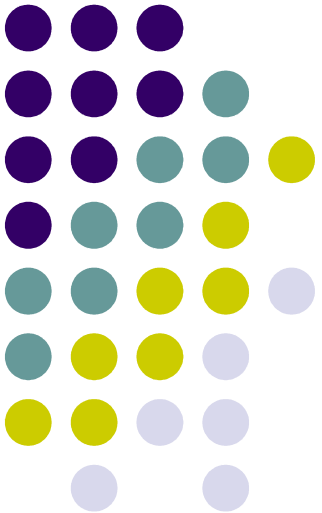
Università di Roma “La Sapienza”
Dipartimento di Matematica
15 – 31 gennaio 2007

OpenMP

Parte Prima



Fabio Bonaccorso
f.bonaccorso@caspur.it

OpenMP

- OpenMP e' un modello di programmazione parallela a memoria condivisa
- Estende le capacita' dei linguaggi di programmazione (seriali)
- L'acronimo vuol dire ”Open specifications for Multi Processing”

Caratteristiche

- OpenMP nasce per il calcolo scientifico: gran numero di calcoli ripetuti per ogni elemento base (punto di griglia, elementi finiti,modi/frequenza, ...)
- in punti limitati del codice
- Gli algoritmi conducono spesso a cicli (FOR in C, DO in Fortran)
- Con OpenMP si dividono le iterazioni dei cicli ai vari tasks (fine-grain)

Obiettivi

- Stabilire uno standard comune per le differenti architetture parallele
  - Portabile: C, C++, Fortran 77/90/95
- Ridurre al minimo i costrutti basilari
  - Semplicita' di utilizzo
- Unico codice seriale/parallelo
  - utilizzabile anche dove non c'e' supporto per OpenMP

Supporto hw e sw

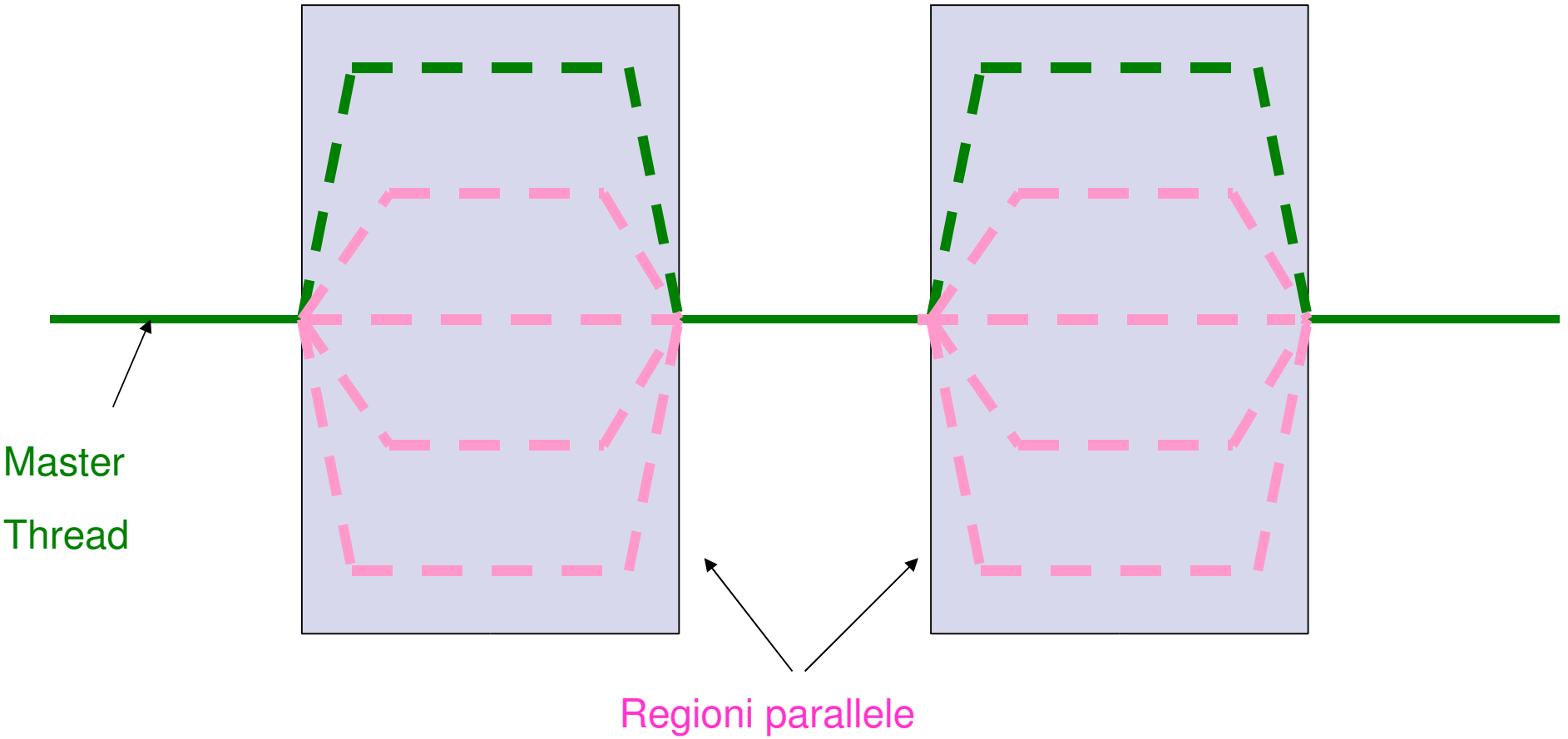
- Standard ormai consolidato supportato da tutti i maggiori fornitori di calcolatori
  - IBM, HP, Intel, SUN
- Supporto completo anche per gli ambienti software
  - Unix, Linux, Windows, Mac OS

Modello di esecuzione concettuale

- Il modello concettuale di OpenMP prevede una esecuzione alternata seriale e parallela, detta ”fork/join”
  - all'inizio e alla fine c'e' un processo seriale (master thread)
  - in alcune parti (regioni parallele) si aprono diversi filoni (threads) di esecuzione indipendenti [fork]
  - I vari threads eseguono le operazioni contemporaneamente
  - terminata la parte parallela, il master thread aspetta tutti gli altri thread [join]
  - continua in seriale (il master thread)

Modello di esecuzione concettuale

- Schematicamente:



Thread

- Unita' di esecuzione di codice minimale...
  - Concettualmente e' solo un puntatore al codice macchina (program counter)
- .... e uno spazio di memoria associato (stack)
  - se il thread richiama una subroutine, memorizza il punto di chiamata per ritornarci
  - contiene le variabili definite in una subroutine eseguita dal thread
- Il resto lo eredita dal processo: variabili globali, file aperti, ...

Threads vs processori

- Il thread e' un concetto logico
  - I sistemi operativi ne hanno una implentazione nativa
- I processori sono oggetti fisici
  - circuiti integrati
- Il numero di thread che si possono creare puo' essere differente dal numero di processori!

Thread safety

- Una subroutine e' detta thread safe se puo' essere eseguita correttamente da piu' thread contemporaneamente
- Esempi non thread safe: I/O , allocazione dinamica, ecc
- Consultare i manuali

Corretta parallelizzazione

- Lo scopo della parallelizzazione e' modificare un codice seriale in modo da distribuire il lavoro che prima veniva svolto da un processo seriale ad un insieme di unita' di esecuzione che lavorano contemporanemente e producono lo STESSO risultato
- In OpenMP le variabili originali dovranno assumere gli stessi valori al termine del calcolo

Modello di memoria

- I thread hanno spazio per variabili proprie
  - Accessibili in modalita' esclusiva
  - Memorizzate nello stack del thread
- I singoli thread ”collaborano” attraverso la memoria condivisa, la quale puo' essere letta/scritta da tutti i thread
- OpenMP fornisce solo gli strumenti per coordinare i threads, ma la corretta parallelizzazione e' compito del programmatore

Memoria: seriale e OpenMP

- Le variabili originali sono quelle definite nel programma seriale
- Una variabile di nome *x* e' condivisa (shared) se per ogni thread il nome *x* si riferisce alla stessa variabile originale *x*
- Una variabile di nome *x* e' privata se ne viene creata una nuova istanza nello stack di ogni thread
  - Stesso tipo e dimensione
  - Scorrelata da *x originale;* non inizializzata

OpenMP: come si presenta

- Direttive al compilatore
  - pensate come commenti nei vari linguaggi supportati, per preservare il seriale e i compilatori non OpenMP
- Libreria di funzioni
  - rendono il codice dipendente da OpenMP
    - Si deve includere l'elenco dei prototipi
  - permettono di determinare/alterare il numero dei thread
- Variabili di ambiente

Sintassi delle direttive

- In Fortran

!$omp *direttiva*[clause[[,] clause]...]

Blocco strutturato

  - E' un commento (inizia nella colonna 1)
  - Anche per i sorgenti ”free-form” del Fortran 90/95

- In C/C++

#pragma omp *direttiva* [clause[[,] clause]...]

Blocco strutturato

  - Usa pragma, una direttiva preprocessing ignorabile

Blocco (di codice) strutturato

- In C/C++ e' una istruzione eseguibile, anche composto, con un singolo punto di inizio e un singolo punto di uscita
  - Niente goto all'esterno, label, getjump/setjump, ...
- In Fortran e' un insieme di istruzioni eseguibili con un singolo punto di inizio e un singolo punto di uscita
- Per semplificare non devono esistere ambiguita' su inizio e fine esecuzione del blocco

I thread: fork e join

- La direttiva PARALLEL crea la regione parallela (fork), in cui oltre al master thread (che esegue il seriale) vengono creati un numero variabile di altri thread
  - in seguito vedremo come definire il numero
- Da questo momento in poi tutti i thread eseguono il codice indipendentemente
  - Tutti eseguono lo stesso codice
- Solo il master thread continua alla fine della regione parallela (join)

Parallel (sintassi)

- In C/C++

```
#pragma omp parallel [clause[ [, ]clause] ...]
   Blocco strutturato
```

- In Fortran 77/90/95

```
!$omp parallel [clause[[,] clause]...]
   Blocco strutturato
!$omp end parallel
```

- Vedremo in seguito esempi di clausole, che sono opzionali

Esempio PARALLEL

- In Fortran

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

- In C/C++

```
#pragma omp parallel
printf("Hello\n");
```

Output

- Con 5 threads

<bonaccor@poseidon ~/OMP-EX>./WriteThrNumC
Hello
Hello
Hello
Hello
Hello

- Con 2 threads

<bonaccor@poseidon ~/OMP-EX>./WriteThrNumF
Hello
Hello

Identificare i thread

- La funzione OpenMP per identificare i thread e' chiamata omp_get_thread_num
  - C/C++: int omp_get_thread_num(void);
  - Fortran: integer function omp_get_thread_num();
- Ogni thread riceve un valore di ritorno diverso
- Il master thread riceve 0
  - Gli altri thread ottengono: 1, 2, 3, ..., N-1 con N threads

omp_get_thread_num

- Come si usa?
  - Si deve prima includere un file [header]
    - in C/C++: #include "omp.h"
    - in Fortran: include "omp_lib.h"
- Dove si usa?
  - All'interno di una regione parallela
- A che serve?
  - Per poter lavorare su dati differenti per ciascun thread
- Se viene chiamata FUORI di una regione parallela?
  - Ritorna il valore 0

Esempio omp_get_thread_num

- In Fortran

```
!$OMP PARALLEL
write(*,*) "Hello from", OMP_GET_THREAD_NUM()
!$OMP END PARALLEL
```

- In C/C++

```
#pragma omp parallel
printf("Hello from %d\n", omp_get_thread_num() );
```

Output

- Con 5 threads

<bonaccor@poseidon ~/OMP-EX>./WriteThrNumC
Hello from 1
Hello from 2
Hello from 0
Hello from 3
Hello from 4
Con 2 threads
<bonaccor@poseidon ~/OMP-EX>./WriteThrNumF
Hello from 0
Hello from 1

Preservare la versione seriale

- Usando le funzioni di libreria di OpenMP la versione seriale non e' piu' preservata
  - Non compila/linka se il compilatore non implementa OpenMP

  La soluzione prevista dallo standard OpenMP e' di usare la compilazione condizionale
  - Se il compilatore non supporta OpenMP verra' ignorata
  - Differente in C e in Fortran

Compilazione condizionale

- In C/C++: Usare macro _OPENMP

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

- in Fortran: Usare il commento speciale !$

```
C234567890  Opzionale
!$ iam = omp_get_thread_num()
```

Variabili condivise

- Il modello di memoria e' shared-memory -> ogni filone di esecuzione parallela puo' leggere scrivere tutta la memoria disponibile
- Per OpenMP questo significa che tutti i thread ”vedono” le variabili del codice seriale
  - Ogni thread (nella regione parallela) puo' leggere un valore di una variabile scalare, di un vettore, di una matrice
  - Ogni thread puo' aggiornare le variabili condivise

Clausola SHARED

- Serve per dichiarare variabili condivise, ed e' una parte opzionale della direttiva PARALLEL
- In Fortran

!$OMP PARALLEL SHARED(*var list*)

Blocco strutturato

!$OMP END PARALLEL

- In C/C++

#pragma omp parallel shared(*var list*)

Blocco strutturato

Variabili private

- In aggiunta alle variabili condivise, ogni thread puo' definire variabili private inaccessibili agli altri thread
- Queste variabili sono utili per calcoli parziali
- Le variabili private sono nello stack dei thread
  - Deve esserci spazio sufficiente
  - Non inizializzate

Clausola PRIVATE

- Serve per dichiarare variabili private, ed e' una parte opzionale della direttiva PARALLEL
- In Fortran

!$OMP PARALLEL PRIVATE(*var list*)

Blocco strutturato

!$OMP END PARALLEL

- In C/C++

#pragma omp parallel private(*var list*)

Blocco strutturato

Lettura di variabile condivisa

- In Fortran

```
REAL X(10)
INTEGER I
do i=1, 10
x(i) = 2*i
enddo
!$OMP PARALLEL SHARED(X) PRIVATE(I)
I = OMP_GET_THREAD_NUM()
write(*,*) "Hello from", I
write(*,*) "X(1)=", X(1)
write(*,*) "X(", I+1, ")=", X(I+1)
!$OMP END PARALLEL
```

- Ogni thread stampa un elemento diverso dello stesso vettore X e il primo, che vale 2

Output

- Con 5 threads

```
bonaccor@poseidon:~/OMP-EX> ./Ex1
 Hello from          0
 X(1)=   2.000000
 X(          1 )=   2.000000
 Hello from          2
 X(1)=   2.000000
 X(          3 )=   6.000000
 Hello from          3
 X(1)=   2.000000
 X(          4 )=   8.000000
 Hello from          1
 X(1)=   2.000000
 X(          2 )=   4.000000
 Hello from          4
 X(1)=   2.000000
 X(          5 )=   10.00000
```

Scrittura di variabile condivisa

- In Fortran

```
REAL X(10)
INTEGER I
do i=1, 10
x(i) = 0.0
enddo
!$OMP PARALLEL SHARED(X) PRIVATE(I)
I = OMP_GET_THREAD_NUM()
write(*,*) "Hello from", I
X(I+1) = 3*(I+1)
!$OMP END PARALLEL
do i=1, 10
    write (*,*) i,x(i)
enddo
```

Output

- Con 5 threads

```
bonaccor@poseidon:~/OMP-EX> ./Ex2
 Hello from         0
 Hello from         2
 Hello from         1
 Hello from         3
 Hello from         4
          1   3.000000
          2   6.000000
          3   9.000000
          4   12.00000
          5   15.00000
          6  0.0000000E+00
            ...
         10  0.0000000E+00
```

Cambiare il numero di thread

- La variabile d'ambiente OMP_NUM_THREADS indica il numero massimo di thread che verranno creati nella regione parallela
  - L'implementazione e' libera di ignorarlo (per efficienza)

- Quindi, a riga di comando:
  - setenv OMP_NUM_THREADS 5 In csh
  - export OMP_NUM_THREADS=5 in bash

Distribuire il lavoro

- Per ripartire le iterazioni di un loop tra i thread si usa DO/for: ogni thread esegue una parte di loop
- In Fortran

```
!$OMP DO  [clause[ [, ]clause] ...]
   do-loop
!$OMP END DO [nowait]
```

- In C/C++

```
#pragma omp for  [clause[ [, ]clause] ...]
   for loop
```

Direttiva DO/for

- Si usa per un loop ”semplice”
  - La variabile di iterazione deve essere intera
  - Gli estremi devono essere UGUALI per tutti i thread
  - In C/C++ sono ammessi solo i test <,<=,>,>=
  - L'esecuzione deve finire solo alla fine del loop
    - Niente exit[F], break[C]
- Esempio loop tipico:
  - DO i = 1, N
- Deve essere all'interno di una regione parallela
  - In seguito vedremo binding dinamico

Esempio DO semplice

- In Fortran

```
REAL X(10)
INTEGER I

!$OMP PARALLEL SHARED(X)
!$OMP DO
do i=1, 10
x(i) = 2*i
enddo
!$OMP END DO
!$OMP END PARALLEL
do i=1, 10
    write (*,*) i,x(i)
enddo
```

Output

- Con 5 threads:

```
bonaccor/OMP-EX> ./Do1
    1   2.000000
    2   4.000000
    3   6.000000
    4   8.000000
    5   10.00000
    6   12.00000
    7   14.00000
    8   16.00000
    9   18.00000
   10   20.00000
```

- Con 3 threads:

```
bonaccor/OMP-EX> ./Do1
    1   2.000000
    2   4.000000
    3   6.000000
    4   8.000000
    5   10.00000
    6   12.00000
    7   14.00000
    8   16.00000
    9   18.00000
   10   20.00000
```

Le iterazioni del DO/for

- Nell'esempio precedente, il loop conteneva 10 cicli; con il costrutto DO/for si ripartiscono i 10 cicli fra i thread
  - Con 2 thread: (N1 + N2) = 10
  - Con 3 thread: (N1 + N2 + N3) = 10
  - Dove Ni e' il numero di cicli del thread i-esimo
- Qual e' lo spirito del costrutto?
  - L'indice deve assumere N valori distinti
  - L'insieme di questi valori viene partizionato tra i thread

Le iterazioni del DO/for (2)

- Nell'esempio i=1,2,...,10, l'insieme degli indici sara':

  I = {1,2,3,4,5,6,7,8,9,10} [Seriale]
- Una partizione di I su 2 thread potrebbe essere:
  - P = { {1,2,3,4,5}, {6,7,8,9,10} } [Par 2 threads]
  - P = { {1,3,5,7,9}, {2,4,6,8,10} } [Par 2 threads]
  - P = { {1,2,3,4,5,6,7,8,9}, {10} } [Par 2 threads]
  - P = { {1,2,3,4,5,6,7,8}, {9,10} } [Par 2 threads]

- Una partizione di I su 4 thread potrebbe essere:
  - P = { {1,2,3}, {4,5,6}, {7,8,9}, {10} } [Par 4 threads]
  - P = { {1,5,9}, {6,10}, {3,7}, {4,8} } [Par 4 threads]

Clausola SCHEDULE STATIC

- Nel costrutto DO/for e' possibile specificare la clausola SCHEDULE che permette di selezionare fra i tipi di partizionamento di OpenMP
- SCHEDULE(STATIC[, n])

Insiemi di n interi consecutivi assegnati una volta per tutte ai thread in base al loro numero identificativo (omp_get_thread_num) ciclicamente fino a coprire # iterazioni totali

Se n non e' specificato, le iterazioni vengono divise in insiemi di n interi consecutivi di dimensioni approssimativamente uguali e assegnati ai thread in base al loro numero identificativo

Clausola SCHEDULE DYNAMIC

- SCHEDULE(DYNAMIC[, n])

Insiemi di n interi consecutivi assegnati al primo thread che li chiede, fino a che sono terminate tutte le iterazioni

  - Ogni thread chiede un insieme e lo esegue, chiede un insieme e lo esegue, ...
  - Il partizionamento non e' prevedibile, dipende dall'esecuzione

Se n non e' specificato il default e' 1.

- SCHEDULE(GUIDED[, n])

Come DYNAMIC con insiemi decrescenti

- SCHEDULE(RUNTIME)

La scelta tra i 3 tipi disponibili e' determinata da variabili ambiente

Esempio DO/for con SCHEDULE(STATIC)

- In Fortran

```
REAL X(10)
INTEGER I
!$OMP PARALLEL SHARED(X) PRIVATE(IAM)
IAM = OMP_GET_THREAD_NUM()
!$OMP DO SCHEDULE(STATIC)
do i=1, 10
x(i) = IAM*100 + i
enddo
!$OMP END DO
!$OMP END PARALLEL
do i=1, 10
    write (*,*) i,x(i)
enddo
```

Output DO con STATIC

- Con 5 threads:

```
bonaccor/OMP-EX> ./Static
        1   1.000000
        2   2.000000
        3   103.0000
        4   104.0000
        5   205.0000
        6   206.0000
        7   307.0000
        8   308.0000
        9   409.0000
       10   410.0000
```

Thread 0,1,2,3,4: 2 iterazioni => 10

- Con 3 threads:

```
bonaccor/OMP-EX> ./Static
        1   1.000000
        2   2.000000
        3   3.000000
        4   4.000000
        5   105.0000
        6   106.0000
        7   107.0000
        8   108.0000
        9   209.0000
       10   210.0000
```

Thread 0 e 1: 4 iterazioni => 8

Thread 2: 2 iterazioni => 2_{45}

Esempio DO/for con SCHEDULE(STATIC,3)

- In Fortran

```
REAL X(10)
INTEGER I
!$OMP PARALLEL SHARED(X) PRIVATE(IAM)
IAM = OMP_GET_THREAD_NUM()
!$OMP DO SCHEDULE(STATIC,3)
do i=1, 10
x(i) = IAM*100 + i
enddo
!$OMP END DO
!$OMP END PARALLEL
do i=1, 10
      write (*,*) i,x(i)
enddo
```

Output DO con STATIC,3

- Con 5 threads:

```
bonaccor/OMP-EX> ./Static3
        1   1.000000
        2   2.000000
        3   3.000000
        4   104.0000
        5   105.0000
        6   106.0000
        7   207.0000
        8   208.0000
        9   209.0000
       10   310.0000
Thread 0,1,2:          3 iterazioni      => 9
Thread 3:              1 iterazioni      => 1
Thread 4:              0 iterazioni      => 0
```

- Con 3 threads:

```
bonaccor/OMP-EX> ./Static3
        1   1.000000
        2   2.000000
        3   3.000000
        4   104.0000
        5   105.0000
        6   106.0000
        7   207.0000
        8   208.0000
        9   209.0000
       10   10.00000
Thread 0:              4 iterazioni      => 4
Thread 1,2:            3 iterazioni      => 6
```

Esempio DO/for con SCHEDULE(DYNAMIC)

- In Fortran

```
REAL X(10)
INTEGER I
!$OMP PARALLEL SHARED(X) PRIVATE(IAM)
IAM = OMP_GET_THREAD_NUM()
!$OMP DO SCHEDULE(DYNAMIC)
do i=1, 10
x(i) = IAM*100 + i
enddo
!$OMP END DO
!$OMP END PARALLEL
do i=1, 10
      write (*,*) i,x(i)
enddo
```

Output DO con DYNAMIC

- Con 4 threads:

```
bonaccor/OMP-EX> ./Dynamic
        1   201.0000
        2   2.000000
        3   303.0000
        4   4.000000
        5   5.000000
        6   6.000000
        7   7.000000
        8   8.000000
        9   9.000000
       10   110.0000
Thread 0:          7 iterazioni     => 7
Thread 1,2,3: 1 iterazioni     => 3
```

- Altro run (4 threads)

```
bonaccor/OMP-EX> ./Dynamic
        1   1.000000
        2   202.0000
        3   3.000000
        4   4.000000
        5   5.000000
        6   6.000000
        7   7.000000
        8   8.000000
        9   9.000000
       10   10.00000
Thread 0:          9 iterazioni     => 9
Thread 1,3:        0 iterazioni     => 0
Thread 2:          1 iterazioni     => 1
```

Barriera

- La barriera e' un meccanismo di sincronizzazione:
  - un task si ferma quando arriva ad una barriera
  - Non appena tutti i task arrivano alla barriera ...
  - .. l'elaborazione può ripartire
- Serve a preservare la coerenza con l'algoritmo seriale
  - Il lavoro che precede la barriera e' stato completato
- In OpenMP il task e' il thread
- Ogni thread compie lavoro utile (calcolo) oppure attende (alle barriere)
  - Minore attesa <=> piu' efficienza parallela

Direttiva BARRIER

- In Fortran

```
!$OMP BARRIER
```

- In C/C++

```
#pragma omp barrier
```

Barriere implicite

- END PARALLEL
- In alcuni costrutti OpenMP sono previste barriere implicite
  - Per proseguire in seriale, si devono terminare la attivita' dei thread
  - END DO
  - Scelta conservativa per preservare il loop
  - Si puo' evitare con la clausola NOWAIT

Esempio BARRIER

- In C

```
#include <omp.h>
#include <stdio.h>
int main( )
{
    int a[5], i;
    #pragma omp parallel
    {
        // Perform some computation.
        #pragma omp for
        for (i = 0; i < 5; i++)
            a[i] = i * i;
        // Print intermediate results.
        #pragma omp master
            for (i = 0; i < 5; i++)
                printf("a[%d] = %d\n", i, a[i]);
        // Wait.
        #pragma omp barrier
        // Continue with the computation.
        #pragma omp for
        for (i = 0; i < 5; i++)
            a[i] += i;
    }
}
```

Critical

- La direttiva CRITICAL serve per eseguire una parte di codice un thread alla volta
- Protegge l'accesso alle variabili SHARED
- NON prevede una barriera implicita alla fine:
  - Ogni thread entra nella parte di codice "critica"
  - Esegue le istruzioni
  - Terminata questa parte, continua senza attendere gli altri thread

Direttiva CRITICAL

- In Fortran

```
!$OMP CRITICAL
    Blocco strutturato
!$OMP END CRITICAL
```

- In C/C++

```
#pragma omp critical
    Blocco strutturato
```

Esempio CRITICAL

- In Fortran

```
REAL X
X = 0

!$OMP PARALLEL SHARED(X)
!$OMP CRITICAL
      X  = X + 1
enddo
!$OMP END CRITICAL
!$OMP END PARALLEL

WRITE (*,*) “X=“, X
```

Master

- La direttiva MASTER serve per far eseguire una parte di codice solo al master thread
  - Esempi: I/O
  - NON prevede una barriera implicita alla fine
- In Fortran

```
!$OMP MASTER
    Blocco strutturato
!$OMP END MASTER
```

- In C/C++

```
#pragma omp master
    Blocco strutturato
```

Atomic

- La direttiva ATOMIC permette di aggiornare una variabile un thread alla volta
  - Piu' veloce di CRITICAL
  - Limitata ad una sola istruzione
- In Fortran

```
!$OMP ATOMIC
   assegnazione
```

- In C/C++

```
#pragma omp atomic
    assegnazione
```

ORDERED

- Se in un loop DO/for l'ordine in cui TUTTI i thread devono eseguire un blocco e' importante si usa la clausola ORDERED
- In Fortran

```
!$OMP ORDERED
    Blocco strutturato
!$OMP END ORDERED
```

- In C/C++

```
#pragma omp ordered
    Blocco strutturato
```