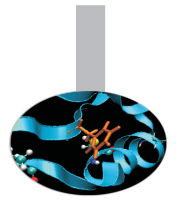
SuperComputing Applications and Innovation

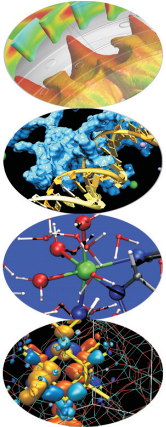


CALCOLO PARALLELO con OpenMP

Modulo 1

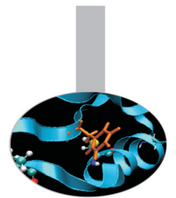
Claudia Truini Luca Ferraro Vittorio Ruggiero

CINECA Roma - SCAI Department



SuperComputing Applications and Innovation

Outline



Concetti di base
- Introduzione
- Il modello di esecuzione
- Ingredienti principali
- Come funziona

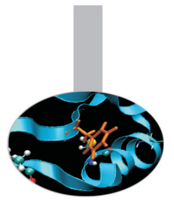
Parallelizzare un calcolo scalare

Parallelizzare un calcolo su una griglia

Parallelizzazione di un codice N-Body

Altre funzionalità

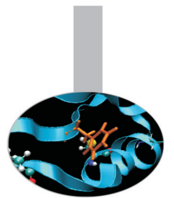
Limiti di MPI



- Ogni processo MPI può accedere soltanto alla sua memoria locale
  - I dati da condividere devono essere scambiati con comunicazioni esplicite tra processi (messaggi)
  - È onere del programmatore progettare ed implementare lo scambio dei dati tra i processi
- Non è possibile adottare una strategia di parallelizzazione incrementale
  - Deve essere implementata la struttura di comunicazione dell'intero programma
- Le comunicazioni hanno un costo
- È difficile avere il programma seriale e quello MPI in un'unica versione del codice
  - Sono necessarie variabili aggiuntive
  - È necessario gestire la corrispondenza tra variabili locali e struttura dati globale
  - La gestione del sorgente è più complessa

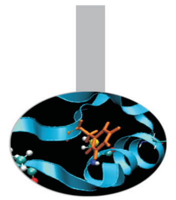
CINECA SCAI
SuperComputing Applications and Innovation

Cos'è OpenMP



- "*Open specifications for Multi Processing*" (`http://www.openmp.org/`)
- Modello di programmazione parallela per architetture a memoria condivisa (*shared memory*)
- I "lavoratori" che compiono il lavoro in parallelo (*thread*) "cooperano" attraverso la memoria condivisa
  - Accessi alla memoria invece di messaggi espliciti
  - Modello di parallelizzazione "locale" del codice seriale
- Permette la parallelizzazione incrementale
  - Riduce il tempo di *startup* della parallelizzazione
  - Un unico sorgente per il seriale e il parallelo!

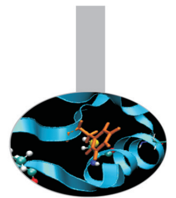
La storia di OpenMP



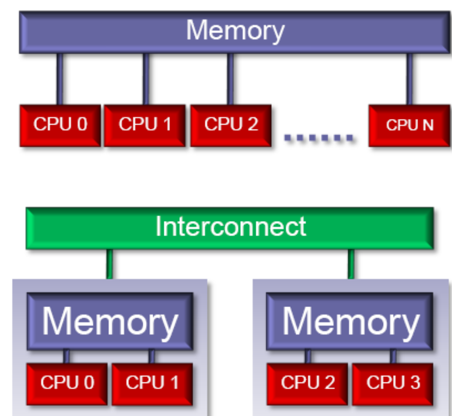
- Nato come unificazione di soluzioni proprietarie
- Il passato:
  - ottobre 1997 - versione 1.0 Fortran
  - ottobre 1998 - versione 1.0 C/C++
  - novembre 1999 - versione 1.1 Fortran (interpretazioni)
  - novembre 2000 - versione 2.0 Fortran
  - marzo 2002 - versione 2.0 C/C++
  - maggio 2005 - versione 2.5 unificazione Fortran/C/C++
  - maggio 2008 - versione 3.0 (*task*!)
  - Luglio 2011 - versione 3.1 released
- Il presente:
  - task final
  - atomics e C/C++ min max reduction
  - initial support for thread binding
  - many clarifications, improvements to examples (100 pages)
- Il futuro:
  - novembre 2013 - versione 3.5/4.0
    - support for accelerators (GPUs, MIC, FPGA, etc)
    - many improvements on tasks and atomics
    - custom *reduction* operations

CINECA SCAI
SuperComputing Applications and Innovation

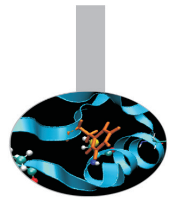
Architetture *Shared Memory*



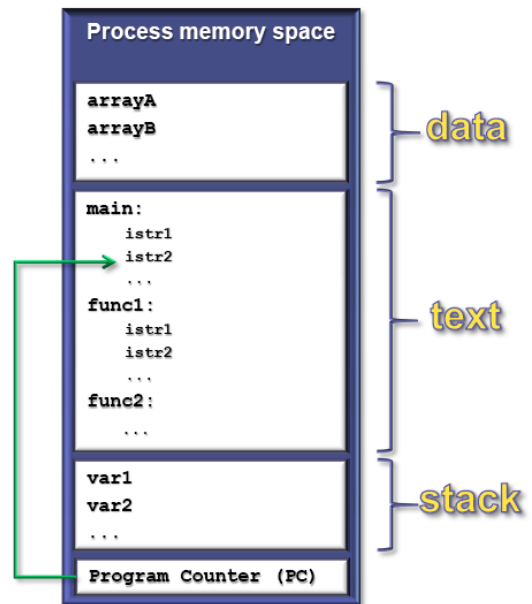
- Visione logica
  - La memoria è accessibile da tutti i processori del computer
- Visione fisica
  - *Uniform Memory Access*
    - I tempi di accesso in memoria sono gli stessi per qualsiasi CPU
  - *Non Uniform Memory Access*
    - I tempi di accesso in memoria dipendono dalla posizione della CPU rispetto alla locazione di memoria



Processo e *thread*

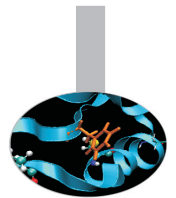


- Il processo è un'istanza del programma in esecuzione
- Alcune informazioni contenute nel processo:
  - *Text*
    - codice macchina
  - *Data*
    - variabili globali
  - *Stack*
    - variabili locali alle funzioni
  - *Program Counter* (PC)
    - puntatore all'istruzione corrente

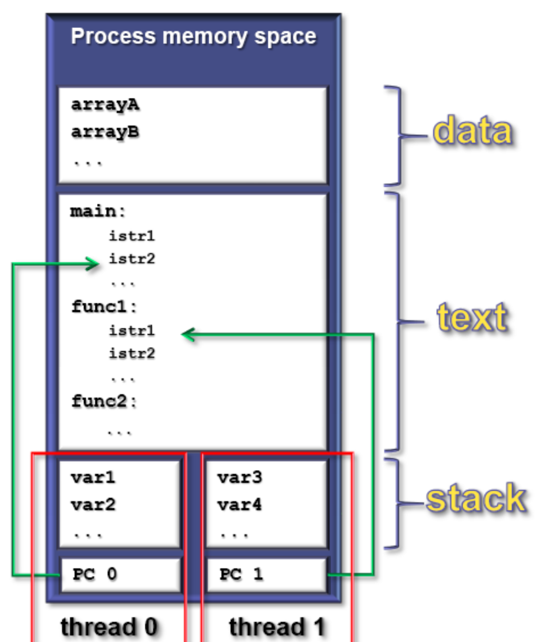


CINECA SCAI
SuperComputing Applications and Innovation

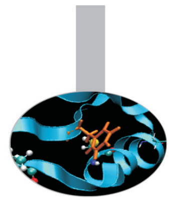
Multi-threading



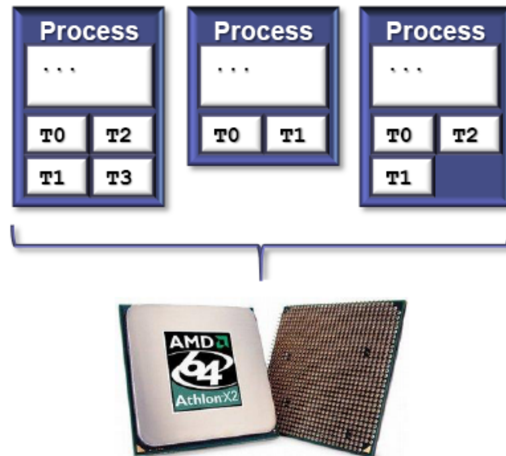
- Il processo contiene diversi flussi di esecuzione concorrenti (*thread*)
  - Ogni thread ha il suo *program counter* (PC)
  - Ogni thread ha il suo *stack* privato (variabili locali al *thread*)
  - Le istruzioni "puntate" dal PC di un *thread* possono accedere:
    - Alla memoria globale del processo (*data*)
    - Allo *stack* locale del *thread*



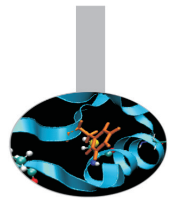
Processori, processi e *thread*

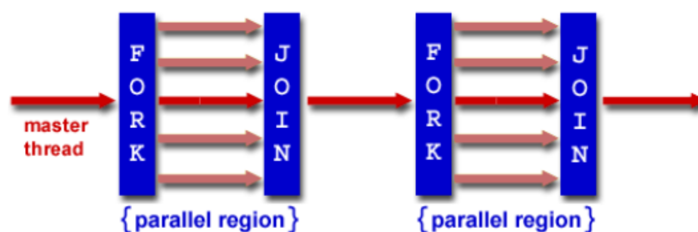


- I processori sono entità fisiche composti da uno o più *core*
- Processi e *thread* sono entità logiche



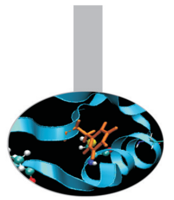
OpenMP: modello *fork & join*





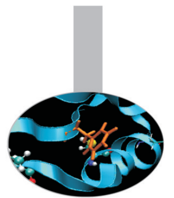
- Ogni programma OpenMP inizia con un solo *thread* di esecuzione che esegue il programma in seriale
- All'inizio di una regione parallela il *thread* crea un *team* di *thread* composto da se stesso (*Master Thread*) e da un insieme di altri *thread*
- Tutti i membri del *team* di *thread* (MT compreso) eseguono in parallelo il codice contenuto nella regione parallela (modello SPMD: *Single Program Multiple Data*)
- Al termine della regione parallela i *thread* del gruppo terminano l'esecuzione e solo MT continua l'esecuzione (seriale) del programma

Direttive



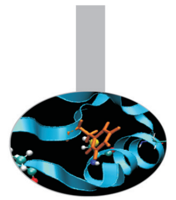
- Sintassi:
  - in C/C++:
    `#pragma omp direttiva`
  - in Fortran (free format):
    `!$omp direttiva`
  - in Fortran (fixed format):
    `c$omp direttiva`
- Annotano un blocco di codice
- Specificano al compilatore come eseguire in parallelo il blocco di codice
- Il codice seriale “convive” con quello parallelo
  - Una compilazione seriale ignora le direttive
  - Una compilazione con supporto di OpenMP ne tiene conto

Clausole



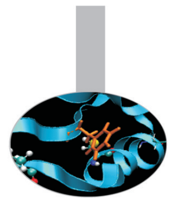
- Sintassi:
  - ***direttiva*** `[`***clausola***`[` ***clausola***`]...]`
- Specificano informazioni addizionali alle direttive
- Gestione delle variabili
  - Quali sono condivise tra tutti i *thread* (è il *default*)
  - Quali sono private ad ogni *thread*
  - Come inizializzare quelle private
  - Quale è il *default*
- Controllo dell’esecuzione
  - Quanti *thread* nel *team*
  - Come distribuire il lavoro
- ATTENZIONE: alterano la semantica del codice
  - Il codice può essere corretto in seriale ma non in parallelo
  - O viceversa
  - O cambiare comportamento col numero di *thread*

Variabili di ambiente e funzioni



- Variabili di ambiente
  - **`OMP_NUM_THREADS`**: imposta il numero di *thread* che devono essere creati in ogni regione parallela
  - **`OMP_DYNAMIC`**: consente di cambiarne il numero automaticamente
  - ...
- Funzioni
  - Determinano/modificano alcuni aspetti dell'ambiente di esecuzione
  - Permettono di gestire accessi a grana fine (*lock*)
  - Misurano il tempo trascorso
  - ...ma il codice non è più compilabile in seriale...
- Interfaccia alle funzioni:
  - Fortran: **`use omp_lib`**
  - C: **`#include <omp.h>`**

Prodotto matrice matrice

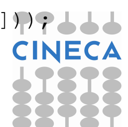


Fortran

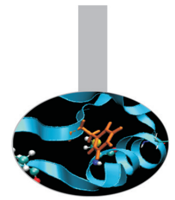
```
program mat_prod
  implicit none

  integer, parameter :: n=500
  integer, parameter :: dp=kind(1.d0)
  real(dp), dimension(n,n) :: a, b, c
  real :: maxd
  integer :: i, j, k

  call random_number(a)
  call random_number(b)
  c = 0.d0

!$omp parallel
!$omp do
  do j=1, n
    do k=1, n
      do i=1, n
        c(i,j) = c(i,j) + a(i,k)*b(k,j)
      end do
    end do
  end do
!$omp end do
!$omp end parallel

!check the result
  maxd = maxval(abs(c-matmul(a,b)))
  write( *,"(e18.9)" ) maxd

end program mat_prod
```

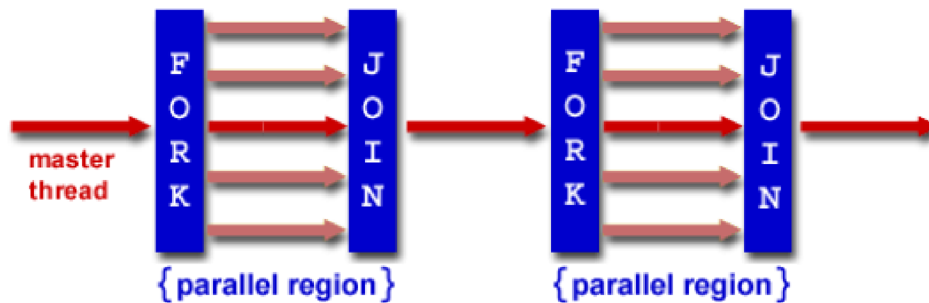
C

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc,char **argv) {
  const unsigned n=500;
  double a[n][n],b[n][n],c[n][n],d[n][n];
  double rmax = (double)RAND_MAX;
  int i, j, k;
  for (i=0; i<n; i++)
    for (j=0; j<n; j++) {
      a[i][j] = ((double)rand())/rmax;
      b[i][j] = ((double)rand())/rmax;
      c[i][j] = 0.0;
    }
#pragma omp parallel
#pragma omp for private(j,k)
  for (i=0; i<n; ++i)
    for (k=0; k<n; k++)
      for (j=0; j<n; ++j)
        c[i][j] += a[i][k]*b[k][j];
// check the result
  dgemm_(...);
  double maxd = 0.0;
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      maxd=fmax(maxd,fabs(c[i][j]-d[j][i]));
  printf("%18.9lE\n", maxd);
  return 0;
}
```

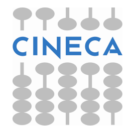


costrutto `parallel`



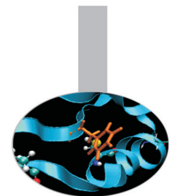


- È il costrutto fondamentale di OpenMP e serve a creare una regione parallela
  - Un costrutto è l’ambito lessicale a cui si applica una direttiva nel codice sorgente
  - Una regione è l’ambito *dinamico* dell’esecuzione del codice contenuto in un costrutto
- Ogni *thread* in una regione parallela esegue il codice contenuto e invocato al suo interno



CINECA SCAI
SuperComputing Applications and Innovation

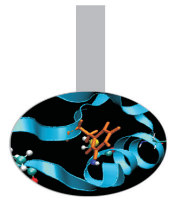
Il costrutto *do/for*

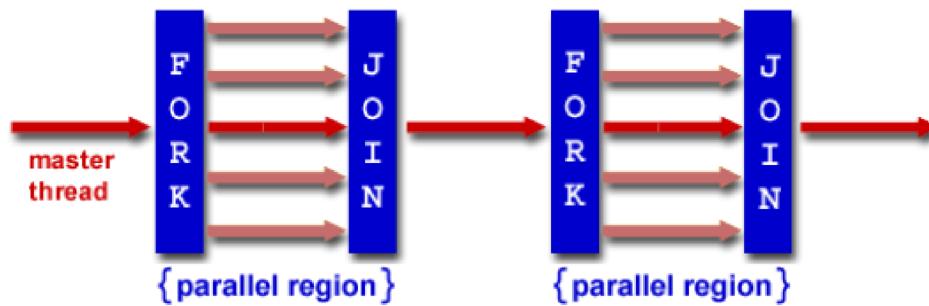


- Distribuisce le iterazioni del *loop* tra i *thread* nel *team*
- La variabile del *loop* è resa privata automaticamente
- Ogni *thread* aggiorna componenti diverse di `c`

- I loop interni sono eseguiti in modo sequenziale da ogni *thread*
- Attenzione!
  - In Fortran, le variabili di iterazione dei loop sequenziali contenuti all’interno di un costrutto parallelo sono private per *default*
  - In C no...

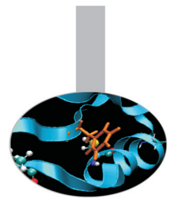
Regioni parallele: variabili **`shared`** e **`private`**





- All’interno di una regione parallela, le variabili del programma seriale possono essere *shared* o *private*
- Variabili *shared*: ogni *thread* può leggerle/scriverle
- Variabili *private*: ogni *thread* ha una sua copia privata
  - È invisibile agli altri *thread*
  - Non è inizializzata
- Per *default* le variabili sono *shared*, ma è possibile cambiarlo con una clausola:
  - **`default(none)`** impedisce ogni assunzione
  - **`default(private)`** in Fortran le rende *private* per *default*

Forma contratta

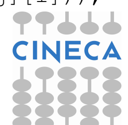


Fortran

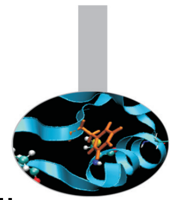
```
program mat_prod
  implicit none

  integer, parameter :: n=500
  integer, parameter :: dp=kind(1.d0)
  real(dp), dimension(n,n) :: a, b, c
  real :: maxd
  integer :: i, j, k

  call random_number(a)
  call random_number(b)
  c = 0.d0

!$omp parallel do
  do j=1, n
    do k=1, n
      do i=1, n
        c(i,j) = c(i,j) + a(i,k)*b(k,j)
      end do
    end do
  end do
!$omp end parallel do

!check the result
  maxd = maxval(abs(c-matmul(a,b)))
  write( *,"(e18.9)" ) maxd

end program mat_prod
```

C

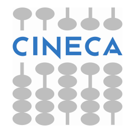
```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc,char **argv) {
  const unsigned n=500;
  double a[n][n],b[n][n],c[n][n],d[n][n];
  double rmax = (double)RAND_MAX;
  int i, j, k;
  for (i=0; i<n; i++)
    for (j=0; j<n; j++) {
      a[i][j] = ((double)rand())/rmax;
      b[i][j] = ((double)rand())/rmax;
      c[i][j] = 0.0;
    }
#pragma omp parallel for private(j,k)
  for (i=0; i<n; ++i)
    for (k=0; k<n; k++)
      for (j=0; j<n; ++j)
        c[i][j] += a[i][k]*b[k][j];
// check the result
  dgemm_(...);
  double maxd = 0.0;
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      maxd=fmax(maxd,fabs(c[i][j]-d[j][i]));
  printf("%18.9lE\n", maxd);
  return 0;
}
```



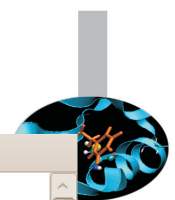
Compilazione ed esecuzione



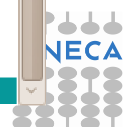
- I compilatori che supportano OpenMP interpretano le direttive solo se vengono invocati con un'opzione (*switch*) di compilazione (**`-fopenmp`** per il compilatore GNU)
- Ad esempio il compilatore Intel
  - In C: **`icc -openmp ...`**
  - In Fortran: **`ifort -openmp ...`**
- Il numero di *thread* di esecuzione si definisce a *runtime* impostando la variabile di ambiente **`OMP_NUM_THREADS`**. Ad esempio per girare un programma OpenMP con 4 *thread*:
  - In csh/tcsh: **`setenv OMP_NUM_THREADS 4`**
  - In sh/bash: **`export OMP_NUM_THREADS=4`**
- Parallelizziamo e compiliamo il programma *prodottoMM*
  - **`ifort -openmp prodottoMM.f90 -o prodottoMMf`**
  - **`icc -openmp prodottoMM.c -o prodottoMMc`**
- Eseguiamolo variando il numero di *thread*:
  - **`setenv OMP_NUM_THREADS n`**
  - **`prodottoMMf;prodottoMMc`**



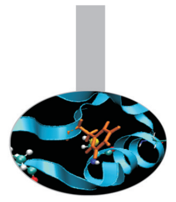
Un comando utile: `htop`



```
File  Edit  View  Terminal  Help

  1  [                              0.0%]   Tasks: 178 total, 1 running
  2  [||                            1.3%]   Load average: 0.38 0.42 0.36
  3  [                              0.0%]   Uptime: 20:18:53
  4  [                              0.0%]
  5  [                              0.0%]
  6  [|||||||||||||                32.9%]
  7  [|                             0.7%]
  8  [||||                          9.8%]
  Mem[||||||||||||||||||||||||3317/16058MB]
  Swp[|                         0/15624MB]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
10621 root       15   0 75888 20128  4476 S  0.0  0.1  0:00.67 /usr/bin/dsmc sched
10622 root       19   0 75888 20128  4476 S  0.0  0.1  0:00.00 /usr/bin/dsmc sched
 9756 ntp        15   0 19828  5524  4420 S  0.0  0.0  0:00.01 ntpd -u ntp:ntp -p /var/run/ntp
 3260 root       18   0 99984  4092  3272 S  0.0  0.0  0:00.02 sshd: botti [priv]
29196 root       21   0   97M  4056  3240 S  0.0  0.0  0:00.03 sshd: afilippi [priv]
 9131 root       22   0 99204  3952  3148 S  0.0  0.0  0:00.02 sshd: ruggiero [priv]
 7920 root       22   0 99204  3948  3144 S  0.0  0.0  0:00.02 sshd: coletta [priv]
 8180 root       19   0 99204  3956  3144 S  0.0  0.0  0:00.02 sshd: coletta [priv]
 8989 root       20   0 99984  3948  3144 S  0.0  0.0  0:00.03 sshd: lgontran [priv]
 9306 root       21   0 99204  3944  3132 S  0.0  0.0  0:00.02 sshd: tacconi [priv]
 9515 root       18   0 99204  3932  3132 S  0.0  0.0  0:00.02 sshd: ruggiero [priv]
 1828 root       23   0 99204  3936  3128 S  0.0  0.0  0:00.03 sshd: bbetti [priv]
 5905 root       18   0 99204  3936  3128 S  0.0  0.0  0:00.02 sshd: valorani [priv]
 6528 root       18   0 99204  3936  3128 S  0.0  0.0  0:00.02 sshd: aschi [priv]
 8472 root       20   0 99204  3940  3128 S  0.0  0.0  0:00.03 sshd: ruggiero [priv]
 8624 root       18   0 99200  3936  3128 S  0.0  0.0  0:00.02 sshd: rorro [priv]
 1563 root       18   0 99204  3928  3124 S  0.0  0.0  0:00.03 sshd: valorani [priv]
 6780 root       19   0 99204  3936  3124 S  0.0  0.0  0:00.02 sshd: yildirim [priv]
F1Help  F2Setup F3SearchF4InvertF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```



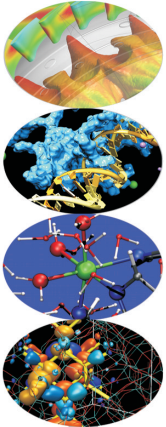
SCAI
SuperComputing Applications and Innovation



CALCOLO PARALLELO con OpenMP
Modulo 2

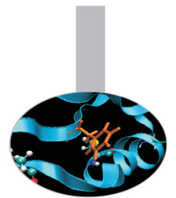
Claudia Truini Luca Ferraro Vittorio Ruggiero

CINECA Roma - SCAI Department



SCAI
SuperComputing Applications and Innovation

Outline



Concetti di base

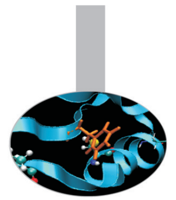
Parallelizzare un calcolo scalare

Parallelizzare un calcolo su una griglia

Parallelizzazione di un codice N-Body

Altre funzionalità

Il calcolo di π



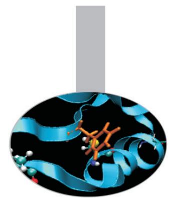
- Il valore di π può essere calcolato tramite

$$\int_0^1 \frac{4}{1+x^2}\mathrm{dx} = 4 \arctan \mathrm{x}\Big|_0^1 = \pi$$

- E approssimato con il metodo di Riemann dalla serie

$$\sum_{i=1}^{N} \frac{4h}{1+\left[(i-\frac{1}{2})h\right]^2} \quad \text{con} \quad h = \frac{1}{N}$$

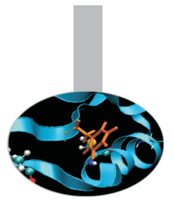
Paralleliziamo il calcolo di π



Fortran

```
program pigreco
  implicit none

  integer, parameter :: li = selected_int_kind(18)
  integer(li) :: i
  integer(li), parameter :: n=1e8
  real(kind(1.d0)) :: dx, sum, x, f, pi

  print *, 'number of intervals: ', n
  sum=0.d0
  dx=1.d0/n
  do i=1,n
    x=dx*(i-0.5d0)
    f=4.d0/(1.d0+x*x)
    sum=sum+f
  end do
  pi=dx*sum
  print '(a13,2x,f30.25)',' Computed PI =', pi
  ...
end program
```

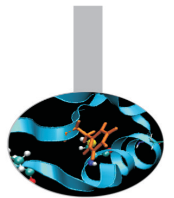
Paralleliziamo il calcolo di π



C

```
#include <stdio.h>
#define N 100000000

int main(int argc, char **argv)
{
  long int i, n = N;
  double x, dx, f, sum, pi;

  printf("number of intervals: %ld\n", n);
  sum = 0.0;
  dx = 1.0 / (double) n;
  for (i = 1; i <= n; i++) {
    x = dx * ((double) (i - 0.5));
    f = 4.0 / (1.0 + x*x);
    sum = sum + f;
  }
  pi = dx*sum;
  printf("Computed PI %.24f\n", pi);
  ...
  return 0;
}
```

Parallelizziamo il calcolo di π



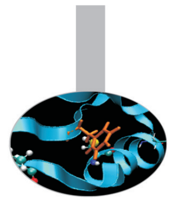
Fortran

```
!$omp parallel do
  do i=1,n
    x=dx*(i-0.5d0)
    f=4.d0/(1.d0+x*x)

    sum=sum+f

  end do
!$omp end parallel do
  pi=dx*sum
```

C

```
#pragma omp parallel for
  for (i = 1; i <= n; i++) {
    x = dx * ((double) (i - 0.5));
    f = 4.0 / (1.0 + x*x);

    sum = sum + f;
  }
  pi = dx*sum;
```

Controllo del *Data-Sharing*

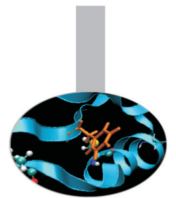


- Compiliamo e testiamo il programma
- Variando il numero di *thread*
- Otteniamo risultati differenti ed errati

- Per *default*, tutte le variabili sono *shared*
- Tutti i *thread* modificano **`x`** ed **`f`** senza coordinarsi
- “Gareggiano” tra loro, è una *data race*

- È evidente che i valori di **`x`** ed **`f`** sono specifici ad ogni iterazione
- Non è necessario condividerle
- La clausola **`private`** rende gli elementi della lista “locali” ai *thread*
  - Ogni *thread* accede ad una copia privata
  - In ingresso ed uscita dal costrutto le variabili private assumono valori indefiniti

Parallelizziamo il calcolo di π



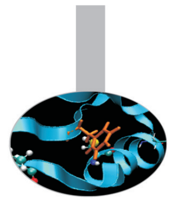
Fortran

```
!$omp parallel do private(x,f)
  do i=1,n
    x=dx*(i-0.5d0)
    f=4.d0/(1.d0+x*x)

    sum=sum+f

  end do
!$omp end parallel do
  pi=dx*sum
```

C

```
#pragma omp parallel for private(x,f)
  for (i = 1; i <= n; i++) {
    x = dx * ((double) (i - 0.5));
    f = 4.0 / (1.0 + x*x);

    sum = sum + f;
  }
  pi = dx*sum;
```

Sezioni critiche

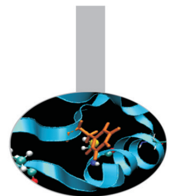


- Compiliamo e testiamo il programma
- Variando il numero di *thread*
- Ed otteniamo ancora risultati differenti ed errati

- C'è un'altra *data race*
- Tutti i *thread* leggono e modificano **`sum`** senza coordinarsi
- Ma è necessario che **`sum`** sia *shared* per ottenere un risultato dopo il *loop* parallelo

- È necessario che i *thread* aggiornino il contenuto di **`sum`** uno per volta
- Il costrutto **`critical`** è un modo di ottenerlo

Parallelizziamo il calcolo di π



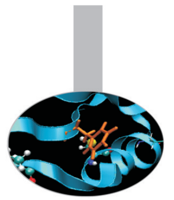
Fortran

```
!$omp parallel do private(x,f)
  do i=1,n
    x=dx*(i-0.5d0)
    f=4.d0/(1.d0+x*x)
!$omp critical
    sum=sum+f
!$omp end critical
  end do
!$omp end parallel do
  pi=dx*sum
```

C

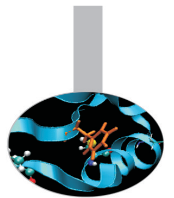
```
#pragma omp parallel for private(x,f)
  for (i = 1; i <= n; i++) {
    x = dx * ((double) (i - 0.5));
    f = 4.0 / (1.0 + x*x);
#pragma omp critical
    sum = sum + f;
  }
  pi = dx*sum;
```

Prestazioni



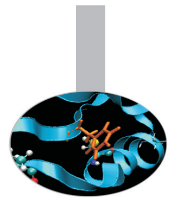
- Compiliamo e testiamo il programma
- Variando il numero di *thread*
- E misurando il tempo impiegato
  - con il comando **`time`**
  - o con la funzione **`omp_get_wtime()`** (che riporta un valore di tipo **`double`**)
- I risultati ora sono corretti, ma il tempo impiegato è enorme
- Tutti i *thread*, ad ogni iterazione, sono in contesa per l'accesso alla sezione critica
- Deve esserci un modo più efficiente

Operazioni collettive: `reduction`



- **`reduction(op:list)`**
- Per ogni variabile nella lista, genera copie locali ai *thread*
- La copia locale delle variabili contenute in **`list`** è inizializzata in modo opportuno in funzione dell'operatore **`op`** scelto (ad esempio, se **`op`** è + le variabili locali sono inizializzate a 0)
- Tutti gli aggiornamenti sono eseguiti sulla copia locale
- E solo alla fine del costrutto viene aggiornata la variabile originale, con una sezione critica generata automaticamente
- Le operazioni **`op`** valide in una **`reduction`** sono:
  - C: `+, *, -, &, |, ^, &&, ||, max, min`
  - Fortran: `+, *, -, .and., .or., .eqv., .neqv., max, min, iand, ior, ieor`

Parallelizziamo il calcolo di π



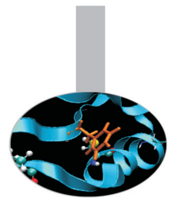
Fortran

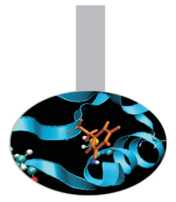
```
!$omp parallel do private(x,f) reduction(+:sum)
  do i=1,n
    x=dx*(i-0.5d0)
    f=4.d0/(1.d0+x*x)

    sum=sum+f

  end do
!$omp end parallel do
  pi=dx*sum
```

C

```
#pragma omp parallel for private(x,f) reduction(+:sum)
  for (i = 1; i <= n; i++) {
    x = dx * ((double) (i - 0.5));
    f = 4.0 / (1.0 + x*x);

    sum = sum + f;
  }
  pi = dx*sum;
```

CINECA SCAI
SuperComputing Applications and Innovation



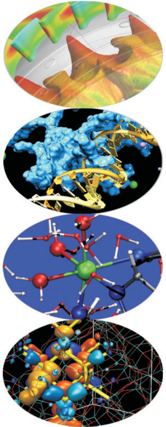


CALCOLO PARALLELO con OpenMP

Modulo 3

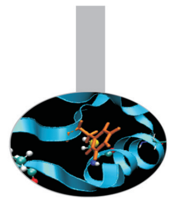
Claudia Truini Luca Ferraro Vittorio Ruggiero

CINECA Roma - SCAI Department



CINECA SCAI
SuperComputing Applications and Innovation

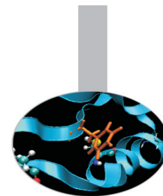
Outline



- Concetti di base
- Parallelizzare un calcolo scalare
- Parallelizzare un calcolo su una griglia
- Parallelizzazione di un codice N-Body
- Altre funzionalità

SCAI SuperComputing Applications and Innovation

L'equazione di Laplace

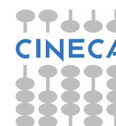


- Formula discreta su griglia 2D con condizioni al contorno di Dirichelet

$$\begin{cases} f(x_{i+1,j}) + f(x_{i-1,j}) - 2f(x_{i,j}) + \\ f(x_{i,j+1}) + f(x_{i,j-1}) - 2f(x_{i,j}) = 0 & \forall x_{i,j} \in (a,b)^2 \\ f(x) = \alpha(x) & \forall x_{i,j} \in \partial[a,b]^2 \end{cases}$$

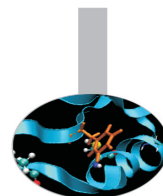
- Soluzione iterativa con il metodo di Jacobi

$$\begin{cases} f_{n+1}(x_{i,j}) = \frac{1}{4}[& f_n(x_{i+1,j}) + f_n(x_{i-1,j}) + \\ & f_n(x_{i,j+1}) + f_n(x_{i,j-1}) \quad] \quad \forall n > 0 \\ f_0(x_{i,j}) = 0 & \forall x_{i,j} \in (a,b)^2 \\ f_n(x) = \alpha(x) & \forall x \in \partial[a,b]^2, \qquad \forall n \geq 0 \end{cases}$$



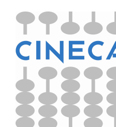
SCAI SuperComputing Applications and Innovation

Paralleliziamo il metodo di Jacobi

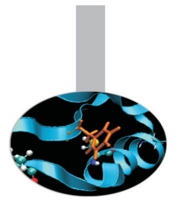


Fortran

```
program laplace
  integer, parameter :: dp=kind(1.d0)
  integer :: n, maxIter, i, j, iter = 0
  real (dp), dimension(:,:), pointer :: T, Tnew, Tmp=>null()
  real (dp) :: tol, var = 1.d0, top = 100.d0
  write( *,* ) 'Enter mesh size, max iter and tol:'
  read( *,* )  n, maxIter, tol
  allocate (T(0:n+1,0:n+1), Tnew(0:n+1,0:n+1))
  T(0:n,0:n) = 0.d0;
  T(n+1,1:n) = (/ (i, i=1,n) /) * (top / (n+1))
  T(1:n,n+1) = (/ (i, i=1,n) /) * (top / (n+1))
  Tnew = T
  do while (var > tol .and. iter <= maxIter)
    iter = iter + 1; var = 0.d0
    do j=1, n
      do i=1, n
        Tnew(i,j)=0.25d0*(T(i-1,j)+T(i+1,j)+T(i,j-1)+T(i,j+1))
        var =  max(var,abs( Tnew(i,j) - T(i,j) ))
      end do
    end do
    Tmp =>T; T =>Tnew; Tnew => Tmp;
    if (mod(iter,100)==0) write( *,* ) ' iter, var:',iter,var
  end do
  ...
```



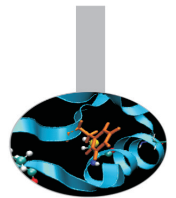
Paralleliziamo il metodo di Jacobi



C

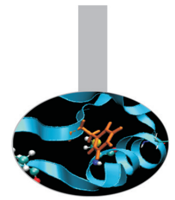
```
int main() {
   double *T, *Tnew, *Tmp;
   double tol, var=DBL_MAX, top=100.0;
   unsigned n, stride, maxIter, i, j, iter = 0;
   int itemsread;
   printf("Enter mesh size, max iterations and tolerance: ");
   itemsread = scanf("%u ,%u ,%lf", &n, &maxIter, &tol);
   stride = n+2;
   T = calloc(stride*stride, sizeof( *T ));
   Tnew = calloc(stride*stride, sizeof( *T ));
   setBoundaryConditions(T, Tnew, stride, top);
   while(var > tol && iter <= maxIter) {
      ++iter; var = 0.0;
      for (i=1; i<=n; ++i)
       for (j=1; j<=n; ++j) {
        Tnew[i*stride+j] = 0.25*(T[(i-1)*stride+j]+
         T[(i+1)*stride+j]+T[i*stride+(j-1)]+T[i*stride+(j+1)]);
        var = MAX(var, ABS(Tnew[i*stride+j] - T[i*stride+j]));
       }
      Tmp=T; T=Tnew; Tnew=Tmp;
      if (iter%100 == 0)
         printf("iter: %8u, var=%12.4lE\n",iter,var);
   }
   ...
```

L'equazione di Laplace: l'algoritmo



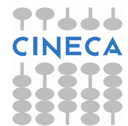
- L'array **T** rappresenta la funzione $f(x, y)$ sui punti della griglia
  - **T** è sovradimensionato in modo da ospitare le condizioni al contorno
  - l'algoritmo calcola solo sui punti interni dell'array
- Ad ogni iterazione, nei punti della griglia si calcola
  - la nuova soluzione **Tnew(i,j) = 0.25*(T(i-1,j)+T(i+1,j)+T(i,j-1)+T(i,j+1))**
  - la differenza (**var**) tra **Tnew** e la soluzione precedente **T**
- Il procedimento iterativo termina quando si raggiunge la convergenza entro la tolleranza oppure si supera il numero massimo di iterazioni (**maxiter**)
- Esercizio: parallelizzare il codice seriale

Parallelizzazione incrementale: Fortran

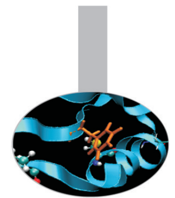


Fortran

```
  do while (var > tol .and. iter <= maxIter)
    iter = iter + 1;
    var = 0.d0
!$omp parallel do reduction(max:var)
    do j=1, n
      do i=1, n
        Tnew(i,j)=0.25d0*(T(i-1,j)+T(i+1,j)+T(i,j-1)+T(i,j+1))
        var =  max(var,abs( Tnew(i,j) - T(i,j) ))
      end do
    end do
!$omp end parallel do
    Tmp =>T; T =>Tnew; Tnew => Tmp;
    if (mod(iter,100)==0) write( *,* ) ' iter, var:',iter,var
  end do
```

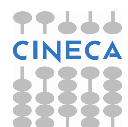


Parallelizzazione incrementale: C

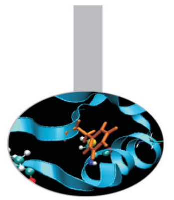


C

```
while(var > tol && iter <= maxIter) {
  ++iter;
  var = 0.0;
#pragma omp parallel
 {
   double pvar = 0.0;
#pragma omp for private(j)
  for (i=1; i<=n; ++i)
    for (j=1; j<=n; ++j) {
      Tnew[i*stride+j] = 0.25*(T[(i-1)*stride+j]+
        T[(i+1)*stride+j]+T[i*stride+(j-1)]+T[i*stride+(j+1)]);
      pvar = MAX(pvar, ABS(Tnew[i*stride+j]-T[i*stride+j]));
       }
#pragma omp critical
    if( pvar > var) var = pvar;
 }
     Tmp=T; T=Tnew; Tnew=Tmp;
     if (iter%100 == 0)
       printf("iter: %8u, var=%12.4lE\n",iter,var);
}
```



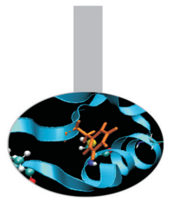
Ridurre l'*overhead*



- Ad ogni iterazione del **`while`** viene aperta e chiusa una regione parallela
- Ed aprire e chiudere una sezione parallela ha un costo
  - I *thread* devono essere creati all'inizio e distrutti alla fine
  - Lo stesso per le strutture dati necessarie al *runtime* per gestirli e distribuire il lavoro
- Molte implementazioni riducono tali costi con una serie di trucchi
- Ma in generale è meglio che le sezioni parallele siano più ampie possibili
- Il che richiede maggiore attenzione nel coordinare i *thread*
- E nuovi costrutti...
- Procedendo per passi

SCAI SuperComputing Applications and Innovation

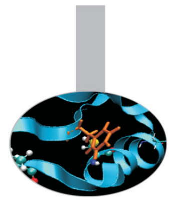
Parallelizzazione incrementale: Fortran



Fortran

```
!$omp parallel
  do while (var > tol .and. iter <= maxIter)
    iter = iter + 1;
    var = 0.d0
!$omp do reduction(max:var)
    do j=1, n
      do i=1, n
        Tnew(i,j)=0.25d0*(T(i-1,j)+T(i+1,j)+T(i,j-1)+T(i,j+1))
        var =  max(var,abs( Tnew(i,j) - T(i,j) ))
      end do
    end do
!$omp end do
    Tmp =>T; T =>Tnew; Tnew => Tmp;
    if (mod(iter,100)==0) write( *,* ) ' iter, var:',iter,var
  end do
!$omp end parallel
```

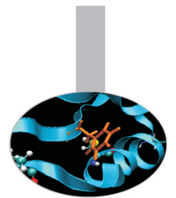
Parallelizzazione incrementale: C



C

```
#pragma omp parallel
 {
while(var > tol && iter <= maxIter) {
  ++iter;
  var = 0.0;

   double pvar = 0.0;
#pragma omp for private(j)
  for (i=1; i<=n; ++i)
    for (j=1; j<=n; ++j) {
      Tnew[i*stride+j] = 0.25*(T[(i-1)*stride+j]+
        T[(i+1)*stride+j]+T[i*stride+(j-1)]+T[i*stride+(j+1)]);
      pvar = MAX(pvar , ABS(Tnew[i*stride+j]-T[i*stride+j]));
       }
#pragma omp critical
      if( pvar > var) var = pvar;

     Tmp=T; T=Tnew; Tnew=Tmp;
     if (iter%100 == 0)
       printf("iter: %8u, var=%12.4lE\n",iter,var);
}
 }
```

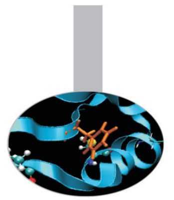
Il costrutto **`single`**



- Problemi:
  - Ora c'è una *data race* sull'azzeramento di **`var`** e l'incremento di **`iter`**
  - Lo scambio dei puntatori deve essere fatto da un solo *thread*
  - Di stampa delle iterazioni svolte ne basta una per tutti i *thread*

- **`single`** è un costrutto di *worksharing*
  - Il primo *thread* che lo raggiunge esegue il blocco associato
  - Gli altri lo saltano

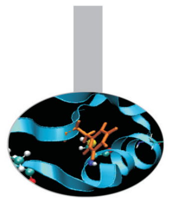
Parallelizzazione incrementale: Fortran



Fortran

```
!$omp parallel
  do while (var > tol .and. iter <= maxIter)
!$omp single
    iter = iter + 1;
    var = 0.d0
!$omp end single
!$omp do reduction(max:var)
    do j=1, n
      do i=1, n
        Tnew(i,j)=0.25d0*(T(i-1,j)+T(i+1,j)+T(i,j-1)+T(i,j+1))
        var =  max(var,abs( Tnew(i,j) - T(i,j) ))
      end do
    end do
!$omp end do
!$omp single
    Tmp =>T; T =>Tnew; Tnew => Tmp;
    if (mod(iter,100)==0) write( *,* ) ' iter, var:',iter,var
!$omp end single
  end do
!$omp end parallel
```

Parallelizzazione incrementale: C

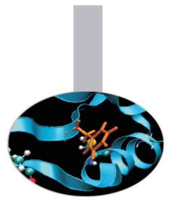


C

```
#pragma omp parallel
 {
while(var > tol && iter <= maxIter) {
#pragma omp single
  {
  ++iter;
  var = 0.0;
  }
  double pvar = 0.0;
#pragma omp for private(j)
      ...
#pragma omp critical
      if( pvar > var) var = pvar;
#pragma omp single
  {
     Tmp=T; T=Tnew; Tnew=Tmp;
     if (iter%100 == 0)
       printf("iter: %8u, var=%12.4lE\n",iter,var);
  }
}
 }
```

SCAI SuperComputing Applications and Innovation

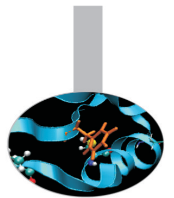
Barriere



- Problemi:
  - Le variabili **iter** e **var** non devono essere aggiornate finché tutti i thread non hanno controllato la condizione nel **while**
  - L'output non deve essere eseguito finché tutti i thread non hanno aggiornato **var**
- Barriere esplicite
  - Realizzate con il costrutto **barrier** che non ha un blocco di codice associato, garantiscono che TUTTO il codice che la precede è stato eseguito
  - un *thread* giunto ad una barriera deve arrestarsi finché tutti i *thread* nel *team* non la raggiungono
- Barriere implicite
  - Alla fine dei costrutti **do**/**for** e **single**
  - Ma non del costrutto **critical**!
- Alcune volte non sono necessarie, e causerebbero rallentamenti
  - Si possono rimuovere con la clausola **nowait**

SCAI SuperComputing Applications and Innovation

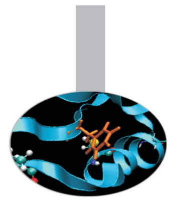
Parallelizzazione incrementale: Fortran



Fortran

```
!$omp parallel
  do while (var > tol .and. iter <= maxIter)
!$omp barrier
!$omp single
    iter = iter + 1;
    var = 0.d0
!$omp end single
!$omp do reduction(max:var)
    do j=1, n
      do i=1, n
        Tnew(i,j)=0.25d0*(T(i-1,j)+T(i+1,j)+T(i,j-1)+T(i,j+1))
        var =  max(var,abs( Tnew(i,j) - T(i,j) ))
      end do
    end do
!$omp end do
!$omp single
    Tmp =>T; T =>Tnew; Tnew => Tmp;
    if (mod(iter,100)==0) write( *,* ) ' iter, var:',iter,var
!$omp end single nowait
  end do
!$omp end parallel
```

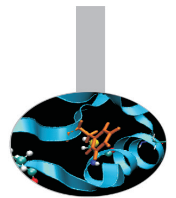
Parallelizzazione incrementale: C



C

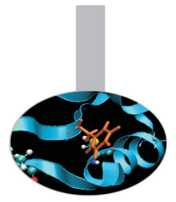
```
#pragma omp parallel
 {
while(var > tol && iter <= maxIter) {
#pragma omp barrier
#pragma omp single
  {
  ++iter;
  var = 0.0;
  }
  double pvar = 0.0;
#pragma omp for nowait private(j)
      ...
#pragma omp critical
      if( pvar > var) var = pvar;
#pragma omp barrier
#pragma omp single nowait
  {
     Tmp=T; T=Tnew; Tnew=Tmp;
     if (iter%100 == 0)
       printf("iter: %8u, var=%12.4lE\n",iter,var);
  }
}
 }
```

Il costrutto **workshare**



- Il blocco di istruzioni racchiuso nel costrutto **workshare** viene diviso in unità di lavoro che sono poi assegnate ai *thread*
- È supportato solo in Fortran allo scopo di parallelizzare l'*array syntax*
- Il compilatore provvede a tradurre l'*array syntax* in *loop*
- Svantaggi:
  - Molte implementazioni sono inefficenti
  - Le possibilità di controllo sono limitate
- Nel caso la *performance* del costrutto **workshare** non sia soddisfacente conviene tradurre a mano l'*array syntax* nei cari, vecchi *loop*

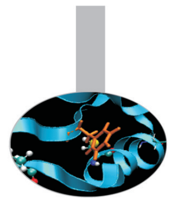
Equazione di Laplace, l'ultimo sforzo!



Fortran

```
!$omp parallel workshare
  T(0:n,0:n) = 0.d0;
  T(n+1,1:n) = (/ (i, i=1,n) /) * (top / (n+1))
  T(1:n,n+1) = (/ (i, i=1,n) /) * (top / (n+1))
!$omp end parallel workshare

  Tnew = T

!$omp parallel
  do while (var > tol .and. iter <= maxIter)
!$omp barrier
!$omp single
    iter = iter + 1;
    var = 0.d0
!$omp end single
    ...
  end do
!$omp end parallel
```

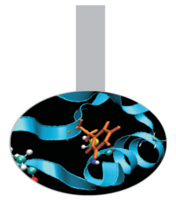
Equazione di Laplace, l'ultimo sforzo!



Fortran

```
!$omp parallel
!$omp workshare
  T(0:n,0:n) = 0.d0;
  T(n+1,1:n) = (/ (i, i=1,n) /) * (top / (n+1))
  T(1:n,n+1) = (/ (i, i=1,n) /) * (top / (n+1))
!$omp end workshare
!$omp single
  Tnew = T
!$omp end single nowait

  do while (var > tol .and. iter <= maxIter)
!$omp barrier
!$omp single
    iter = iter + 1;
    var = 0.d0
!$omp end single
    ...
  end do
!$omp end parallel
```

SuperComputing Applications and Innovation

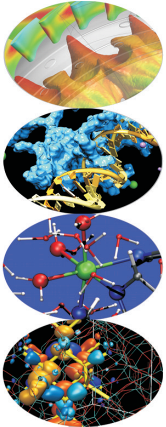


CALCOLO PARALLELO con OpenMP

Modulo 4

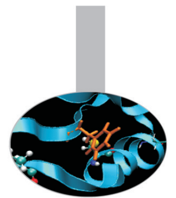
Claudia Truini Luca Ferraro Vittorio Ruggiero

CINECA Roma - SCAI Department



SuperComputing Applications and Innovation

Outline



Concetti di base

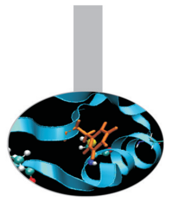
Parallelizzare un calcolo scalare

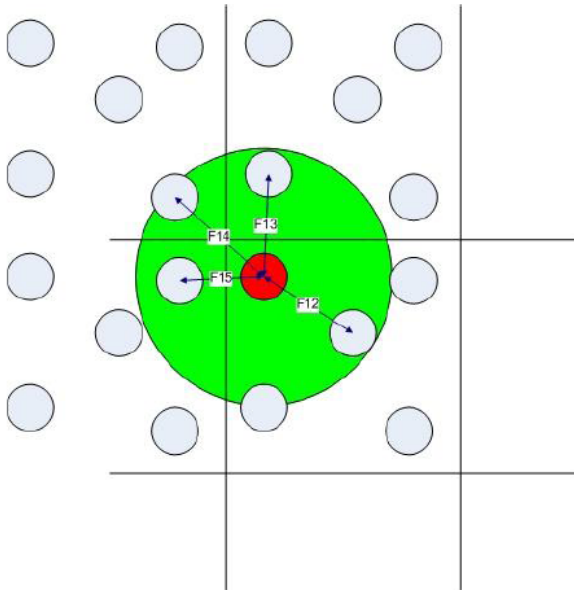
Parallelizzare un calcolo su una griglia

Parallelizzazione di un codice N-Body

Altre funzionalità

Calcolo del potenziale



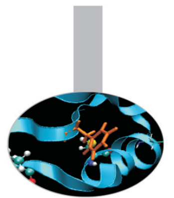


- N cariche puntiformi positive identiche
- Potenziale coulombiano:

$$V = \begin{cases} \frac{1}{r} & \text{se } r \leq cutoff \\ 0 & \text{se } r > cutoff \end{cases}$$

- Condizioni periodiche

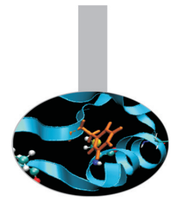
Calcolo delle forze e dell'energia



Fortran

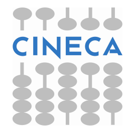
```
program nbody
   implicit none
   real(kind(1.d0)) :: pos(3,DIM), forces(3,DIM), f(3), ene
   real(kind(1.d0)) :: rij(3), d2, d, cut2=1000
   integer :: i, j, k
   read( *,* ) pos  !legge le posizioni dei pianeti
   forces = 0.d0
   ene = 0.d0
   do i = 1, DIM
      do j = i+1, DIM
         rij(:) = pos(:,i) - pos(:,j)
         d2 = 0.d0
         do k = 1, 3
            d2 = d2 + rij(k)**2
         end do
         if (d2 .le. cut2) then
            d = sqrt(d2)
            f(:) = - 1.d0 / d**3 * rij(:)
            forces(:,i) = forces(:,i) +  f(:)
            forces(:,j) = forces(:,j) -  f(:)
            ene = ene + (-1.d0/d)
         end if
      end do
   end do
   write ( *,fmt='(e20.10)' ) ene
   write ( *,fmt='(i5,1x,3e20.10)' ) (i,  forces(:, i), i =1, DIM)
end program nbody
```

Calcolo delle forze e dell'energia

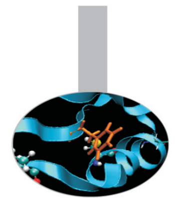


C

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
int main() {
   char fn[FILENAME_MAX];
   FILE *fin;
   double ( *pos )[3], ( *forces )[3];
   double rij[3], d, d2, d3, ene, cut2=1000;
   unsigned i, j, k, nbodies=DIM;
   /* read input file */
   ene = 0.0;
   for(i=0; i<nbodies; ++i)
      for(j=i+1; j<nbodies; ++j) {
         d2 = 0.0;
         for(k=0; k<3; ++k) {
            rij[k] = pos[i][k] - pos[j][k];
            d2 += rij[k]*rij[k];
         }
         if (d2 <= cut2) {
            d = sqrt(d2);
            d3 = d*d2;
            for(k=0; k<3; ++k) {
               double f = -rij[k]/d3;
               forces[i][k] += f;
               forces[j][k] -= f;
            }
            ene += -1.0/d;
         }
      }
      /* write energy and forces */
   return 0;
}
```



Test seriale



- Compiliamo il codice seriale:

  ifort -fast -DDIM=55000 Nbody.F90 -o nbodyf

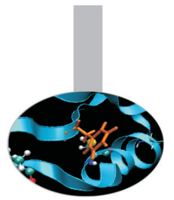
  icc -fast -DDIM=55000 Nbody.c -o nbodyc

- Eseguiamolo e salviamo l'output:

  nbodyf < positions.xyz.55000 > outf

  nbodyc < positions.xyz.55000 > outc

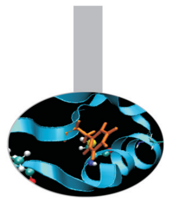
Parallelizzazione



Fortran

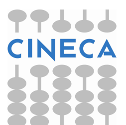
```
!$omp parallel do private(i,j,k,rij,d,d2,f) reduction(+:ene)
   do i = 1, DIM
      do j = i+1, DIM
         rij(:) = pos(:,i) - pos(:,j)
         d2 = 0.d0
         do k = 1, 3
            d2 = d2 + rij(k)**2
         end do
         if (d2 .le. cut2) then
            d = sqrt(d2)
            f(:) = - 1.d0 / d**3 * rij(:)
            do k=1,3
            forces(:,i) = forces(:,i) + f(:)

            forces(:,j) = forces(:,j) - f(:)


            ene = ene -1.d0/d
         end if
      end do
   end do
```

Parallelizzazione



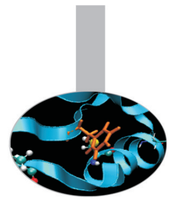
C

```
#pragma omp parallel for private(i,j,k,rij,d,d2,d3) \
reduction(+:ene)
   for(i=0; i<nbodies; ++i)
      for(j=i+1; j<nbodies; ++j) {
         d2 = 0.0;
         for(k=0; k<3; ++k) {
            rij[k] = pos[i][k] - pos[j][k];
            d2 += rij[k]*rij[k];
         }
         if (d2 <= cut2) {
            d = sqrt(d2);
            d3 = d*d2;
            for(k=0; k<3; ++k) {
               double f = -rij[k]/d3;

               forces[i][k] += f;

               forces[j][k] -= f;
            }
            ene += -1.0/d;
         }
      }
```



SCAI — SuperComputing Applications and Innovation

Race condition



Fortran

```
f(:) = - 1.d0 / d**3 * rij(:)
forces(:,i) = forces(:,i)+f(:)
forces(:,j) = forces(:,j)-f(:)
```

C

```
for(k=0; k<3; ++k) {
   double f = -rij[k]/d3;
   forces[i][k] += f;
   forces[j][k] -= f; }
```

- Esperimento ideale
  - La carica 1 interagisce con 2
  - la carica 2 interagisce con 3
  - Iterazione $i = 1$ al *thread* 0, iterazione $i = 2$ al *thread* 1

Thread 0

1. Legge le posizioni delle cariche 1 e 2
2. Calcola la forza tra la coppia 1 — 2 (**f_12**)
3. Legge **forces(:, 1) = 0** e **forces(:, 2) = 0**
4. Somma **f_12** a **forces(:, 1) = 0** e **forces(:, 2) = 0**
5. **forces(:, 1) = −forces(:, 2) = f_12**

Thread 1

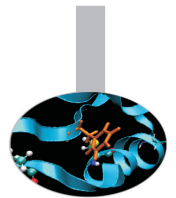
1. Legge le posizioni delle cariche 2 e 3
2. Calcola la forza tra la coppia 2 — 3 (**f_23**)
3. Legge **forces(:, 2) = 0** e **forces(:, 3) = 0**
4. Somma **f_23** a **forces(:, 2) = 0** e **forces(:, 3) = 0**
5. **forces(:, 2) = −forces(:, 3) = f_23**

In seriale invece:

forces(:, 2) = −f_12 + f_23

SCAI — SuperComputing Applications and Innovation

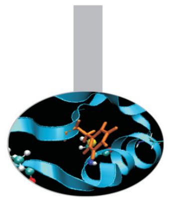
Parallelizzazione: reduction



Fortran

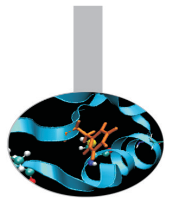
```
!$omp parallel do private(i,j,k,rij,d,d2,f) &
!$omp reduction(+:ene,forces)

   do i = 1, DIM
      do j = i+1, DIM
         rij(:) = pos(:,i) - pos(:,j)
         d2 = 0.d0
         do k = 1, 3
            d2 = d2 + rij(k)**2
         end do
         if (d2 .le. cut2) then
            d = sqrt(d2)
            f(:) = - 1.d0 / d**3 * rij(:)

            forces(:,i) = forces(:,i) +  f(:)
            forces(:,j) = forces(:,j) -  f(:)

            ene = ene + (-1.d0/d)
         end if
      end do
   end do
!$omp end parallel do
```

Il costrutto **atomic**



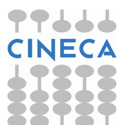
- **`reduction(+:forces)`** crea una copia dell'*array* per ogni *thread*
  - L'uso di memoria può essere elevato
  - Variabile di ambiente **`OMP_STACKSIZE`** per garantire uno *stack* sufficiente ad ogni *thread*
- Il costrutto **`atomic`** si applica solo a *statement* che aggiornano il valore di una variabile
  - Garantisce che nessun altro *thread* modifichi la variabile tra la lettura e la scrittura
- NON è un costrutto **`critical`**!
  - Solo la modifica è serializzata, e solo se due o più *thread* accedono allo stesso indirizzo di memoria
  - Può dare significativi vantaggi nel tempo di esecuzione rispetto ad un costrutto **`critical`**
- Le istruzioni permesse differiscono tra Fortran e C/C++
  - Consultare le specifiche dello standard

Parallelizzazione: **atomic**

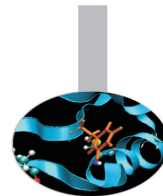


Fortran

```
!$omp parallel do private(i,j,k,rij,d,d2,f) reduction(+:ene)
   do i = 1, DIM
      do j = i+1, DIM
         rij(:) = pos(:,i) - pos(:,j)
         d2 = 0.d0
         do k = 1, 3
            d2 = d2 + rij(k)**2
         end do
         if (d2 .le. cut2) then
            d = sqrt(d2)
            f(:) = - 1.d0 / d**3 * rij(:)
            do k=1,3
!$omp atomic
               forces(k,i) = forces(k,i) + f(k)
!$omp atomic
               forces(k,j) = forces(k,j) - f(k)
            end do
            ene = ene -1.d0/d
         end if
      end do
   end do
```

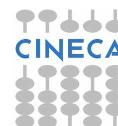


Parallelizzazione: **atomic**

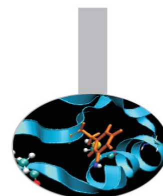


C

```
#pragma omp parallel for private(i,j,k,rij,d,d2,d3) \
reduction(+:ene)
    for(i=0; i<nbodies; ++i)
       for(j=i+1; j<nbodies; ++j) {
          d2 = 0.0;
          for(k=0; k<3; ++k) {
             rij[k] = pos[i][k] - pos[j][k];
             d2 += rij[k]*rij[k];
          }
          if (d2 <= cut2) {
             d = sqrt(d2);
             d3 = d*d2;
             for(k=0; k<3; ++k) {
                double f = -rij[k]/d3;
#pragma omp atomic
                forces[i][k] += f;
#pragma omp atomic
                forces[j][k] -= f;
             }
             ene += -1.0/d;
          }
       }
```



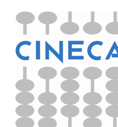
Parallelizzazione: riduzione esplicita



C

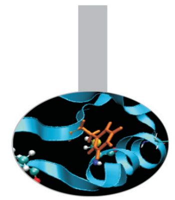
```
int tot_threads;
double  ( *gforces )[3];

#pragma omp parallel  private(i,j,k,rij,d,d2,d3)
{
  #pragma omp single
  { tot_threads = omp_get_num_threads();
    gforces = calloc(nbodies*tot_threads, sizeof( *gforces )); }
//...
```

- In C/C++ non è definita la *reduction* sugli *array*
- Perché in C/C++ gli *array* non sono tipi aritmetici
- Potremmo procedere come con Laplace, ma avremmo una regione **critical** contenente un *loop*
- Come evitarlo?
  - Allochiamo un vettore *shared* globale, puntato da **gforces**
  - Ed ogni *thread* ne userà una porzione distinta in base al proprio *rank*



SuperComputing Applications and Innovation

Parallelizzazione: riduzione esplicita

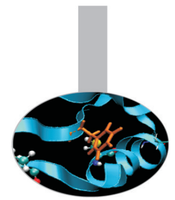


C

```
#pragma omp parallel private(i,j,k,rij,d,d2,d3)
{
   double ( *pforces )[3];

  #pragma omp single
  { tot_threads = omp_get_num_threads();
    gforces = calloc(nbodies*tot_threads, sizeof( *gforces )); }

   pforces = gforces + nbodies*omp_get_thread_num();

#pragma omp for reduction(+:ene)
   for(i=0; i<nbodies; ++i)
      for(j=i+1; j<nbodies; ++j) {
         d2 = 0.0;
            for(k=0; k<3; ++k) {
               rij[k] = pos[i][k] - pos[j][k];
               d2 += rij[k]*rij[k];
            }
            if (d2 <= cut2) {
               d = sqrt(d2);
               d3 = d*d2;
               for(k=0; k<3; ++k) {
                  double f = -rij[k]/d3;
                  pforces[i][k] += f;
                  pforces[j][k] -= f;
               }
               ene += -1.0/d;
            }
  //...
```

SuperComputing Applications and Innovation

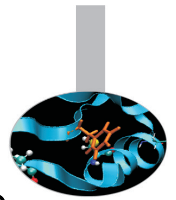
Parallelizzazione: riduzione esplicita

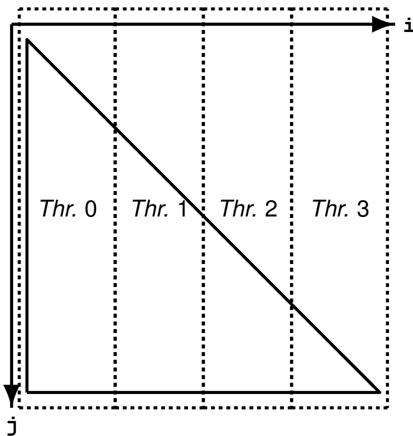


C

```
 //...

#pragma omp for
    for(i=0; i<nbodies; ++i)
       for(j=0; j<tot_threads; j++)
          for(k=0; k<3; ++k)
             forces[i][k] += gforces[i+j*nbodies][k];
}
```

Sbilanciamento tra i *thread*



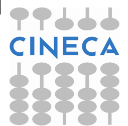


Loop nest triangolare: il numero di iterazioni in **j** è una funzione linearmente decrescente di **i**

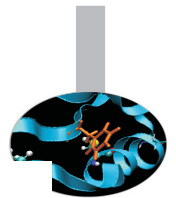
Il lavoro di ogni *thread* è proporzionale all'area assegnata del triangolo di iterazioni

Il *thread* 0 ha sempre più lavoro degli altri e determina il tempo di esecuzione

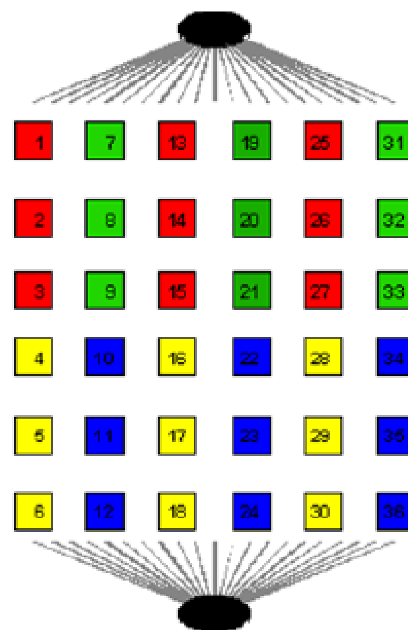
- Per *default* le iterazioni del *loop* parallelizzato vengono suddivise tra gli *N thread* in *N chunk* di pari dimensioni
- La clausola **schedule(*tipo*[,*chunk_size*])** consente di controllare la distribuzione del lavoro



static scheduling



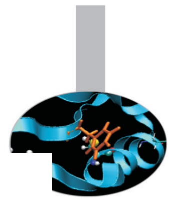
- Il numero di iterazioni del *loop* parallelo viene diviso in *chunk* di dimensione costante, assegnati in *round robin* ai *thread* in ordine di *thread_id* crescente
- Di *default* la dimensione del *chunk* è pari a $N_{iter}/N_{threads}$ (salvo arrotondamenti)



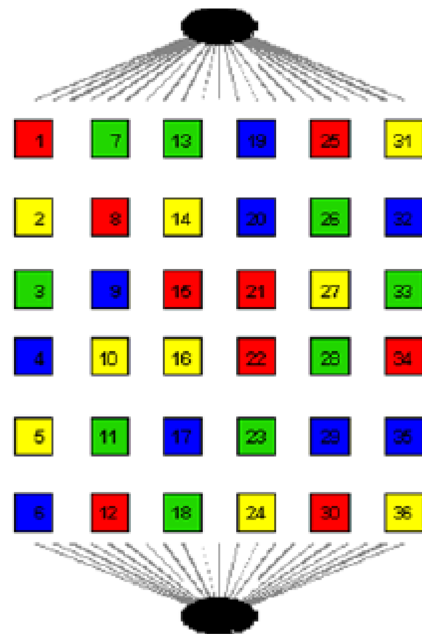
- Esempio:

```
!$omp parallel do &
!$omp schedule(static,3)
```

`dynamic` scheduling



- Il numero di iterazioni del *loop* parallelo viene diviso in *chunk* di dimensione costante ed ognuno è assegnato ad un *thread* non appena questo termina l'esecuzione del *chunk* precedente
- Di *default* la dimensione del *chunk* è pari a 1

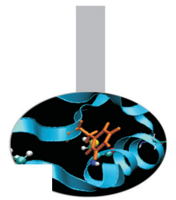


- Esempio:

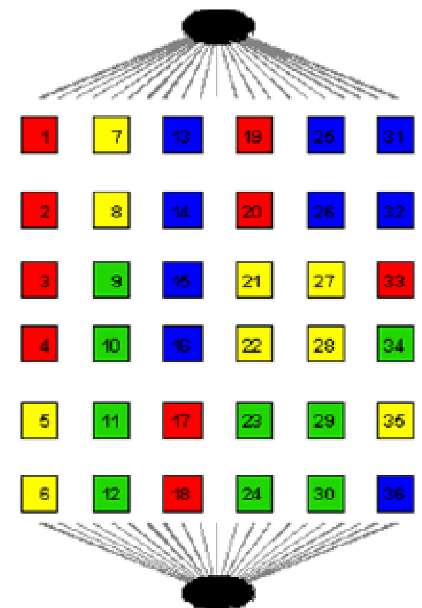
```
!$omp parallel do &
!$omp schedule(dynamic,1)
```

CINECA SCAI
SuperComputing Applications and Innovation

`guided` scheduling



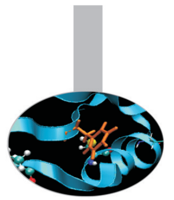
- Il numero di iterazioni è diviso in *chunk* di dimensione decrescente ed ognuno è assegnato ad un *thread* non appena questo termina l'esecuzione del *chunk* precedente
- Di *default* la dimensione minima del *chunk* è pari a 1



- Esempio:

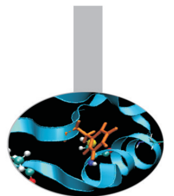
```
!$omp parallel do &
!$omp schedule(guided,1)
```

`runtime` scheduling



- Lo *schedule* e la dimensione dei *chunk* possono essere definite nel momento dell'esecuzione del programma attraverso la variabile d'ambiente **`OMP_SCHEDULE`**
- Esempio:
  ```
  !$omp parallel do &
  !$omp schedule(runtime)
  ```
- Lo scheduling potrà essere cambiato senza ricompilare il programma modificando la variabile d'ambiente **`OMP_SCHEDULE`**, ad esempio:
  ```
  setenv OMP_SCHEDULE "dynamic,50"
  ```
- Utile solo a fini sperimentali durante la parallelizzazione

Parallelizzazione: `runtime` scheduling

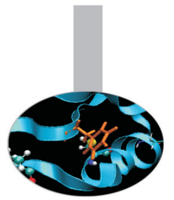


Fortran

```
!$omp parallel do private(i,j,k,rij,d,d2,f) &
!$omp reduction(+:ene,forces)   &
!$omp schedule(runtime)
   do i = 1, DIM
      do j = i+1, DIM
         rij(:) = pos(:,i) - pos(:,j)
         d2 = 0.d0
         do k = 1, 3
            d2 = d2 + rij(k)**2
         end do
         if (d2 .le. cut2) then
            d = sqrt(d2)
            f(:) = - 1.d0 / d**3 * rij(:)

            forces(:,i) = forces(:,i) +  f(:)
            forces(:,j) = forces(:,j) -  f(:)

            ene = ene + (-1.d0/d)
         end if
      end do
   end do
```

SuperComputing Applications and Innovation

Parallelizzazione: **runtime** scheduling

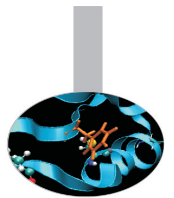


C

```
#pragma omp parallel private(i,j,k,rij,d,d2,d3)
{
  double ( *pforces )[3];

  #pragma omp single
  { tot_threads = omp_get_num_threads();
    gforces = calloc(nbodies*tot_threads, sizeof( *gforces )); }

  pforces = gforces + nbodies*omp_get_thread_num();
#pragma omp for reduction(+:ene) schedule(runtime)
   for(i=0; i<nbodies; ++i)
      for(j=i+1; j<nbodies; ++j) {
         d2 = 0.0;
            for(k=0; k<3; ++k)  {
               rij[k] = pos[i][k] - pos[j][k];
               d2 += rij[k]*rij[k];
            }
            if (d2 <= cut2) {
               d = sqrt(d2);
               d3 = d*d2;
               for(k=0; k<3; ++k) {
                  double f = -rij[k]/d3;
                  pforces[i][k] += f;
                  pforces[j][k] -= f;
               }
               ene += -1.0/d;
            }

 //...
```

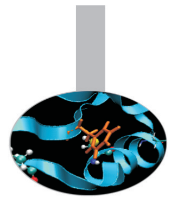
SuperComputing Applications and Innovation

Test parallelo



- Misurare i tempi delle tre versioni (seriale, parallela con **atomic** e parallela con **reduction**)
- Calcolare le performance (*speedup* ed efficienza) delle versioni parallele rispetto a quella seriale
- Come variano le prestazioni al variare del tipo di *scheduling/chunking*?
- I risultati delle differenti versioni sono uguali?
- Dovrebbero essere uguali?

SCAI
SuperComputing Applications and Innovation

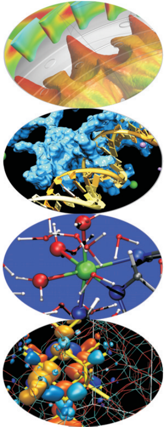


CALCOLO PARALLELO con OpenMP

Modulo 5

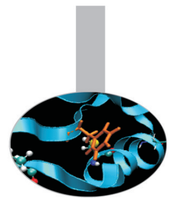
Claudia Truini Luca Ferraro Vittorio Ruggiero

CINECA Roma - SCAI Department



SCAI
SuperComputing Applications and Innovation

Outline



Concetti di base

Parallelizzare un calcolo scalare

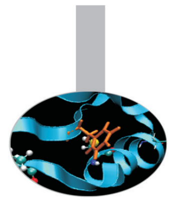
Parallelizzare un calcolo su una griglia

Parallelizzazione di un codice N-Body

Altre funzionalità
- Parallelismo irregolare
- Conclusioni

SCAI SuperComputing Applications and Innovation

Il costrutto `sections`



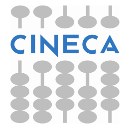
Fortran

```
!$omp sections
!$omp section
! codice
!$omp section
! altro codice
!...
!$omp end sections
```

C

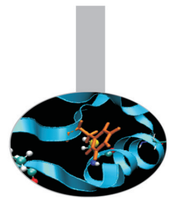
```
#pragma omp sections
{
  #pragma omp section
    // codice
  #pragma omp section
    // altro codice
  // ...
}
```

- È un costrutto di *worksharing* per distribuire blocchi di codice che possono essere eseguiti in parallelo
  - Ogni *thread* riceve una *section*
  - Quando un *thread* ha finito di eseguire una *section*, ne riceve un'altra
  - Se non ce ne sono altre da eseguire, aspetta che gli altri finiscano



SCAI SuperComputing Applications and Innovation

Il costrutto `master`



Fortran

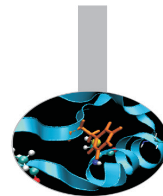
```
!$omp master
 write ( *,* ) 'Residual:', res
!$omp end master
```

C

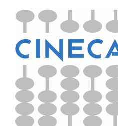
```
#pragma omp master
 printf("Residual: %lf\n", res);
```

- Solo il *Master Thread* esegue il blocco di codice associato
- Gli altri lo ignorano senza aspettare

Altre clausole per i dati

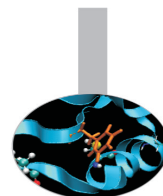


- **`firstprivate(`*`list`*`)`**
  - Come **`private(`*`list`*`)`**
  - Ma inizializza ogni copia col valore della variabile originale
- **`copyprivate(`*`list`*`)`**
  - Solo sul costrutto **`single`**
  - Le variabili elencate in *list* devono essere già *private*
  - Alla fine del costrutto copia i valori nelle copie locali al *thread* che lo ha eseguito nelle copie private di tutti gli altri *thread*
  - Di fatto è un'operazione di *broadcast* da un *thread* a tutto il *team*

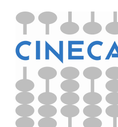


CINECA SCAI
SuperComputing Applications and Innovation

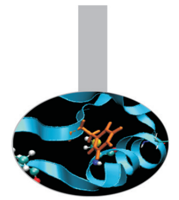
Orphaning



- Supponiamo di avere una funzione o *subroutine*
- Che vogliamo chiamare sia da regioni seriali che da regioni parallele del codice
- E che nelle regioni parallele le elaborazioni debbano essere distribuite tra i *thread*
- Non è necessario scrivere una versione parallela ed una seriale
- Possiamo avere un'unica versione ed inserirvi i costrutti di *worksharing* opportuni
- Un costrutto di *worksharing* non incluso lessicalmente in un costrutto **`parallel`**:
  - è detto orfano
  - viene eseguito serialmente se chiamato da una regione seriale
  - ma le variabili dichiarate *private* saranno comunque privatizzate

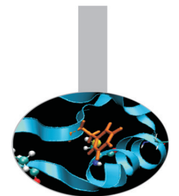


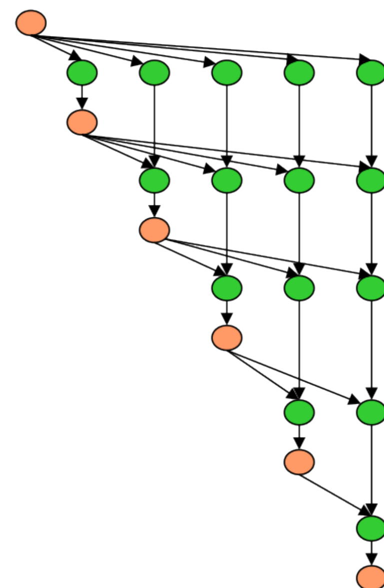
La direttiva `threadprivate`



- È una direttiva dichiarativa
- Si applica:
  - In Fortran, a blocchi **`common`** o a variabili di modulo
  - In C/C++ a variabili a *file-scope*, *namespace-scope* o *static*
- Deve seguire la dichiarazione della variabile, nella stessa unità di programma
- Permette di creare una copia privata per ciascun *thread* di variabili globali al programma
- Fondamentale per rendere *thread safe* funzioni di librerie
- E per parallelizzare codici suddivisi in più file sorgente che accedono a variabili globali
- È possibile inizializzarle con i valori della copia del *Master Thread* tramite la clausola **`copyin`**

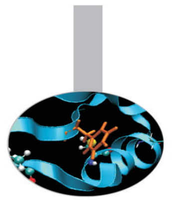
Decomposizione LU





- Un caso molto comune in algoritmi per l'algebra lineare
- Le frecce indicano dipendenze nell'ordine di esecuzione
- Il grado di parallelismo varia nel corso del calcolo
- Una gestione rigida riduce l'efficienza di un fattore 2

Un caso comune



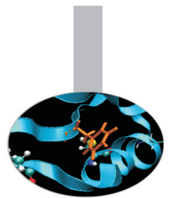
Fortran	C

```fortran
do while(associated(p))
  call process(p)
  p => p%next
end do
```

```c
while(p) {
  process(p);
  p = p->next;
}
```

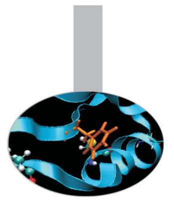
- È un *pointer chasing loop*
- Il numero di iterazioni non è noto all'inizio del loop
- Ogni iterazione dipende dalla precedente per la determinazione del nuovo **`p`**
- Non è parallelizzabile tramite un costrutto **`do`**/**`for`**

Parallelismo irregolare



- Macchine sempre più potenti
- Problemi sempre più grandi e complessi
- Algoritmi e strutture dati sempre più complicati
  - Algoritmi su grafi
  - Algoritmi di ricerca su sequenze di simboli
  - Ottimizzazioni combinatorie
  - Simulazioni *agent-based*
  - Discretizzazioni su griglie multiblocco complesse
  - *Embedded Grids*
  - *Adaptive Mesh Refinement*
  - Simulazioni di circuiti elettronici e chip complessi
  - Simulazioni di veicoli spaziali
- Sempre più applicazioni sono caratterizzate da parallelismo irregolare, che cambia durante l'esecuzione in maniera non facilmente prevedibile

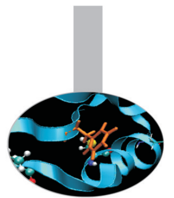
Il costrutto `task`



- Ogni volta che un *thread* incontra un costrutto **`task`**
  - genera un *task*
  - corrispondente all'esecuzione del blocco di codice associato
  - e dotato di un suo *data environment*

- Un *task* può essere eseguito immediatamente
- Oppure la sua esecuzione può essere differita, o affidata ad un altro *thread*
- Tutti i *thread* arrivati ad una barriera eseguono i *task* ancora in coda finché non sono tutti completati

Trattare una lista con i *task*



Fortran

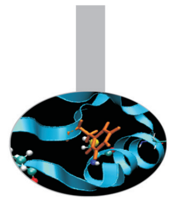
```
!$omp parallel
!$omp single
  do while(associated(p))
    !$omp task firstprivate(p)
      call process(p)
    !$omp end task
    p => p%next
  end do
!$omp end single
!$omp end parallel
```

Fortran

```
#pragma omp parallel
{
#pragma omp single
  while(p) {
    #pragma omp task firstprivate(p)
      process(p);
    p = p->next;
  }
}
```

- Un *thread* genera *task* per tutti
- Quando finisce di generarli arriva alla barriera implicita e comincia anche lui ad eseguirli

- Ma che valore assume **`p`** in ogni *task*?
  - Quello che aveva al momento della creazione del *task*



Load balancing con i *task*

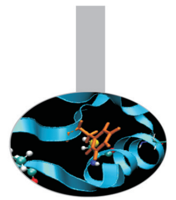
Fortran

```
!$omp parallel private(p)
  !$omp do
  do i=1, numlists
    p => listheads(i)%list
    do while(associated(p))
      !$omp task
        call process(p)
      !$omp end task
      p => p%next
    end do
  end do
  !$omp end do
!$omp end parallel
```

C

```
#pragma omp parallel private (p)
 {
   #pragma omp for
   for (i=0; i<num_lists; i++) {
     p = listheads[i];
     while(p) {
       #pragma omp task
         process(p);
       p=next(p);
     }
   }
 }
```

- Un numero dato di liste, di lunghezza variabile
- Assegnarne una per *thread* provocherebbe uno sbilanciamento
- Con i *task* è possibile bilanciare il carico di lavoro
- Per *default*, variabili *private* diventano *firstprivate* in un costrutto **`task`**



Strutture ad albero binario

Fortran

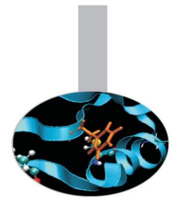
```
recursive subroutine preorder(p)
  type(node), pointer :: p
  call process(p%data)
  if (associated(p%left))
    !$omp task
      call preorder(p%left)
    !$omp end task
  end if
  if (associated(p%right))
    !$omp task
      call preorder(p%right)
    !$omp end task
  end if
end subroutine
```

C

```
void preorder(node *p) {
  process(p->data);
  if (p->left)
    #pragma omp task
      preorder(p->left);
  if (p->right)
    #pragma omp task
      preorder(p->right);
}
```

- Un *task* può generarne altri
- Anche in modo ricorsivo
- L'albero è attraversato in *preorder*: ogni nodo è elaborato prima di procedere ai figli
- I *dummy argument* diventano *firstprivate* in un costrutto **`task`** orfano

Albero binario: attraversamento *postorder*



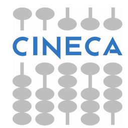
Fortran

```
recursive subroutine postorder(p)
  type(node), pointer :: p
  if (associated(p%left))
    !$omp task
      call postorder(p%left)
    !$omp end task
  end if
  if (associated(p%right))
    !$omp task
      call postorder(p%right)
    !$omp end task
  end if

  !$omp taskwait

  call process(p%data)
end subroutine
```

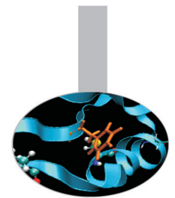
C

```
void postorder(node *p) {
  if (p->left)
    #pragma omp task
      postorder(p->left);
  if (p->right)
    #pragma omp task
      postorder(p->right);

  #pragma omp taskwait

  process(p->data);
}
```

- Nell'attraversamento in *postorder* ogni nodo è elaborato dopo l'elaborazione dei figli
- Il costrutto **`taskwait`**:
  - sospende l'esecuzione di un *task*
  - finché l'esecuzione dei *task* che lui ha generato non è stata completata

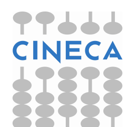


Come andare avanti

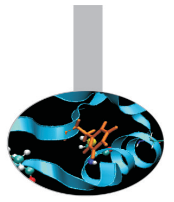


- Questo corso introduttivo consente di leggere e scrivere in OpenMP
- Ci sono altre funzionalità che non abbiamo descritto
- Ci sono modi più o meno efficienti di fare le cose
- Ci sono regole e restrizioni che dovrete conoscere in futuro

- Dove cercare approfondimenti?
  - Nel documento di specifiche di OpenMP, scaricabile dal sito **`http://www.openmp.org`**
  - Nei forum del sito **`http://www.openmp.org`**
  - Nel libro *Using OpenMP*, di B. Chapman, G. Jost, R. Van der Pas, MIT Press
  - O, sul parallelismo in generale, nel libro *Patterns for Parallel Programming*, di T. Mattson, B. Sanders, B. Massingill, Addison-Wesley



Credits



- Hanno collaborato alla realizzazione del materiale del corso:
  - Federico Massaioli
  - Alessandro Federico
  - Simone Meloni

CINECA SCAI SuperComputing Applications and Innovation