guided** : lo spazio delle iterazioni è diviso in parti tali per cui le loro dimensioni decrescono esponenzialmente fino a quella minima definita da chunk.
- **runtime** : in questo caso lo scheduling viene fatto a run time, settando prima dell'esecuzione del codice i valori di *type* e *chunk* attraverso la variabile d'ambiente OMP_SCHEDULE nel seguente modo:

```
setenv OMP_SCHEDULE "type,chunk"
```



Sections

Per delimitare le sezioni di codice che saranno eseguite da thread distinti si utilizza la clausola `sections`:

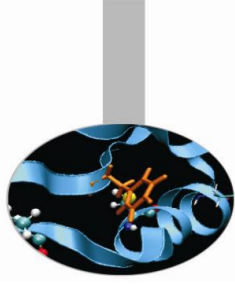
fortran

```
!$OMP SECTIONS
 !$OMP SECTION
   istruzioni fortran
 !$OMP SECTION
   istruzioni fortran
!$OMP END SECTIONS
```

c/c++

```
#pragma omp sections
{
#pragma omp section
 istruzioni c/c++
#pragma omp section
 istruzioni c/c++
...
}
```

Ogni sezione parallela di codice contenuta all'interno di tali direttive deve essere preceduta da `!$OMP SECTION` oppure da `#pragma omp section` . Se queste istruzioni vengono incontrate quando non è attiva una regione parallela non provocheranno la distribuzione del lavoro, e tutte le sezioni verranno eseguite dal thread che incontra questo costrutto.



Sections

CODICE A e CODICE B sono eseguiti da due thread diversi

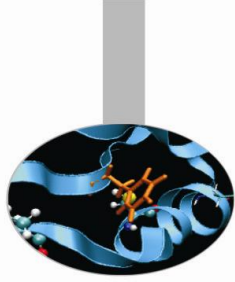
fortran

```
!$OMP PARALLEL
        .
        .
        .

 !$OMP SECTIONS

  !$OMP SECTION
    CODICE A
  !$OMP SECTION
    CODICE B
 !$OMP END SECTIONS
        .
        .
 .
 !$OMP END PARALLEL
```

c/c++

```
#pragma omp parallel
{
        .
        .
        #pragma omp sections
        {
            #pragma omp section
                CODICE A
            #pragma omp section
                CODICE
        }
}
```

Queste direttive permettono un parallelismo di tipo *funzionale:* ogni processore opera su un insieme di istruzioni differenti e indipendenti, a differenza dal parallelismo *sui dati* in cui ogni processore esegue le medesime operazioni su un differente insieme di dati (come nei loop distribuiti `DO/END DO - for`**).**



Single

Per definire una sezione che deve essere eseguita da un solo thread si utilizza la clausola `single`:

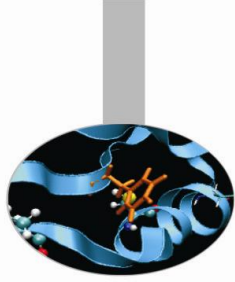
fortran
```
!$OMP SINGLE
…
!$OMP END SINGLE
```

c/c++
```
#pragma omp single{
      istruzioni c/c++
     }
```

Il primo che arriva eseguirà le istruzioni all'interno, gli altri rimarranno bloccati ad aspettare il termine dell'esecuzione di tale parte di codice, a meno che non si specifichi la clausola `NOWAIT` che permette ad essi di continuare a lavorare, saltando direttamente alle istruzioni successive.

L'utilizzo di tali direttive è obbligatorio durante l'apertura, lettura o scrittura di file, oppure se si fanno operazioni su variabili condivise da tutti i threads

Single

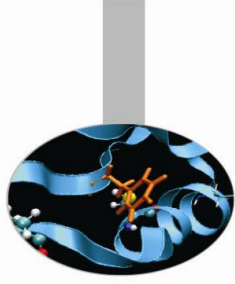


fortran

```
!$OMP PARALLEL
     …
!$OMP SINGLE
!     T1 e' in unita' di tempi dinamici
      print *,'**********************'
      print *,'STEP ', ij
      print *,'**********************'
     T1=T/TFF
     PRINT *,'t=',T1
     WRITE(18,110)T1,RL1TES,RL25TE,RL5TES
   &,RL75TE,RL9TES,RR(IR(1)),RR(IR(NPAR))
!$OMP END SINGLE
        …
!$OMP END PARALLEL
```

c/c++

```
#pragma omp parallel
{
     ...
#pragma omp single
          {
/* t1 è in unita’ di tempi dinamici*/
           cout << “********************”;
           cout << “Step “ << ij
           cout << “********************”;
           t1=t/tff
           cout << “t=“ << t1
           outfile << t1 << rl1tesrl25te <<
rl5tes << rl75te << rl9tes << rr(ir[1]) <<
rr(ir[npar]);
     }
}
```



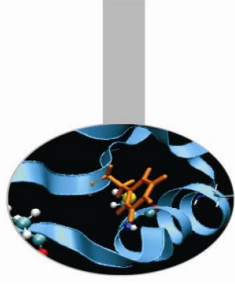
Master

la sezione racchiusa tra le direttive `master` viene eseguita solo dal master thread, gli slave thread saltano questa sezione e continuano l'esecuzione del codice. Da notare che le due direttive non implicano una barriera in entrata o in uscita dalla sezione.

fortran
```
!$OMP MASTER
…
!$OMP END MASTER
```

c/c++
```
#pragma omp master{
      istruzioni c/c++
     }
```

Parallel do – parallel for



fortran

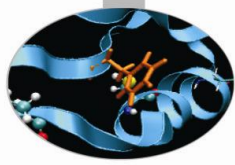
```
!$OMP PARALLEL DO
    DO
     istruzioni fortran
    END DO
!$OMP END PARALLEL DO
```

c/c++

```
#pragma omp parallel for
      for {
         istruzioni c/c++
       }
```

La direttiva `parallel` crea una regione parallela all'interno della quale verrà eseguito il do loop distribuito. A differenza di `DO/END DO` oppure `for`, non è necessario aprire prima una regione parallela con le direttive `PARALLEL/END PARALLEL` o `parallel`, ma in uscita l'esecuzione continua ad essere in seriale.

If



`IF` è una clausola che viene posta dopo le direttive `PARALLEL` `PARALLEL SECTIONS`, `PARALLEL DO`: l'esecuzione avviene in parallelo soltanto se la condizione dell'IF risulta vera, altrimenti continua in seriale.

fortran

```
!$OMP PARALLEL DO &
!$OMP IF (((N-K)<M).OR.((M-K)>N))
   DO I = M, N
     A(I) = A(I-K) + B(I)/C(I)
   END DO
!$OMP PARALLEL END DO
```

c/c++

```
#pragma omp parallel for
if (((n-k)<m) || ((m-k)>n))
{
    for(i>=m;i<n;i++)
        a[i] = a[i-k] + b[i]/c[i]
}
```

Il precedente ciclo è parallelizzabile soltanto se I-K è esterno ad un certo intervallo di valori. Può essere utile verificare se il numero di iterazioni è sufficientemente grande per avere un guadagno per l'esecuzione parallela

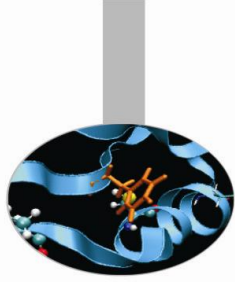
fortran

```
!$OMP PARALLEL DO IF (N>1000)
DO I=1,N
  A(i)=...
END DO
!$OMP END PARALLEL DO
```

c/c++

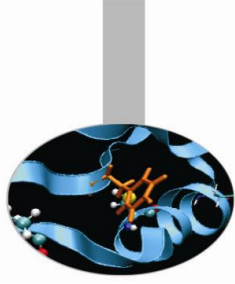
```
#pragma omp parallel for if(n>1000)
{
    for(i=1;i<n;i++)
        a[i]=...
}
```

Variabili SHARED e PRIVATE



All'interno di ogni regione parallela bisogna dichiarare esplicitamente se le variabili sono **shared**, cioè condivise da tutti i thread, oppure **private**, ovvero ogni processo ha una copia di tali variabili: per l'intera regione parallela supponendo di avere P processori, si avranno in totale P+1 copie delle variabili private, una per ogni processore, più una copia globale che è attiva fuori della regione parallela.

Ad esempio, la variabile contatore in un ciclo `DO` distribuito deve essere necessariamente "private", perché ogni processo si occuperà di un'iterazione distinta del ciclo; se invece c'è bisogno di una variabile che abbia lo stesso valore per tutti i thread (variabili di sola lettura, array le cui componenti sono distribuite tra i processori, variabili globali aggiornate durante i calcoli), allora verrà dichiarata "shared", in questo caso però il programmatore dovrà fare attenzione ai problemi di sincronizzazione.



Variabili SHARED e PRIVATE

La dichiarazione delle variabili shared va fatta nelle direttive che aprono le regioni parallele (`PARALLEL - parallel`, `PARALLEL DO - parallel for`, `PARALLEL SECTIONS - parallel sections`), le variabili private possono invece essere dichiarate anche nei `DO - for` e nelle `SECTIONS - sections`.

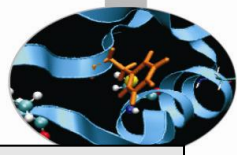
La clausola `default(shared[private])` stabilisce che tutte le variabili non dichiarate siano shared [private], oppure nel caso `default(none)` dovranno essere dichiarate tutte, sia quelle shared che quelle private; in caso di omissione nella dichiarazione il programma andrà in errore.

Per il C/C++ **non è possibile utilizzare** la clausola `default(private)`

Se nulla è specificato si assume `default(shared)`.

E' molto importante fare attenzione: infatti si ottengono risultati ben diversi se si dichiara private una variabile che deve essere shared, o viceversa, e a volte l'errore può non essere evidente.

Variabili SHARED e PRIVATE



fortran

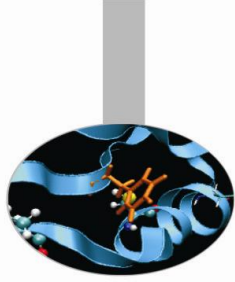
```
....
REAL, DIMENSION :: a(N)
INTEGER :: i,k
.....
!$OMP PARALLEL &
!$OMP DEFAULT(NONE) &
!$OMP SHARED (a,...) &
!$OMP PRIVATE (i,k,...)
...
!$OMP DO
  DO i=1,N
   a(i)=a(i)+k
  END DO
!$OMP END DO
.....
!$OMP END PARALLEL
```

c/c++

```
...
float a[n];
int i,k;
...
#pragma omp parallel
default(none) shared (a,...)
private(i,k,...)
{
     #pragma omp for
     for(i=1;i<n;i++)
          a[i]=a[i]+k
}
```

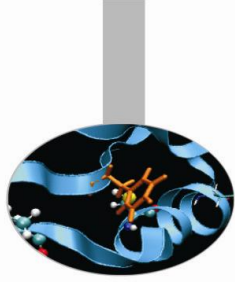
In questo esempio k è dichiarata private. Ogni processore sommerà alla componente di a che sta calcolando un valore diverso di k, perché esistono P locazioni di memoria diverse, ognuna per ogni processore, in cui sono conservate le variabili k. Se invece k fosse stata dichiarata shared, allora ad ogni componente si sarebbe aggiunto lo stesso valore. Il vettore a invece è shared, e non si hanno problemi di sovrapposizione tra i processi; infatti nel ciclo do distribuito ogni processore si occuperà soltanto dell'i-esima componente, e i è privata.

Variabili e Subroutines



Quando nella regione parallela vi sono chiamate a **subroutine** occorre ricordare che:

- Ogni thread che incontra una chiamata ad una subroutine la esegue (indipendentemente)
- Ogni variabile definita nella subroutine sara' privata al thread
- Le variabili argomenti della subroutine mantengono il loro stato originario



Firstprivate

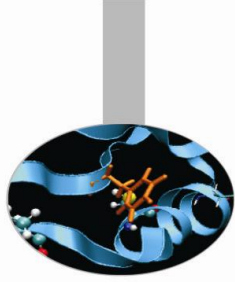
`FIRSTPRIVATE - firstprivate` inizializza la copia privata di ogni thread con il valore della variabile seriale; è una clausola che viene posta all'inizio della regione parallela. L'inizializzazione delle variabili private è cura del programmatore:

fortran

```
...
 k=a+b
!$OMP PARALLEL FIRSTPRIVATE(k)
!$OMP DO PRIVATE(i)
 DO i=1,N
   v(i)=k
   k=i+1
 END DO
!$OMP END DO
   ...
!$OMP END PARALLEL
   ...
```

c/c++

```
...
k = a+b
#pragma omp parallel firstprivate(k)
{
    #pragma omp for private(i)
    for(i=1;i<n;i++){
        v[i]=k
        k=i+1
    }
}
```



Lastprivate

`LASTPRIVATE – lastprivate` è una clausola utilizzata soltanto nei *loop distribuiti*; in uscita dalla regione parallela restituisce alla variabile seriale il valore della variabile privata dell'ultima iterazione del ciclo dell'ultimo processore.

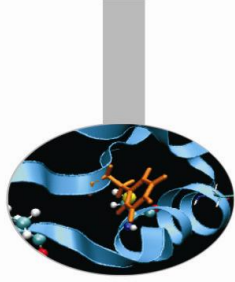
fortran

```
...
REAL, DIMENSION ::  v(N)
REAL :: b
...
!$OMP PARALLEL DO &
!$OMP PRIVATE (i) &
!$OMP LASTPRIVATE (b)
 DO i=1,N
  b=v(i)
  ...
END DO
!$OMP END PARALLEL DO
WRITE (*,*) b ! ritroveremo b=v(N)
```

c/c++

```
...
float v[n];
float b;
...
#pragma parallel for
private(i) lastprivate(b)
{
    for(i=1;i<n;i++)
        b=v[i]
}
cout << b; /* ritroveremo b=v[n]
```

Threadprivate



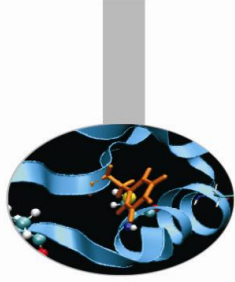
In Fortran consente una privatizzazione dei blocchi COMMON; deve essere posto subito dopo la dichiarazione del blocco.

```fortran
subroutine sub(c,n)
integer :: n
real :: x,y
real, dimension(n) :: a,b
real, dimension(n,n):: c
common /dati/ a,b
!$omp THREADPRIVATE(/dati/)
do i=1,n
 a(i)=10+i
 b(i)=5-i
end do
x=5
y=6
```

```fortran
!$omp parallel do &
!$omp default(none)&
!$omp shared(c,n)&
!$omp private (i,j,x,y) &
!$omp copyin(a,b)
 do i=1,n
   do j=1,i
     a(j)=a(i)*sin(real(i))
     b(j)=b(i)*cos(real(i))
   end do
 end do
!$omp end parallel do
end
```

Se a e b non fossero private, nel PARALLEL DO si riscontrerebbero problemi di sovrapposizione.

La direttiva **COPYIN** - **copyin** viene utilizzata per inizializzare i common privati o la lista di variabili di threadprivate.



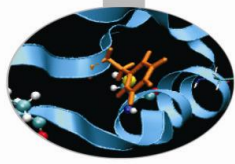
Threadprivate

In C/C++ consente la privatizzazione della lista delle variabili indicate che hanno lo scope di un blocco, di file o di un namespace e deve precedere ogni riferimento a tali variabili.

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

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

La variabile `counter` che ha scope di file deve essere dichiarata privata ad ogni processo affinchè non vi siano problemi di sovrascrittura della stessa.

Direttive di Sincronizzazione:critical



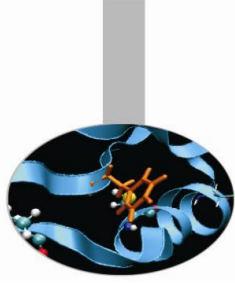
fortran
```
!$OMP CRITICAL
…
!$OMP END CRITICAL
```

c/c++
```
#pragma omp critical {
      istruzioni c/c++
      }
```

Indica una sezione di codice che viene eseguita da tutti i processi, ma in modo mutuamente esclusivo:

fortran
```
...
NMAX=k
!$OMP PARALLEL DO
 DO i=1,N
  if (a(i).gt.NMAX) then
     !$OMP CRITICAL
      if (a(i).gt.NMAX) then NMAX=a(i)
   !$OMP END CRITICAL
  end if
 END DO
!$OMP END PARALLEL DO
```

c/c++
```
...
nmax=k
#pragma omp parallel for
for(1=1;i<n;i++){
    if(a[i]>nmax){
         #pragma omp critical
             if(a[i]>nmax)
                  nmax=a[i];
    }
}
```

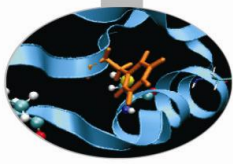


Barrier

Obbliga tutti i thread a riunirsi in un particolare punto del codice; ogni membro del team aspetta nel punto in cui è posta la `BARRIER - barrier` fino a che non siano arrivati tutti gli altri.

La clausola `NOWAIT - nowait` alla fine di un loop distribuito, come abbiamo già visto, permette ai thread che hanno finito la loro parte di iterazioni, di continuare ad eseguire le istruzioni del codice. Sicuramente questo comando minimizza i tempi di attesa dei processori, ma deve essere usato con cautela, facendo attenzione che nelle istruzioni successive non vengano usate variabili che sono aggiornate all'interno del ciclo. E' importante notare che la `BARRIER` non può essere usata all'interno dei costrutti di lavoro condiviso, o nelle sezioni critiche, altrimenti il programma termina in errore.

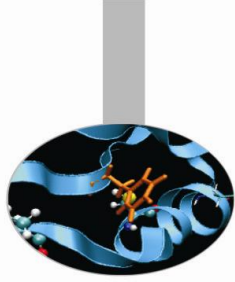
Reduction



Quando bisogna lavorare con operatori algebrici (come +, - ,*) o logici (come `.AND., .OR.,.EQ. , .NEQ. , &, ^, |, &&, ||`) oppure con le funzioni intrinseche (come `.MAX., .MIN., .IAND., .IOR., IEOR`) è possibile utilizzare `reduction`: tutti i processi fanno l'operazione in ***locale*** contemporaneamente, e poi si raggiunge un risultato ***globale*** facendo una ***riduzione*** sui risultati privati di ogni thread.

Le variabili coinvolte nella `reduction`, in cui andrà conservato il valore globale, dovranno essere dichiarate necessariamente **shared**, poi ogni processore creerà una sua variabile privata temporanea su cui fare le operazioni locali.

In questo modo c'è un notevole risparmio di tempo: se ad esempio voglio trovare il massimo su N variabili e ho `P(<<N)` processori, verranno fatte in parallelo N/P operazioni, e soltanto P operazioni in seriale.



Reduction

Per utilizzare la clausola di reduction si scrive:

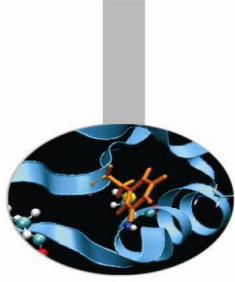
fortran
```
reduction (operatore|funzione
        intrinseca: variabile di
        riduzione)
```

c/c++
```
reduction(operatore|funzione
        intrinseca: variabile di
        riduzione)
```

Esempio di reduction per trovare il massimo di un vettore:

fortran
```
!$omp parallel do &
!$omp shared (a,N)&
!$omp private (i) &
!$omp reduction (max:maxa)
 do i=1,N
   maxa=max(a(i),maxa)
 end do
!$omp end parallel do
```

c/c++
```
Operazione non prevista nelle
Specifiche OpenMP in c/C++ perché non
esiste un'implementazione standard
della funzione "max"
```



Reduction

Quest'esempio è completamente equivalente al precedente, tranne per il fatto che fa uso delle sezioni critiche:

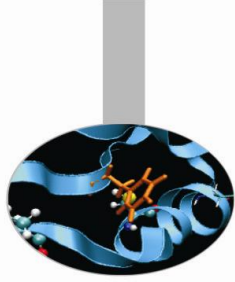
fortran

```
!$omp parallel &
!$omp shared(a,N,maxa) &
!$omp private (i,maxa_local)

maxa_local=minimo_per_il_tipo_di_maxa
!$omp do
   do i=1,N
    maxa_local=max(maxa_local,a(i))
   end do
!$omp end do nowait
!$omp critical
  maxa=max (maxa,max_local)
!$omp end critical
!$omp end parallel
```

c/c++

```
#pragma omp parallel
shared(a,n,maxa)
private(i,maxa_local)
{
maxa_local=minimo_per_il_tipo_di_maxa;
 #pragma omp for nowait
   for(i=1;i<n;i++)
     if (maxa_local > a[i])
         maxa_local=a[i];
 #pragma omp critical
 {
   if (maxa_local > max)
       max = maxa_local);
 }
}
```

Reduction



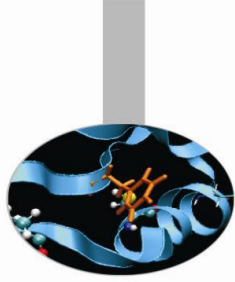
Un altro esempio che usa la reduction con la somma:

fortran

```
!$OMP DO
!$OMP REDUCTION(+:EKIN)&
!$OMP REDUCTION(+:ETERM)
        DO I=1,NPAR
          X=DBLE(P1(I))
          Y=DBLE(P2(I))
          Z=DBLE(P3(I))
          VX=DBLE(VF1(I))
          VY=DBLE(VF2(I))
          VZ=DBLE(VF3(I))
          M=DBLE(MS(I))
          .......


EKIN=.5*M*(VX*VX+VY*VY+VZ*VZ)+EKIN
          IF (I.LE.NSPH) THEN

ETERM=ETERM+MEN*DBLE(UF(I))
          END IF
        END DO
!$OMP END DO
```

c/c++

```
#pragma omp for reduction(+:ekin)
reduction(+:eterm)
    for(i=1;i<npar;i++){
          x=dble(p1[i]);
          y=dble(p2[i]);
          z=dble(p3[i]);
          vx=dble(vf1[i]);
          vy=dble(vf2[i]);
          vz=dble(vf3[i]);
          m=dble(ms[i]);
          ...
   ekin=.5*m*(vx*vx+vy*vy+vz*vz)+ekin;
   if(i<nsph)
       eterm=eterm+men*dble(uf(i));
    }
```

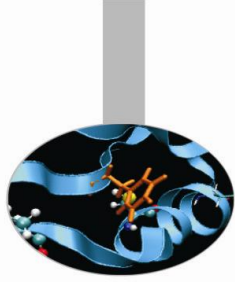


Direttive Orfane

Sono direttive parallele che si trovano in subroutine che non contengono al loro interno sezioni parallele.

Se la chiamata alla subroutine avviene all'interno di una sezione parallela le istruzioni della subroutine vengono eseguite in parallelo, altrimenti le direttive OpenMP contenute nella subroutine vengono interpretate come commenti, e la subroutine viene eseguita in seriale.

Direttive orfane



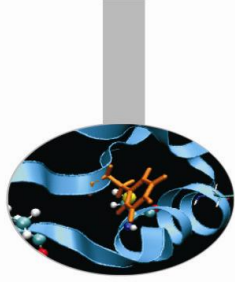
fortran

```
integer ,parameter :: N=100,M=N*100
real, dimension :: a(N)
real, dimension :: b(M)
real :: x,y
.....
do i=1,N
  a(i)=real(i)
end do
call somma (x,a,N)
!$omp parallel &
!$omp shared (b,N)&
!$omp do private(i)
do i=1,M
 b(i)=1/real(i+1)
end do
!$omp end do
```

c/c++

```
int n,m;
n=100;
m=n*100;
float a[n],b[m];
float x,y;
...
for(i=1;i<n;i++)
   a[i]=(float)i;
somma(x,a,n)
#pragma omp parallel for shared(b,n)
 private(i)
{
for(i=1;i<n;i++)
   b[i]=1/(float)(i+1);
}
```

Direttive orfane



fortran

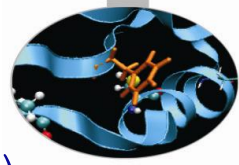
```
y=0.
call somma (y,b,M)
!$omp end parallel
....
subroutine somma(z,c,L)
integer :: i,L
real, dimension :: c(L)
real:: z
!$omp do reduction (+:z)
 do i=1,L
   z=z+c(i)
 end do
!$omp end do
end
```

c/c++

```
    y=0;
    somma(y,b,m)
}

function somma(z,c,l){
    int i,l;
    float c[l];
    float z;
    #pragma omp for reduction(+:z)
    {
          for(i=1;i<l;i++)
              z=z+c[i];
    }
}
```

All'istruzione `call somma(x,a)` la subroutine viene eseguita in seriale; invece la `call somma(y,b)` viene eseguita in parallelo perché all'interno di una regione parallela.

Funzioni Intrinseche



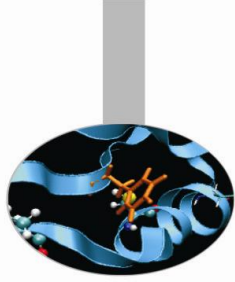
La funzione `OMP_GET_THREAD_NUM()` - `omp_get_thread_num()` ritorna un intero con l'identità del singolo processore (se P è il numero totale di processi utilizzati questa funzione può assumere un valore da 0 a P-1).

`OMP_GET_NUM_THREADS()` - `omp_get_num_threads()` ritorna il numero totale di processori utilizzati.

`OMP_SET_DYNAMIC ( logical dynamic_threads)`-
`void omp_set_dynamic(int dynamic_threads)` abilita o disabilita il settaggio dinamico del numero di threads per l'esecuzione delle regioni parallele che seguono.

`OMP_SET_NUM_THREADS(num_threads)` - `omp_set_num_threads( int num_threads)` setta il numero totale di processori da utilizzare per la regione o le regioni parallele che seguono che non hanno la clausola `num_threads`.

`double precision OMP_GET_WTIME(), double omp_get_wtime()` ritorna l'elapsed wall clock time in secondi.



Funzioni Intrinseche

In fortran la sintassi per l'utilizzo di tali funzioni è leggermente differente da quella necessaria all'inserimento delle direttive in quanto esse devono essere poste dopo la sentinella `!$`.

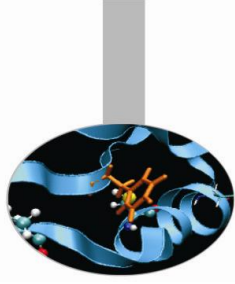
In C/C++ per utilizzare le funzioni precedenti è necessario includere la libreria **omp.h** e affinchè non vi siano problemi in compilazione conviene racchiudere le funzioni in un costrutto `#ifdef _OPENMP ..... #endif`. In questo modo viene mantenuta la possibilità di compilare il codice anche in modo seriale.

fortran

```
!$ thread_id = OMP_GET_THREAD_NUM()

!$ threads = OMP_GET_NUM_THREADS()
```

c/c++

```
#ifdef _OPENMP

threadid = omp_get_thread_num()

threads = omp_get_num_threads()

#endif
```



Compilazione

Su Linux con compilatore Intel

```
ifort   -openmp -O3 -o nomefile.exe nomefile.f90
icpc    -openmp -O3 -o nomefile.exe nomefile.cpp
icc    -openmp -O3 -o nomefile.exe nomefile.c
```

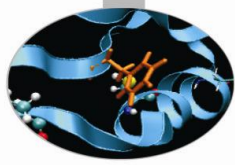
Con compilatore GNU

```
gfortran   -fopenmp -O3 -o nomefile.exe nomefile.f90
g++    -fopenmp -O3 -o nomefile.exe nomefile.cpp
gcc    -fopenmp -O3 -o nomefile.exe nomefile.c
```

Definizione del numero di processori con le variabili d'ambiente

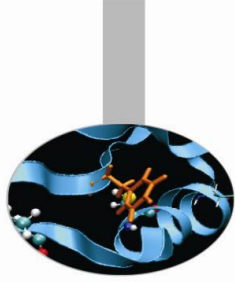
```
setenv OMP_NUM_THREADS numero di processori (shell tcsh)
export OMP_NUM_THREADS=numero di processori (shell bash)
```

Esercizi



Scrivere un programma **OpenMP** che:

1. riporti il numero totale di processori utilizzati e l'identità di ciascuno (hello.f90)
2. modifichi il programma precedente facendo uso delle SECTIONS (sezioni.f90)
3. calcoli il π attraverso l'integrale della funzione f(x)=4/(1+x2). Utilizzare una reduction sulla variabile sum.(pi.f90)
4. inizializzi una variabile in un DO LOOP distribuito con la clausola FIRSTPRIVATE (firpriv.f90), e un altro con la clausola LASTPRIVATE (lastpriv.f90).
5. da un vettore di lunghezza N=1000 ne crei uno di lunghezza N-1, calcolando per ogni elemento la media con l'elemento successivo. (filter.f90)
6. faccia il prodotto scalare tra due vettori (dotprod.f90).
7. parallelizzi il prodotto matrice *x* vettore: c'è differenza se si parallelizza il ciclo interno o quello esterno?
   (matvec.f90, matvec2.f90)
8. utilizzi la REDUCTION per fare la somma dei primi N=100 numeri (somma.f90)
9. trovi il massimo degli elementi di un vettore utilizzando le sezioni critiche, definendo per ogni thread la sezione di sua competenza (max.f90)
10. trovi il massimo degli elementi di un vettore utilizzando la REDUCTION (maxRED.f90)



Bibliografia

- Calcolo parallelo con moduli Fortran 90 e direttive HP e OpenMP: *Panoramica sulle tecnologie e sugli strumenti per la programmazione parallela (I parte)*, G. Bottoni, M. Cremonesi, Bollettino del CILEA, N. 73, giugno 2000
- Developing Multithreaded Applications: A Platform Consistent Approach
- OpenMP.org
- OpenMP Tutorial