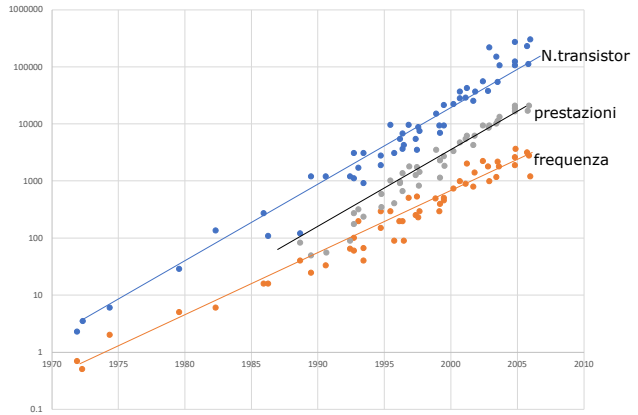
1

crescita delle prestazioni delle CPU

per circa 30 anni (1975-2005), le prestazioni delle CPU
sono cresciute semplicemente aumentando la frequenza del clock



Horowitz et al., 2010

2

qual e' stato il prezzo

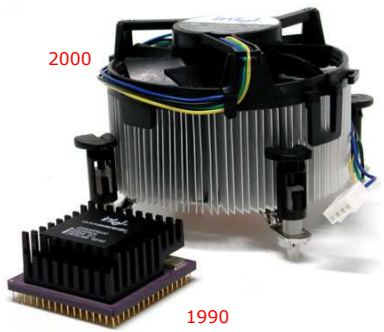
La potenza elettrica necessaria ad una CPU segue la legge

$$Power = C\,V^2 f$$

dove: C = capacitanza V = tensione elettrica f = frequenza

Inoltre, le dimensioni fisiche delle CPU non sono aumentate

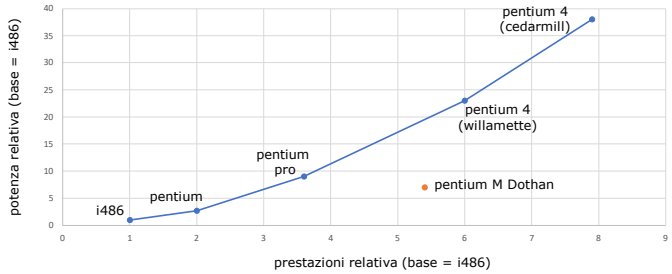
Il rapporto Potenza/cm²
(calore per cm²)
diventa ingestibile
(decine/centinaia di gradi/cm²)



3

uno studio interessante

Grochowski et al. (Intel), 2006



la potenza richiesta cresce esponenzialmente con le prestazioni

$$Power \sim perf^{1.75}$$

4

una soluzione

i processori per il segmento mobile hanno

- architetture piu' semplici
- frequenze minori
- minore consumo
- prestazioni paragonabili al segmento desktop
- miglior rapporto prestazioni/potenza

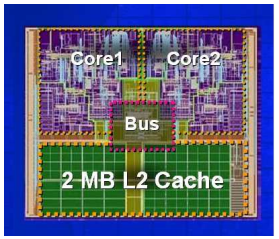
CPU	Anno	Frequenza	Potenza	Prestaz.	segmento
Pentium 4 Cedar Mill	2006	3.6 GHz	86 Watt	1764	Desktop
Pentium 4 Northwood	2004	3.4 GHz	84 Watt	1342	Desktop
Pentium M Dothan	2005	2.0 GHz	21 Watt	1429	Mobile

Per aumentare le prestazioni e ridurre la potenza richiesta,
e' possibile utilizzare 2 (o piu') CPU semplici al posto di 1 CPU piu' potente

le CPU multicore

Esempio: raddoppiare le prestazioni di una CPU_0

- CPU_1 : raddoppiare le prestazioni della singola unita' di calcolo ($perf_1 = 2\, perf_0$)
  da $Power \sim perf^{1.75}$ si ha $Power_1 = 2^{1.75}\, Power_0 = 3.34\, Power_0$
- CPU_2 : raddoppiare le unita' di calcolo all'interno della CPU
  si ha $Power_2 = 2\, Power_0$ e $perf_2 = 2\, perf_0$

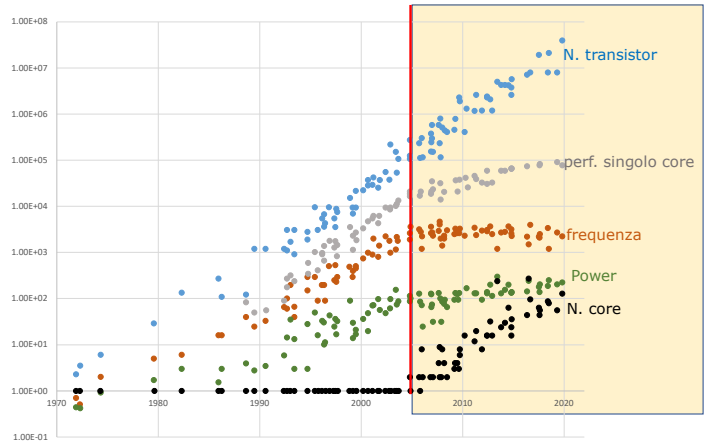


Intel Core Duo (Yonah, 2006)
ottenuta montando
due CPU Pentium M (Dothan)
sullo stesso chip

prima CPU intel dual core

evoluzione CPU 1975 - 2020

nel 2005 cambia l'idea di architettura delle CPU

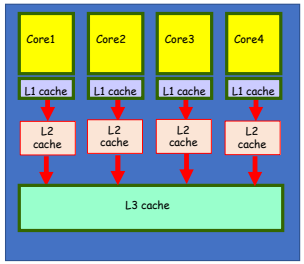


CPU Multicore

Una CPU multicore e' un insieme di unita' processanti autonome in un unico chip,
che condividono risorse (cache, bus, e memoria centrale)

I core possono avere cache
condivise e/o cache private

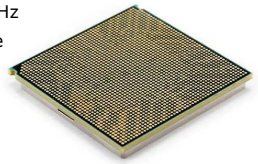
E' un esempio di sistema che implementa il modello a memoria condivisa



Core i7 (Ivy bridge, 2012)
- freq = 3.1 GHz
- cache L1 = 64 KB
- cache L2 = 256 KB
- cache L3 = 8 MB

Alcune moderne CPU multicore

IBM Power 9 (2017)
- 3.07 GHz
- 22 core

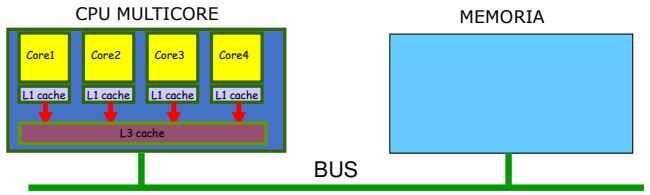


Intel Xeon Platinum 8280 (2019)
- 2.7 GHz
- 28 core

AMD EPYC 7742 (2019)
- 2.25 GHz
- 64 core

Fujitsu A64FX (2020)
- 2.2 GHz
- 48 core

sviluppo applicazioni per CPU multicore



Tutti i core accedono alla stessa memoria principale (spesso alla stessa cache)

modello a **memoria condivisa**

Lo strumento naturale per lo sviluppo di applicazioni
per CPU multicore sono
i threads

definizione di thread

Nei moderni s.o. un thread (anche detto processo leggero) e' una sequenza di istruzioni che puo' essere gestita autonomamente come un unico task dallo scheduler

Spesso un processo e' suddiviso in piu' thread

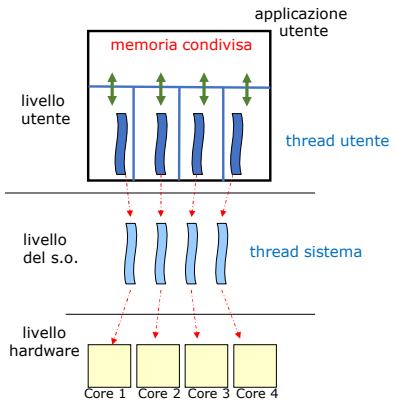
All'interno di un processo ogni thread possiede un proprio
- program counter
- insieme dei registri/cache
- lo stack

} esecuzione indipendentemente e gestione di variabili locali

mentre condivide con gli altri thread ad esso associati
- lo spazio di indirizzamento globale
- file aperti

} condivisione di variabili condivise attraverso cui scambia informazioni

modello di esecuzione dei thread



I threads utente, oltre ad un proprio spazio di indirizzamento, condividono tra loro un'area di memoria, attraverso cui si possono scambiare informazioni

I moderni S.O. associano un thread utente ad un thread di sistema secondo il modello 1-1 (Kernel level threads)

Il S.O. schedula threads indipendenti su differenti core della CPU

il modello di programmazione multithreading implementa naturalmente una forma di parallelismo a memoria condivisa

Threads POSIX (Pthreads)

Uno standard POSIX (IEEE 1003.1c) per la creazione e sincronizzazione dei threads

Definizione delle API

Threads conformi allo standard POSIX sono chiamati Pthreads

Lo standard POSIX stabilisce che registri dei processori, stack e signal mask sono individuali per ogni thread

Lo standard specifica come il sistema operativo dovrebbe gestire i segnali ai Pthtrads i specifica differenti metodi di cancellazione (asincrona, ritardata, …)

Permette di definire politiche di scheduling e priorita'

Alla base di numerose librerie di supporto per vari sistemi

creazione di un thread

```
#include <pthread.h>
int pthread_create  ( pthread_t  *tid,     const pthread_attr_t  *attr,
                                 void *(*func)(void*),     void *arg );
```

tid: puntatore all'identificativo del thread ritornato dalla funzione

attr: attributi del thread (priorita', dimensione stack, …) . A meno di esigenze particolari si usa NULL

func: funzione di tipo **void *** che costituisce il corpo del thread

arg : unico argomento di tipo **void *** passato a **func**

Attesa per la fine di un thread

```
#include <pthread.h>

int pthread_join( pthread_t  tid,  void ** status );
```

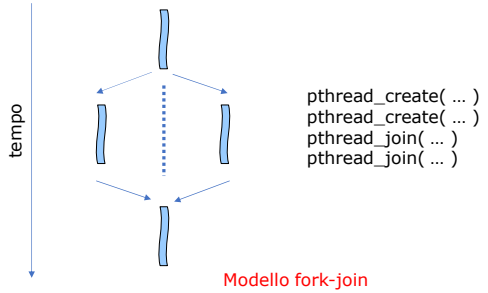
tid: identificativo del thread di cui attendere la fine

status: puntatore al valore di ritorno del thread. Di solito NULL

osservazione

la funzione pthread_create ritorna subito dopo aver creato un nuovo thread

due (o piu') chiamate successive alla funzione creano thread eseguiti contemporaneamente su distinti thread di sistema



Modello fork-join

esempio: 4 thread su CPU 4 core (1/3)

```
#include<pthread.h>
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<sys/types.h>
#include<unistd.h>
#include"c_timer.h"

int main( ){                // inizio main
          int nt;
          void ciao (int);

          for (nt = 1; nt <= 5; nt++) {
                printf(" esecuzione con %d thread\n",nt);
                ciao(nt);
          }

}  //fine main
```

esegue 5 volte la funzione ciao con un numero variabile di thread

esempio: 4 thread su CPU 4 core (2/3)

```
void ciao(int nt){
        void *thread(void *) ;
        int i, argmain[8];
        pthread_t  tid[8];
        double  t1, t2, tt;

        t1=get_cur_time();
        for(i=0; i<nt; i++){
                argmain[i] = i;
                pthread_create(&tid[i], NULL, thread, &argmain[i] );
        }

        for(i=0; i<nt; i++){
          pthread_join(tid[i], NULL);
        }
        tt = get_cur_time() -t1;

        printf("TEMPO TOTALE = %f  \n\n\n", tt);
}
```

dichiarazioni

creazione thread

attesa terminazione

stampa tempo totale

esempio: 4 thread su CPU 4 core (3/3)

```
void *thread (void *argmain){
        int *argthread, i, j;
        float a=1, sum=0 ;
        double t1, t2, tt;

        argthread = (int *)argmain;

        t1=get_cur_time();
        sum=0;
        for (i=0;  i<10000  ; i++){
        for (j=0;  j<100000  ; j++){
                sum=sum+a;
                a = -a;
        }}
        tt = get_cur-time()-t1;

        printf("thread %d, tempo = %f  \n", *argthread, tt);
}
```

dichiarazioni

conversione puntatore alla struct argomento

lavoro del thread

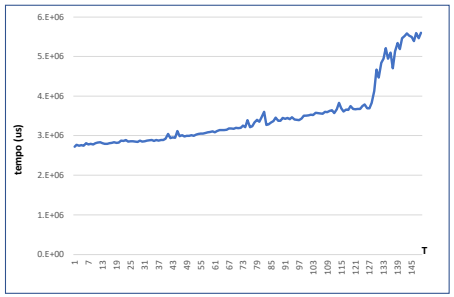
stampa tempo

esecuzione (Intel Core i5-4460S 2.9 Ghz)

```
esecuzione con 1 thread
thread 0, tempo = 2.698076
TEMPO TOTALE = 2.698168

esecuzione con 2 thread
thread 0, tempo = 2.763541
thread 1, tempo = 2.768131
TEMPO TOTALE = 2.771368

esecuzione con 3 thread
thread 2, tempo = 2.939556
thread 0, tempo = 2.956570
thread 1, tempo = 2.949608
TEMPO TOTALE = 2.961432

esecuzione con 4 thread
thread 0, tempo = 3.008063
thread 1, tempo = 3.026047
thread 2, tempo = 3.038589
thread 3, tempo = 3.165634
TEMPO TOTALE = 3.165736

esecuzione con 5 thread
thread 2, tempo = 3.582642
thread 3, tempo = 3.587059
thread 0, tempo = 3.748010
thread 1, tempo = 3.759567
thread 4, tempo = 4.099845
TEMPO TOTALE = 4.099981
```

Un esempio

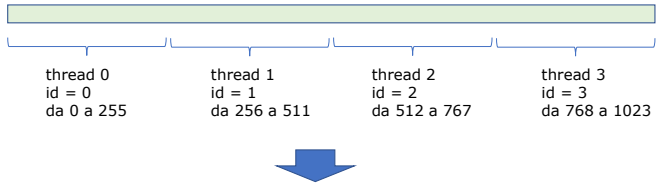
- il grafico in figura mostra, al variare del numero di thread, il tempo di esecuzione (in microsecondi) di un programma multithreading che esegue la somma di 10^9 somme f.p. in ciascun thread sulla CPU AMD Ryzen Threadripper 3990X (2020) con 64-Core, ciascuno dei quali capace di gestire 2 threads indipendenti.
- Al crescere del numero di thread T, si nota un moderato incremento del tempo, dovuto principalmente all'overhead di sistema per la loro generazione, fino a T=128. Superato tale valore i thread in eccesso vengono eseguiti in time sharing sulle singole unità di calcolo, con un significativo aumento del tempo di esecuzione dovuto alla concorrenza nell'uso simultaneo delle stesse risorse.



gestione dei thread

Una comune forma di parallelismo si ottiene
dividendo i dati del problema tra i thread (o processi)

Esempio: Somma di N=1024 elementi di un array con NT=4 thread
sottoproblemi di dimensione N/NT = 256



ogni thread ha bisogno di conoscere:

- numero totale di thread
- un identificativo che lo distingue dagli altri

inoltre

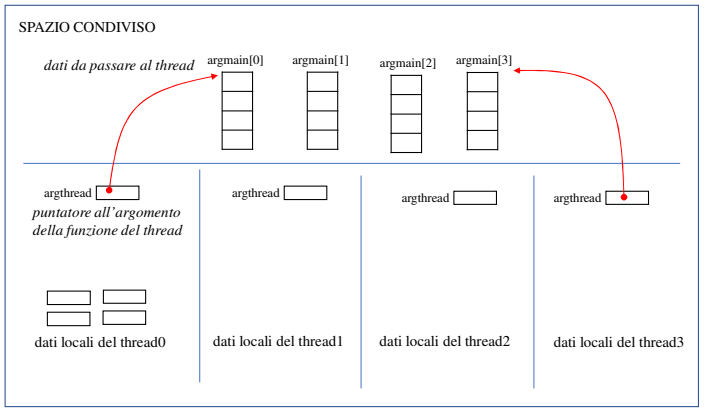
la funzione che contiene il codice del thread ha un solo argomento
(di tipo puntatore)

nel caso di piu' dati da passare alla funzione del thread

- si raggruppano tali dati in una struct
- si passa il puntatore alla struct

attraverso tale puntatore, il thread puo' accedere ai dati globali del processo

passaggio del puntatore a struct



problema

Il principale problema nell'uso dei threads e' il

rischio di corsa critica sui dati condivisi

sincronizzare gli accessi alla memoria condivisa

Uso di **semafori** e/o **altri strumenti**

Dal punto di vista dell'utente un **semaforo** e' una **variabile intera** a cui si puo' accedere solo mediante **due operazioni atomiche**, convenzionalmente chiamate

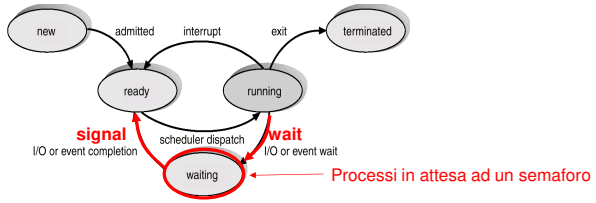
- wait(S)
- signal(S)

wait (S)

- se S>0 allora S=S-1 e il processo continua l'esecuzione
- altrimenti (S=0) il processo si blocca

signal(S)

- esegue S=S+1
- eventualmente rimanda in esecuzione un processo bloccato



Un semaforo permette al S.O. di rimuovere un processo dalla ready-list, e quindi dalla competizione per l'uso della CPU, risolvendo il problema della corsa critica

uso dei semafori (1)

Problema della corsa critica (accesso simultaneo su dati condivisi)

- N thread con dati condivisi
- si definisce una sezione critica attraverso un semaforo
- Si pone inizialmente un semaforo S = 1

I thread hanno una struttura del tipo

```
…
sezione non critica
wait(S)
sezione critica (accesso ai dati condivisi)
signal(S)
sezione non critica
…
```

i thread entrano uno alla volta all'interno della sezione critica

uso dei semafori (2)

sincronizzazione dei thread (definire un ordine nelle istruzioni di thread differenti)

Esempio: 2 thread

Si vuole eseguire prima l'istruzione A nel thread 0 e poi l'istruzione B nel thread 1 (es. produttore-consumatore)

si pone inizialmente un semaforo S = 0

thread 0

```
…
…
istruzione A
signal(S)
…
```

thread 1

```
…
wait(S)
istruzione B
…
…
```

thread Posix e semafori

i principali strumenti POSIX per implementare le sezioni critiche sono i mutex

Un mutex è un semaforo binario (0 oppure 1)

Un mutex va definito e inizializzato

ad esempio:

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER

definisce un semaforo di nome mutex e lo inizializza a 1

wait e signal

```
#include <pthread.h>

int pthread_mutex_lock( pthread_mutex_t  * mutex)
```

implementazione della funzione wait

```
#include <pthread.h>

int pthread_mutex_unlock( pthread_mutex_t  * mutex)
```

implementazione della funzione signal

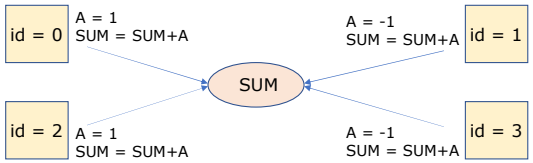
esempio: somma con variabile condivisa

scrivere un programma multithread (numero thread NT pari) tale che

tutti i thread condividono una variabile globale SUM

- i thread con indice pari sommano 100000 volte A = 1 alla variabile SUM
- i thread con indice dispari sommano 100000 volte A = -1 alla variabile SUM

Il risultato finale deve essere SUM = 0



somma multithread (1/3)

```
#include<pthread.h>
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<sys/types.h>
#include<unistd.h>
#include"c_timer.h"

struct argtype{
int id;         // identificativo thread
int  *SUM;     // somma elementi array
pthread_mutex_t *sem; // semaforo condiviso
};

int main( ){                                  // inizio main
          int nt, SUM;
          int  ciao( int);
          nt = 4;
          SUM = ciao(nt);
          printf(" esecuzione con %d thread, SUM = %d \n",nt, SUM);
}  //fine main
```

struttura per il passaggio di argomenti al thread

somma multithread (2/3)

```
int ciao(int nt){
        void *thread(void *) ;
        int i, sum;
        pthread_t  tid[4];
        struct argtype argmain[4];
        pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;

        sum=0.;
        for(i=0; i<nt; i++){
                argmain[i].id = i;
                argmain[i].SUM = &sum;
                argmain[i].sem = &mutex;
                pthread_create(&tid[i], NULL, thread, &argmain[i] );
        }

        for(i=0; i<nt; i++){
                pthread_join(tid[i], NULL);
        }
        return sum;
}
```

array di strutture da passare ai thread

definizione campi delle strutture e creazione thread

attesa terminazione dei thread

somma multithread (3/3)

```
void *thread (void *argmain){                  // inizio thread function
        pthread_mutex_t *sem;
        struct argtype  *argthread;
        int j, id, *SUM, a;

        argthread = (struct argtype *)argmain;
        id = argthread->id;
        SUM = argthread->SUM;
        sem = argthread->sem;

        if(id%2 == 0 ){
          a=1.;
        }else{
          a=-1.;
        }
        for (j=0;  j<100000  ; j++){
              pthread_mutex_lock(sem);
                *SUM = *SUM + a;
              pthread_mutex_unlock(sem);
        }
}
```

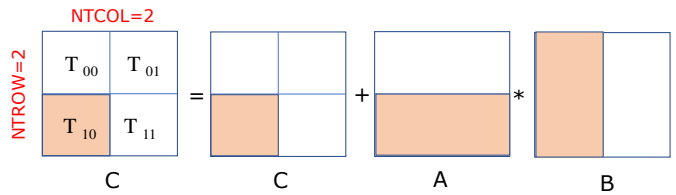
conversione del puntatore e copia su variabili locali

somma nella variabile condivisa

provare con e senza semaforo

elaborato: prodotto matrici multithread

- dividere la matrice C in blocchi pari al numero di thread,
- identificare i thread con una coppia di indici (cosi' da rispettare naturalmente la suddivisione della matrice in blocchi)
- Esempio: NTROW = 2, NTCOL = 2 (NT = 4 thread)



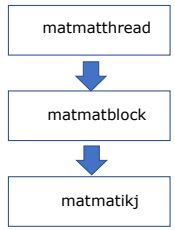
- ogni thread esegue il calcolo di un blocco della matrice C

elaborato: prodotto matrici multithread

- testata della function

```
void matmatthread (int lda, int ldb, int ldc,
        float *A, float *B, float *C,        int N,  int M,  int P,
        int dbN,   int dbM,   int dbP,       int ntrow,  int ntcol)
```

- riutilizzare la function matmatblock
- esperimenti con matrici di ordine fino a N=M=P=1500
- misurare speedup e efficienza con 1, 2 e 4 thread



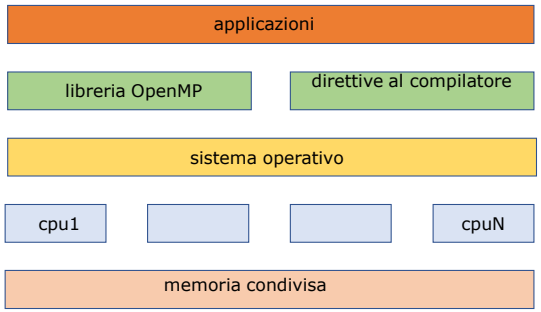
un cenno a OpenMP

- OpenMP e' un consorzio no-profit di
  - universita' (Tennessee, Manchester, Delaware, Basel,…)
  - centri di ricerca (Argonne, Oak Ridge, Sandia, Barcelona S.C., Inria, …)
  - aziende (AMD, Intel, ARM, NVIDIA, HP, Nec, SUSE, …)

finalizzato alla definizione di un insieme di API per la gestione dei thread

- Le API sono disponibili per vari linguaggi (C/C++, Fortran) e implementate in vari compilatori
- E' oramai uno standard de facto per il calcolo scientifico ad alte prestazioni
- modello a memoria condivisa
  - i thread comunicano attraverso variabili condivise
  - rischio di race condition, uso di strumenti di sincronizzazione

stack software



OpenMP gestisce una applicazione multithread attraverso

- libreria di routine
- direttive al compilatore

esempio

```
#include <stdio.h>
#include <omp.h>
int main(){

   omp_set_num_threads(4);

   #pragma omp parallel
   {
       printf( " hello world\n" );
   }
.....
}
```

esempio di funzione OpenMP

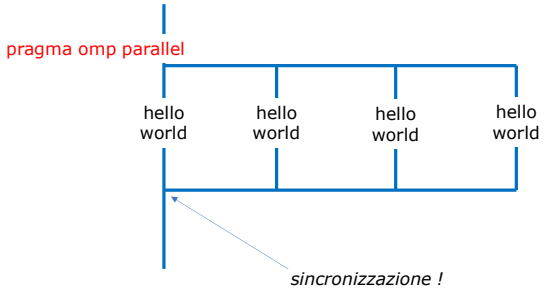
esempio di direttiva al compilatore (inizia con #)

corpo del thread (tra { })

modello di programmazione

modello fork-join

- il thread principale genera gli altri thread
- tutti i thread eseguono le stesse istruzioni
- i thread si sincronizzano al termine del blocco di istruzioni



cosa fa OpenMP

i due codici sono equivalenti

```
#include <stdio.h>
#include <omp.h>
int main{

omp_set_num_threads(4);
#pragma omp parallel
{
   printf( " hello world\n" );
}

}
```

versione OpenMP

versione pthread

```
#include <stdio.h>
int main{

for (i = 1; i < 4; i++){
   pthread_create(&tid[i],0,func,0);
}
func();

for (int i = 1; i < 4; ++i)
   pthread_join (tid[i]);

}


void func(){
   printf(" hello world\n");
}
```

Solo 3 thread sono creati, perché l'ultima sezione parallela è eseguita dal thread principale

principali funzioni

```
void omp_set_num_threads(int num_threads) ;
```

Imposta il numero di thread nelle successive aree parallele

```
int omp_get_num_threads();
```

ritorna il numero di thread nell'area parallela corrente

```
int omp_get_thread_num ();
```

ritorna l'identificativo del thread nel gruppo corrente

esempio

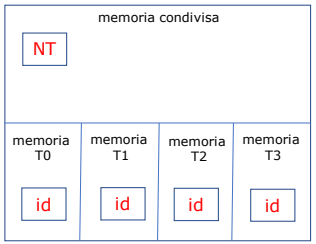
```
#include <stdio.h>
#include <omp.h>
int main{

   omp_set_num_threads(4);
   #pragma omp parallel
   {

        printf( " hello world from thread %d of %d\n",
                omp_get_thread_num () , omp_get_num_threads () );

   }

}
```

```
hello world from thread 0 of 4
hello world from thread 1 of 4
hello world from thread 2 of 4
hello world from thread 3 of 4
```


direttiva pragma omp parallel

- definisce una sezione parallela eseguita da thread distinti
- il codice dei thread e' specificato in un blocco strutturato racchiuso da { … }
- E' possibile dichiarare variabili locali al thread all'interno del blocco strutturato

```
#include <omp.h>
int main{
   int id, NT;        <- globale
   NT = 4;
   omp_set_num_threads(NT);

   #pragma omp parallel private(id)
   {
        …
   }                  <- locale

}
```



direttiva pragma omp critical

```
#include <omp.h>
#include <stdio.h>
int main{
   int NT , s, ID, k, a;

   NT = 4;
   s= 0;
   omp_set_num_threads(NT);
   #pragma omp parallel private(ID, k, a)
   {
        ID = omp_get_thread_num();

        a = 1 – 2*(ID%2);

        for (k = 0; k < 10000; k++){
           #pragma omp critical
           {
                s = s + a;
           }
        }
   }
   printf(" somma = %d\n", s);

}
```

- definisce una sezione critica.
- Il codice nel blocco strutturato viene eseguito da un thread alla volta
- pragma omp atomic e' applicabile ad una singola variabile ed ha un piu' basso overhead

pthreads vs OpenMP

- i pthreads sono uno strumento di basso livello che permette il controllo completo della gestione dei threads
- OpenMP e' uno strumento di piu' alto livello, non limitato al solo linguaggio C
- l'overhead per la gestione dei thread e' comparabile (leggermente piu' alto per OpenMP)
- OpenMP ben si presta ad applicazioni basate sul modello SPMD. Pthreads permette di far eseguire azioni diverse a thread differenti.
- Pthread richiede in genere un maggiore numero di linee di codice
- In sintesi:
  - OpenMP: piu' semplice da usare e adatto alla maggior parte delle applicazioni
  - pthread: piu' flessibile e piu' adatto a casi specifici