Programmazione parallela su sistemi shared memory

Sistemi HPC a memoria condivisa

Quando l'architettura prevede la condivisione di memoria tra nodi, è possibile utilizzare il modello di interazione tra processi a **memoria comune**.

In questo caso, a seconda del modello architetturale, esistono tecnologie di programmazione diverse:

- **sistemi multicore/multiprocessor**: **OpenMP**, pthreads ecc.
- **GP-GPU**: esistono librerie specifiche che consentono lo sviluppo di programmi destinati ad eseguire su GPU: **CUDA** (libreria proprietaria, tecnologia NVidia), **openCL.**

NB: MPI funziona anche su sistemi con condivisione di memoria tra nodi.

OpenMP: Open specifications for Multi-Processing.

E’ una libreria che permette di parallelizzare il codice di programmi C, utilizzando un approccio «dichiarativo». **(https://www.openmp.org)**

OpenMP offre strumenti per:

- **gestire i thread** paralleli
- **ottenere/impostare informazioni sull’ambiente di esecuzione** (quanti thread, qual è l’id del singolo thread, ecc)
- definire la **visibilità delle variabili** rispetto ai thread paralleli (shared, private, ecc.)
- **sincronizzare** i thread (sezioni critiche, barriere di sincronizzazione, ecc.)

Strumenti a disposizione del programmatore:

- direttive per il compilatore C: «`#pragma omp ...`».
- funzioni di libreria (previa inclusione header file `<omp.h>`)

Direttive # pragma omp

Esempio:

```
main()
{     ...
      # pragma omp parallel
            <blocco di istruzioni>
      ...
}
```

👉 il <blocco di istruzioni> viene eseguito da un insieme di thread paralleli. In questo caso (in assenza di indicazioni) la decisione sul numero di thread viene presa dal sistema (di solito: 1 thread/core disponibile).

Un primo programma OpenMP

```
/* hello.c */

#include <stdio.h>

int main( int argc, char **argv) {

    int n=atoi(argv[1]);

    #pragma omp parallel num_threads(n)
    {
        printf("Ciao !\n");
    }
}
```

direttiva

clausola

Compilazione e esecuzione:

```
$ gcc -fopenmp -o ciao hello.c
$./ciao 4
Ciao !
Ciao !
Ciao !
Ciao !
```

La printf viene eseguita da 4 thread paralleli: il processo iniziale (Master) + altri 3 (Workers) creati per effetto della direttiva:

```
# pragma omp parallel num_threads(n)
```

attraverso la clausola **num_threads**, si specifica il numero dei processi in esecuzione parallela

ogni processo esegue il blocco parallelo (printf)

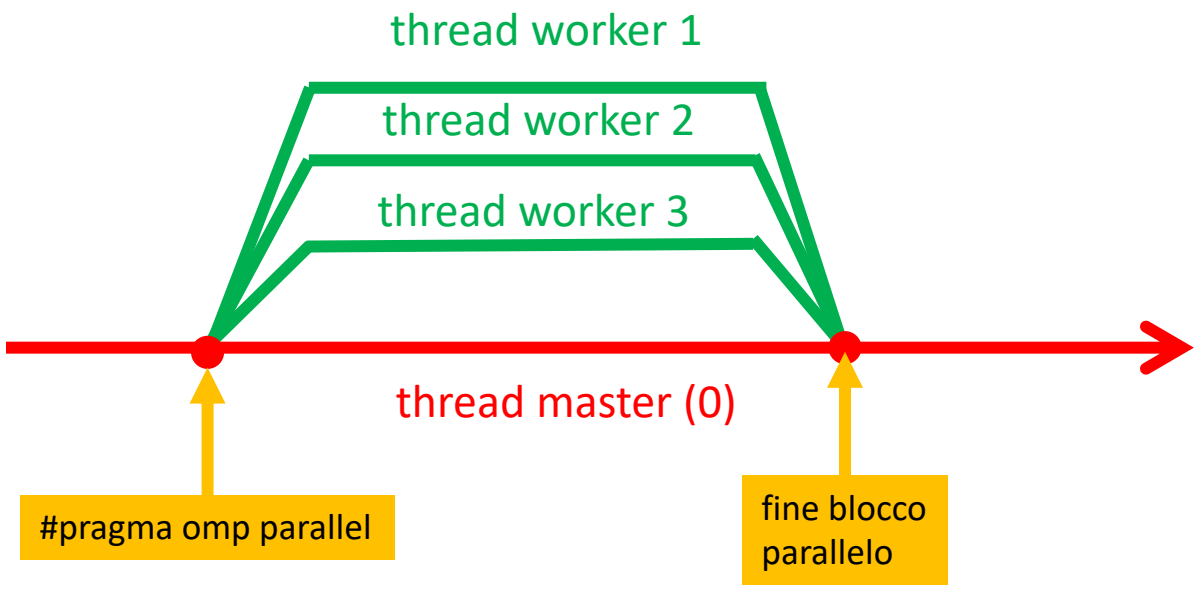
Un primo programma OpenMP

```
/* hello.c */

#include <stdio.h>

int main( int argc, char **argv) {

    int n=atoi(argv[1]);

    #pragma omp parallel num_threads(n)
    {
        printf("Ciao !\n");
    }
}
```

direttiva

clausola

```
$./ciao 4
Ciao !
Ciao !
Ciao !
Ciao !
```

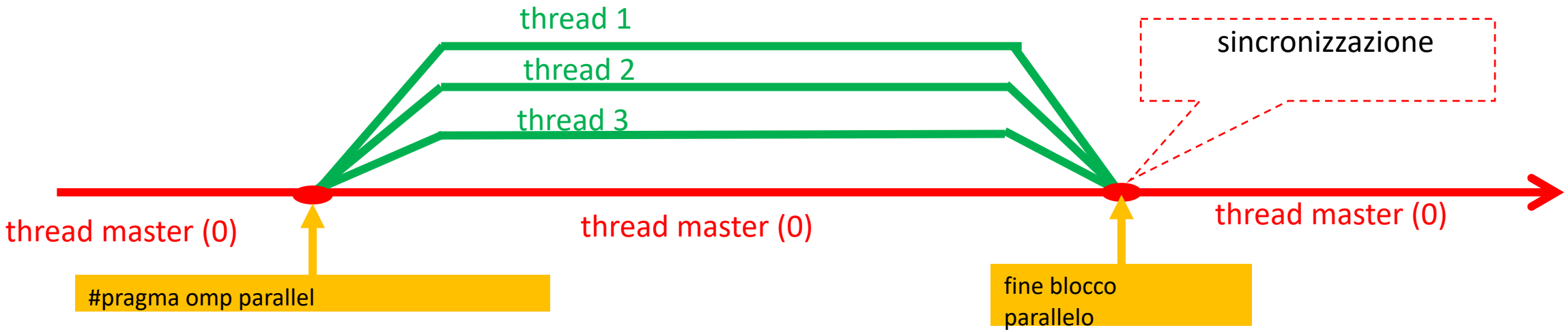


Creazione thread: `# pragma omp parallel`

La creazione dei thread paralleli è implicita e avviene per effetto della direttiva:

```
# pragma omp parallel ...
    <blocco parallelo>
```

- Modello **cobegin-coend** puro
- **sincronizzazione implicita**: al termine del blocco parallelo il processo master attende che tutti i workers siano terminati.



Clausole # pragma omp parallel

Sintassi:

```
# pragma omp parallel [eventuali clausole]
```

Clausole:

- **num_threads(N): (v.esempio)** il numero di thread paralleli è N (compreso il master). Il gruppo di thread paralleli [1 master e (N-1) workers]è il **team**
- **shared, private, firstprivate:** per specificare lo scope di visibilità di variabili
- **if:** parallelizzazione condizionale
- **for**: per la parallelizzazione di un ciclo
- **reduction**: per specificare operazioni di reduce
- **schedule**: per gestire la suddivisione del workload tra thread paralleli

Scope delle variabili

Quali variabili sono **condivise** dai processi e quali **private**?

Default:

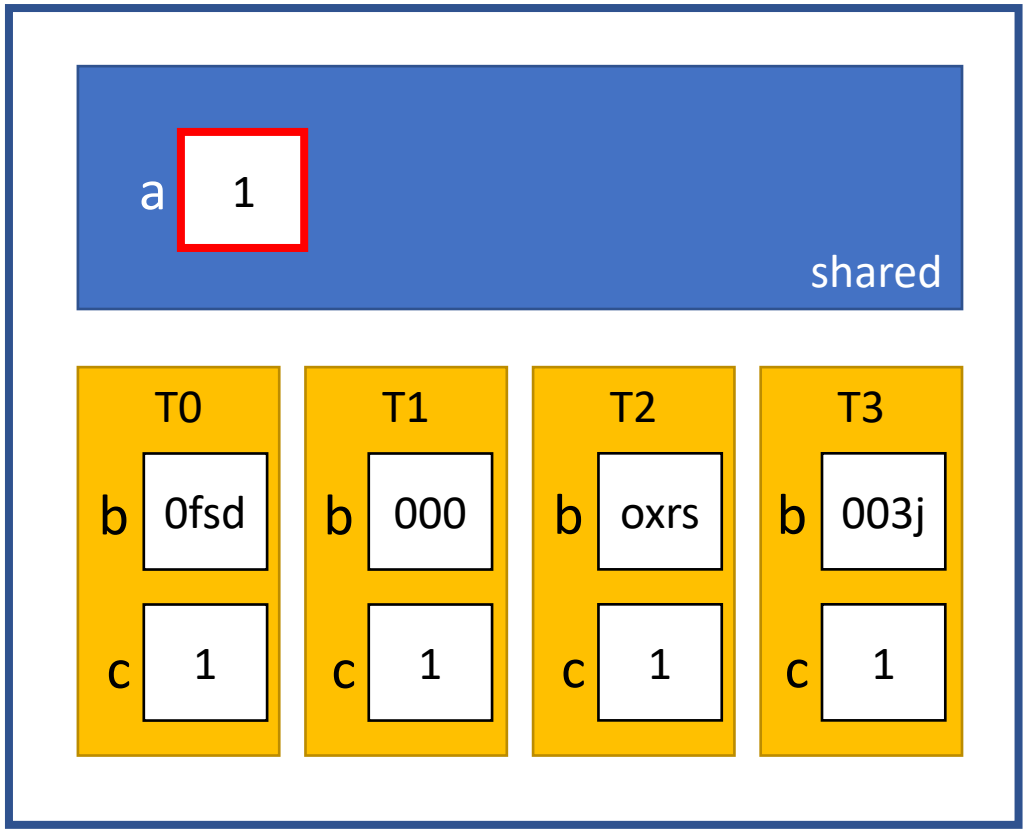
- ogni variabile definita esternamente a # pragma omp parallel è **condivisa** tra i thread paralleli
- ogni variabile definita internamente al blocco parallelo è locale al singolo thread

👉C'è la possibilità di specificare diversamente dal default con le clausole **shared**, **private**, **firstprivate**

Visibilità delle variabili

Esempio:

```
#include <omp.h>
#include <stdio.h>
int main( int argc, char *argv[] )
{
    int a=1, b=1, c=1;
    #pragma omp parallel shared(a) \
        private(b) firstprivate(c)
    {   a++;
        b++;
        c++;
    }
}
```



a,b,c sarebbero shared (default), ma, per effetto delle clausole:

- **private(b):** b è privata; ogni processo ne usa una istanza locale non inizializzata;
- **firstprivate(c)**: ogni processo usa una copia privata di c inizializzata al valore che c aveva prima di # pragma
- **shared(a)**: a è condivisa tra tutti i thread paralleli (è il default →la clausola è superflua).

Mutua esclusione: direttiva critical

L'accesso a variabili shared da parte di thread paralleli va sincronizzato: è necessario imporre la **mutua esclusione** nell'esecuzione di sezioni critiche.

A questo scopo è possibile usare la direttiva:

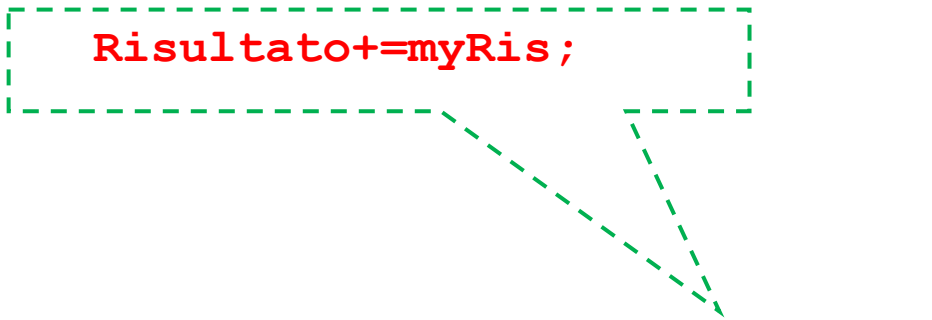
```
#   pragma omp critical
      <sezione critica>
```

👉 Il blocco di istruzioni immediatamente successivo alla direttiva viene eseguito da un solo processo alla volta.

Mutua esclusione: esempio

Calcolo della sommatoria di 20 valori:

```
main()
{       int Risultato=0;
        # pragma omp parallel num_threads(20)
        {       int myRis;
                myRis=Calcolo(..);
                # pragma omp critical
                Risultato+=myRis;
        }
        ...
}
```



Ambiente di esecuzione

In generale, il grado di parallelismo viene deciso a runtime.

Per gestire le informazioni sui thread in esecuzione, sono disponibili le funzioni:

- **`omp_get_num_threads()`:** restituisce il **numero di thread** paralleli → da utilizzare all'interno di un blocco parallelo.
- **`omp_get_thread_num()`:** restituisce il **rank** del thread che lo invoca (0 è il master).
- **`omp_set_num_threads(n)`: imposta a n il numero di thread** paralleli nei successivi blocchi paralleli.

Esempio:

```
omp_set_num_threads(3);
#pragma omp parallel
     printf(" Hello\n"); // eseguita da 3 thread
```

Esempio: calcolo dell'integrale con il metodo dei trapezi

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
void Trap(double a, double b, int n, double* global_result_p);
double f(double x){return x*x;}

int main (int argc, char* argv[]) {
    double a, b, global_result = 0.0;
    int n, thread_count;
    printf("Dammi a, b, and n:\n"); scanf("%lf %lf %d", &a, &b, &n);
    thread_count = atoi(argv[1]);
#   pragma omp parallel num_threads(thread_count)
       Trap(a, b, n, &global_result);
    printf("Con n = %d trapezi il risultato dell'integrale da %f a %f = %.14e\n",a, b, global_result);
    return 0;
}
```

```
void Trap(double a, double b, int n, double* Ris)
{   double h, x, my_result,local_a, local_b;
    int i, local_n;
    int my_rank = omp_get_thread_num(); //funzione che restituisce il rank del thread corrente
    int thread_count = omp_get_num_threads(); //funzione che restituisce numero totale di thread
    h=(b-a)/n;
    local_n = n/thread_count;
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    my_result = (f(local_a) + f(local_b))/2.0;
    for (i = 1; i <= local_n-1; i++) {
        x = local_a + i*h;
        my_result += f(x);
    }
    my_result = my_result*h;
#   pragma omp_critical
    *Ris += my_result;  // sezione critica: un thread alla volta
}
```

Osservazioni sull'esempio

La soluzione proposta utilizza una versione della funzione Trap specificamente concepita per essere eseguita da thread paralleli:

```
void Trap(double a, double b, int n, double* Ris)
{     ....
      #   pragma omp_critical
      *Ris += my_result; // effetto collaterale
}
```

E' possibile una soluzione in cui la Trap sia la stessa della versione seriale (senza # pragma..)?

Trap: versione alternativa

```
double Trap(double a, double b, int n)
{       double h, x, my_result, local_a, local_b;
        int i, local_n;
        h=(b-a)/n;
        int my_rank = omp_get_thread_num();
        int thread_count = omp_get_num_threads();
        local_n = n/thread_count;
        local_a = a + my_rank*local_n*h;
        local_b = local_a + local_n*h;
        my_result = (f(local_a) + f(local_b))/2.0;
        for (i = 1; i <= local_n-1; i++) {
                x = local_a + i*h;
                my_result += f(x);
        }
    my_result = my_result*h;
    return my_result;
    }
```

Versione modificata esempio trapezi:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
double Trap(double a, double b, int n);
double f(double x){return x*x;}

int main (int argc, char* argv[]) {
    double global_result = 0.0;
    ...
    #   pragma omp parallel num_threads(thread_count)
    {   double my_ris;
        my_ris=Trap(a, b, n);
        # pragma omp critical
                global_result+=my_ris;
    }
    ...
}
```

Versione modificata esempio trapezi:

Notiamo che al termine della parte parallela, tutti i risultati locali vengono aggregati in una sommatoria → incremento in modo mutuamente esclusivo della variabile condivisa **`global_result`** (# pragma omp critical) → operazione di «**reduction**»

In OMP è disponibile la clausola **`reduction`** da utilizzare congiuntamente al blocco parallelo.

Clausola reduction:

Associa a un blocco parallelo una variabile condivisa <var> che viene utilizzata per memorizzare il risultato di un'operazione <op> di reduce:

```
#    pragma omp parallel ... reduction(<op>: <Var>)
{     ...
      <Var><op>=<espressione>;
```

L'espressione viene **valutata in parallelo**; l'aggiornamento di <Var> avviene in **mutua esclusione**.

Operazioni possibili: `+, *, &&, ||, &, |, ...`

Clausola reduction:

Esempio:

```
double Ris; // non è importante inizializzarla
#    pragma omp parallel ... reduction(+: Ris)
     {
      Ris+=Trap(a, b, n);
     }
```

- la funzione Trap viene eseguita parallelamente dai thread, ma l'incremento della variabile Ris (associata a reduction) viene fatto in modo mutuamente esclusivo
- la variabile Ris viene inizializzata automaticamente a 0 (elemento neutro della somma)

Versione modificata esempio trapezi:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
double Trap(double a, double b, int n);
double f(double x){return x*x;}
int main (int argc, char* argv[]) {
    double a, b, global_result; // non è necessario inizializzare global_result
    int n, thread_count;
    printf("Dammi a, b, and n:\n"); scanf("%lf %lf %d", &a, &b, &n);
    thread_count = atoi(argv[1]);
    #   pragma omp parallel num_threads(thread_count) reduction(+:global_result)
        global_result+=Trap(a, b, n);
    printf("Con n = %d trapezi il risultato \n", n);
    printf("dell'integrale da %f a %f = %.14e\n",a, b, global_result);
    return 0;
}
```

Clausola if

E’ possibile condizionare la parallelizzazione al verificarsi di una condizione con la clausola if.

Esempio:

```
int main( int argc, char *argv[] ) {
t=atoi(argv[1]);
#pragma omp parallel if (t > 10) num_threads(t)
     f(); //eseguita in parallelo da t threads, solo se argv[1]>10;
          //altrimenti pragma è ignorato (esecuzione seriale).
}
```

Direttiva master

Permette di far eseguire un blocco di istruzioni al thread Master (rank 0).

Esempio:

```
...
int main( int argc, char **argv) {

#pragma omp parallel num_threads(12)
{     int rank=omp_get_thread_num();
      int T=omp_get_num_threads();
      printf("sono il thread %d di %d\n", rank, T);
      #pragma omp master
        printf("MASTER: sono il thread %d di %d\n", rank, T);
}
```

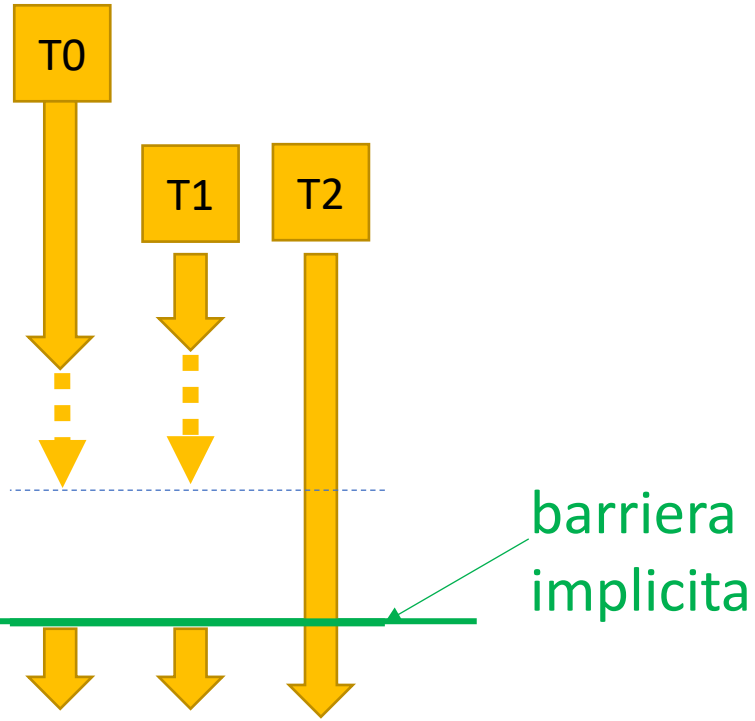
Direttiva single

Fa eseguire un blocco di istruzioni ad uno solo tra i thread paralleli.

Quale? → scelta non deterministica

Esempio:

```
...
int main( int argc, char **argv) {

#pragma omp parallel num_threads(3)
{       int rank=omp_get_thread_num();
        int T=omp_get_num_threads();
        # pragma omp single
          printf("Single: sono il thread %d di %d\n", rank, T);
        printf("il thread %d di %d ha finito\n", rank, T);

}
```



 # pragma omp single realizza una **barriera di sincronizzazione implicita**: ogni thread, prima di eseguire l'istruzione successiva al blocco single (nell'es. la printf finale), attende tutti i thread (compreso il single) abbiano terminato le istruzioni precedenti.

Parallelizzazione di cicli: clausola for

Un'esigenza comune nella programmazione parallela è la **parallelizzazione di cicli.**

in openMP si può usare la clausola for:

```
#pragma omp parallel for [altre clausole..]
   <ciclo for di K iterazioni>
```

👉se n è il num di thread paralleli, ogni thread esegue un insieme (***chunk)*** di K/n iterazioni contigue.

Parallelizzazione di cicli: clausola for

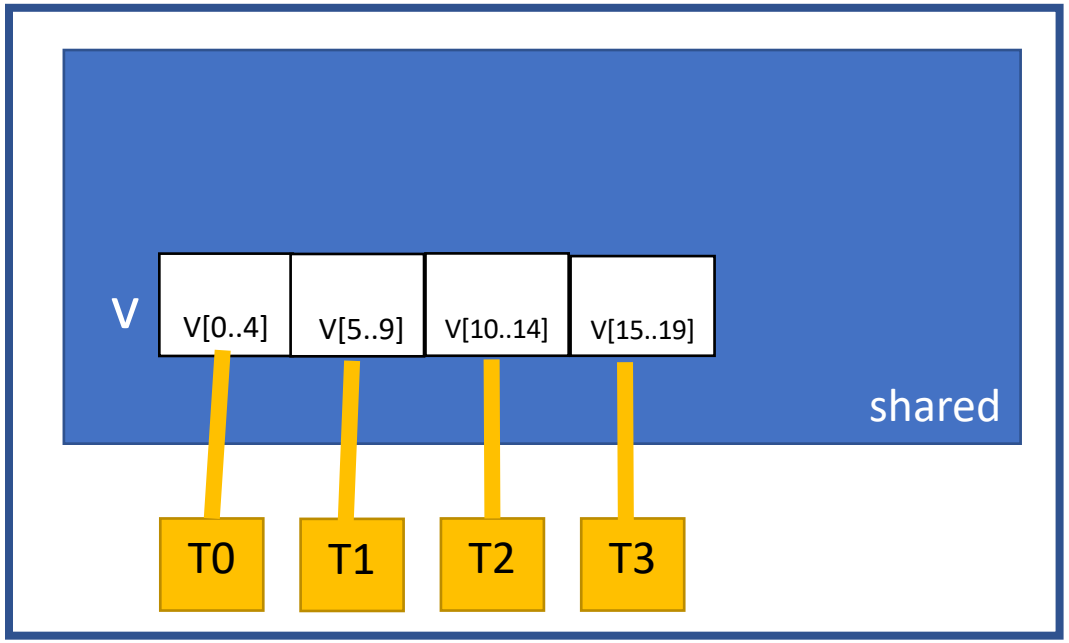
Esempio:

```
#pragma omp parallel for num_threads(4)
    for(i=0; i<20; i++)
      v[i] = i*i;
```

👉 Il workload viene suddiviso tra 4 threads (5 iterazioni per ognuno):

- T0 esegue le prime 5 iterazioni (i ∈ [0,..4])
- T1 esegue le succ. 5 iterazioni (i ∈ [5,..9])
- T2 esegue le succ. 5 iterazioni (i ∈ [10,..14])
- T3 esegue le succ. 5 iterazioni (i ∈ [15,..19])

→«**Block**» workload distribution:
a ogni thread è assegnato un **chunk** di 5 iterazioni contigue.

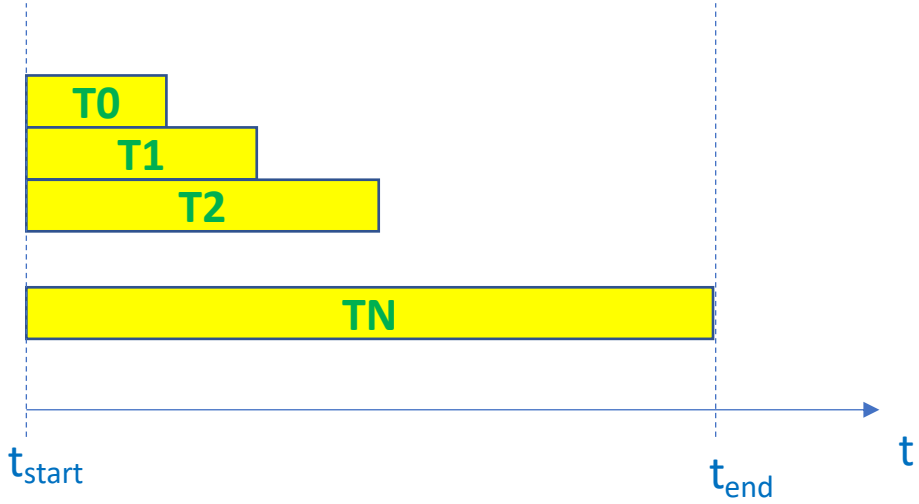


Distribuzione del carico

Non sempre la suddivisione a blocchi è quella ottimale. E' possibile che il carico di iterazioni distinte sia diverso.

Ad esempio:

```
sum = 0.0;
for (i = 1; i <= n; i++)
      for (j=0; j<i; j++)
            sum += f(i);
```



considero il for esterno:

- la prima iterazione (i=1) chiama la funzione f una volta
- la seconda iterazione (i=2) chiama f due volte
- ...
- l'ultima iterazione (i=n) chiama la f n volte

Una parallelizzazione a blocchi produrrebbe un **carico sbilanciato**: il primo thread lavora meno del secondo, che lavora meno del terzo ecc.

E' possibile ottenere distribuzioni diverse attraverso la clausola **schedule.**

Clausola schedule

Tramite **`schedule`** è possibile realizzare distribuzioni diverse dalla block distribution:

```
#pragma omp parallel for schedule(<tipo>, <chunksize>)
```

Dove:

- `<tipo>` indica il tipo di distribuzione (esempio: `static, dynamic`)
- `<chunksize>` indica l'unità di carico (espressa in numero di iterazioni)

L'uso di schedule determina un'assegnazione del workload secondo la politica associata al tipo.

Clausola schedule: static

il tipo di scheduling **`static`** prevede che la distribuzione del lavoro venga fatta assegnando i diversi chuncks ai thread «staticamente» secondo una politica **round-robin**.

Esempio:

```
#pragma omp for num_thread(4) schedule (static, 1)
for(i=0; i<20; i++)
        v[i] = i*i;
```

Lo scheduler assegnerà (chunk di dim 1):

al thread 0 le iterazioni 0, 4, 8, 12, 16

al thread 1 le iterazioni 1, 5, 9, 13, 17

al thread 2 le iterazioni 2, 6, 10, 14, 18

al thread 1 le iterazioni 3, 7, 11, 15, 19

👉utile per distribuire il carico in modo bilanciato, quando dipende in modo deterministico dal numero di iterazione.

Clausola schedule: dynamic

L'assegnamento del lavoro ai thread avviene dinamicamente: quando un thread termina l'elaborazione di un chunk, lo scheduler gli assegna il prossimo chunk (seguendo l'ordine degli indici).

Ad esempio:

```
#pragma omp for num_thread(4) schedule (dynamic, 2)
for(i=0; i<20; i++)
        v[i] = i*i;
```

Inizialmente lo sheduler assegnerà (chunk di dim 2):

al thread 0 le iterazioni 0,1

al thread 1 le iterazioni 2,3

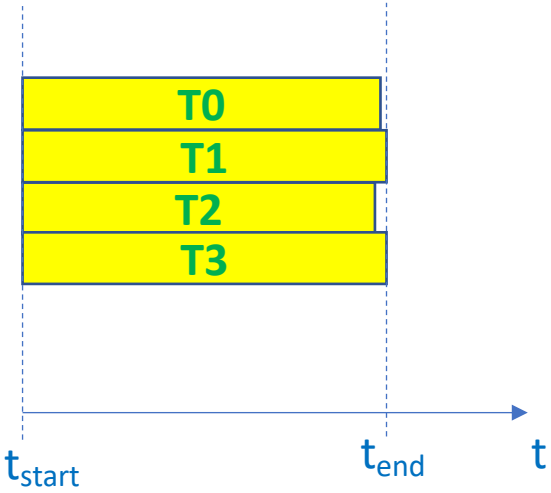
al thread 2 le iterazioni 4,5

al thread 3 le iterazioni 6,7

Al primo thread che termina il suo primo chunk verrà assegnato il chunk 8,9

Al successivo thread che termina il suo primo chunk verrà assegnato il chunk 10,11

ecc.



👉in questo modo è possibile mantenere il carico bilanciato, anche in presenza di non determinismo nel numero di operazioni da eseguire ad ogni iterazione

Sincronizzazione tra thread in OMP

Oltre alla clausola **`critical`** per la mutua esclusione, OMP offre altri strumenti per realizzare in modo esplicito la sincronizzazione tra thread paralleli:

- direttiva **`barrier`**: implementa la classica barriera di sincronizzazione in un team di threads.
- gestione di **`lock`**: tramite le funzioni **`omp_set_lock`** e **`omp_unset_lock`** è possibile realizzare schemi di sincronizzazione ad hoc tra threads.

Sincronizzazione: barrier

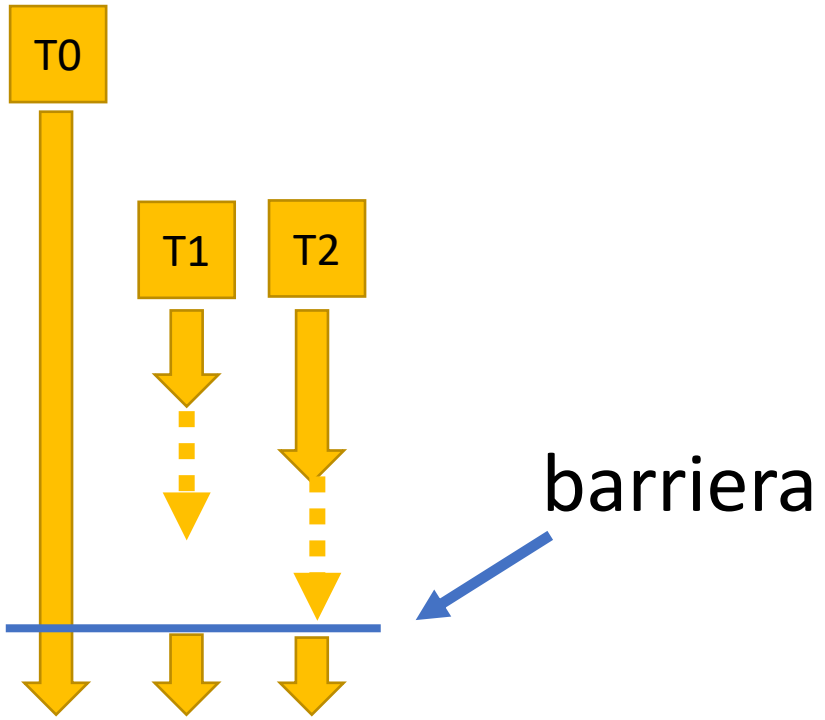
Quando, all'interno di un blocco parallelo si vuole imporre un punto di sincronizzazione tra tutti i processi, è possibile utilizzare la direttiva

```
# pragma omp barrier
```

che realizza una classica barriera di sincronizzazione.

Esempio:

```
int main( int argc, char *argv[] ) {
    int v[MAX];
    a=8;
    #pragma omp parallel num_threads(3)
    {
        int iam = omp_get_thread_num();
        v[iam] = iam*iam;
        #pragma omp master
        a += 800; //ran only by one thread
        #pragma omp barrier
        printf("thread %d: valore di a: %d\n", iam,a);
    }
}
```



Sincronizzazione: lock

Come in tutti gli ambienti di programmazione basati sul modello a memoria comune, anche in OMP sono disponibili strumenti specifici per imporre politiche di sincronizzazione particolari a thread concorrenti.

A questo scopo è disponibile il **lock**: è implementato da una variabile del tipo `omp_lock_t` condivisa tra i thread che devono sincronizzarsi, alla quale è possibile applicare le funzioni:

- `omp_set_lock(..)`
- `omp_unset_lock()`

} acquisizione e rilascio di lock (v. lock e unlock)

- `omp_init_lock(..)`, `omp_destroy_lock(..)`: creazione ed eliminazione di lock.

lock

Un lock è un oggetto che ha 2 stati: libero e occupato.

Inizializzazione: ogni lock va preventivamente creato e inizializzato (a libero) con la funzione: `void omp_init_lock(omp_lock_t *L);`

Operazioni su lock:

- `void omp_set_lock(omp_lock_t *L);` blocca il thread che esegue la funzione fino a quando il lock L non è libero; quindi occupa L (→ stato **occupato**).
- `void omp_unset_lock(omp_lock_t *L);` libera il lock (→ stato libero).

Eliminazione: quando il lock non viene più usato, va eliminato con la funzione:
`void omp_destroy_lock(omp_lock_t *lock);`

Esempio: sezione critica

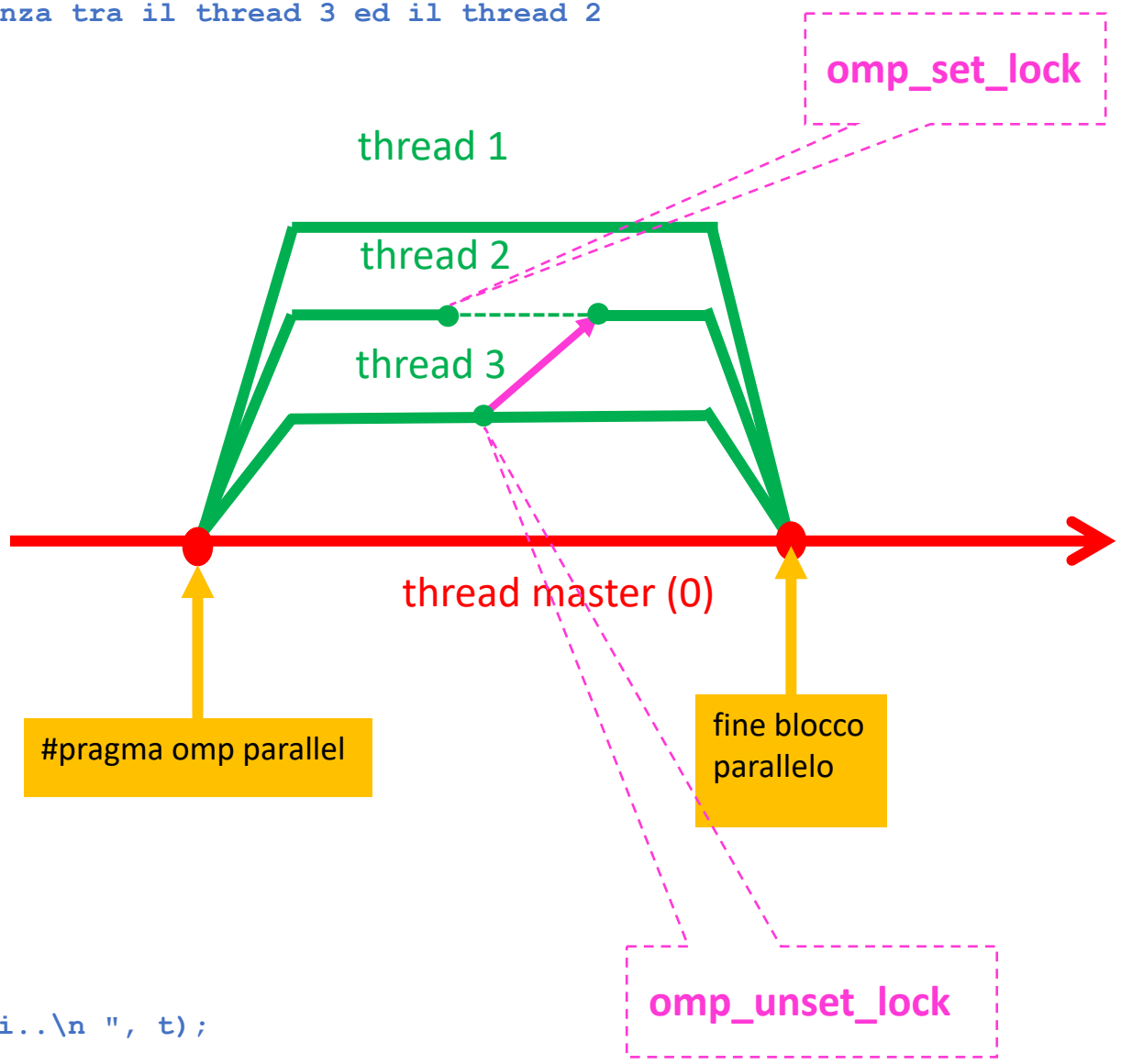
```
// il lock viene utilizzato per realizzare sezioni critiche
#include <stdio.h>
#include <omp.h>
omp_lock_t my_lock;
int main() {
   omp_init_lock(&my_lock);
   #pragma omp parallel num_threads(4)
   {    int i, j, t = omp_get_thread_num( );
        for (i = 0; i < 5; ++i) {
          omp_set_lock(&my_lock); //prologo sezione critica
          printf("Thread %d - inizio sezione critica %d\n", t,i);
          printf("Thread %d - fine sezione critica %d\n", t, i);
          omp_unset_lock(&my_lock); //epilogo sezione critica
       }
   }
omp_destroy_lock(&my_lock);
}
```

equivalente a:

```
# pragma omp critical
```

Esempio: vincolo di precedenza

```
// il lock viene utilizzato per imporre un vincolo di precedenza tra il thread 3 ed il thread 2
#include <stdio.h>
#include <omp.h>
omp_lock_t my_lock;
int main() {
   omp_init_lock(&my_lock);
   omp_set_lock(&my_lock); //il master occupa il lock
   #pragma omp parallel num_threads(4)
   {  int t = omp_get_thread_num( );
      if (t==3)
         {          printf("Thread %d: prima..\n", t);
                    omp_unset_lock(&my_lock);
          }
      else if (t==2)
      {   omp_set_lock(&my_lock);
          printf("Thread %d: dopo..\n", t);
      }
      else           printf("Thread %d: sono libero da vincoli..\n ", t);
} // fine blocco parallelo
omp_destroy_lock(&my_lock);  }
```



Misurazione del tempo

Per misurare il tempo è disponibile la funzione:

```
double omp_get_wtime(void);
```

che restituisce il tempo (in secondi) trascorso da un certo istante iniziale (fissato dal sistema). («wallclock»)

👉 Per misurare il tempo trascorso tra l'inizio e la fine dell'esecuzione di un programma:

```
#include "omp.h"
#include <stdio.h>
#include <windows.h>

int main() {
    double tempo, end, start;
    start = omp_get_wtime( ); // Master: istante iniziale
    <istruzioni programma OMP>
    end = omp_get_wtime( ); // Master: istante finale
    tempo=end-start;
    printf("tempo di esecuzione: %lf", tempo);

}
```

OpenMP vs. pthread

Pthread:

- paradigma MPMD, ovvero il programmatore definisce la (potenzialmente diversa) sequenza di istruzioni che ogni singolo processo parallelo eseguirà (v. pthread_create)
- creazione dei processi: modello **fork-join**.
- ampio set di strumenti per esprimere politiche di sincronizzazione specifiche (mutex,semafori, condition,..)
- Adatto per algoritmi task-parallel.

OpenMP:

- approccio di più alto livello, basato su SPMD.
- La creazione dei processi segue lo schema **cobegin-coend**.
- Sincronizzazione tramite direttive/clausole che implementano schemi predefiniti (barrier, critical, reduction..). Politiche di sincronizzazione ad hoc possono essere realizzate combinando lock e schemi predefiniti.
- Particolarmente adatto per la modellazione di algoritmi Data-parallel.

MPI vs. OpenMP

MPI:

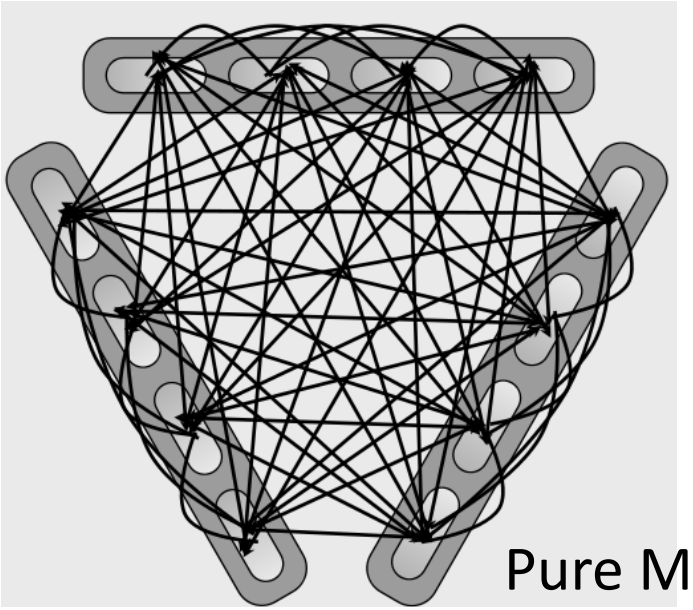
- interazione basata su **message passing**
- **complessità** di uso (spesso)
- bilanciamento del carico a carico del programmatore
- elevata **scalabilità**
- elevata **portabilità**: funziona anche su shared memory systems

OpenMP:

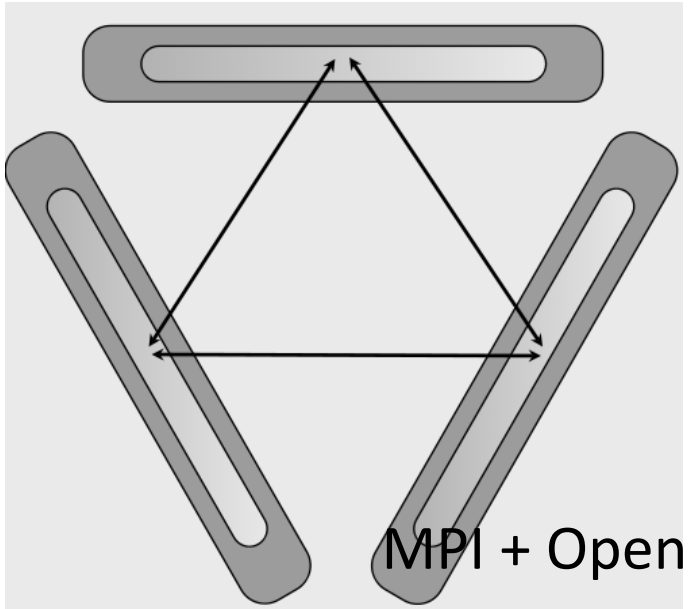
- interazione basata su **memoria condivisa**
- **semplicità** di uso (nella maggior parte dei casi)
- load balancing semplice da realizzare
- scalabilità limitata al numero di CPU disponibili sul nodo utilizzato
- compatibile solo con shared memory systems (multicore/multiprocessors)

Hybridization

E’ possibile combinare MPI e OpenMP per beneficiare dei vantaggi di entrambi:



Pure MPI



MPI + OpenMP

Approccio interessante quando si ha un carico di comunicazione elevato (ad es: molte comunicazioni collettive)

image credits: CINECA summer school on HPC programming

Hybridization in pratica

```
int main( … ) {
   MPI_Init_thread(&argc, &argv, requested, &provided);
   …
   #pragma omp parallel
   {     …     }
}
```

livello di supporto ai thread (richiesto e ottenuto)

al posto di `MPI_Init()`

Livello di supporto:

- **MPI_THREAD_SINGLE**: only one thread can run during MPI calls => all MPI calls must be outside the OpenMP pragma sections
- **MPI_THREAD_FUNNELED**: MPI calls can be inside pragmas but they are performed only by the master thread
- **MPI_THREAD_SERIALIZED**: MPI calls by multiple threads are allowed but only one at a time
- **MPI_THREAD_MULTIPLE**: MPI calls can be performed by multiple threads with no restrictions
- …

[https://www.openmp.org//wp-content/uploads/HybridPP_Slides.pdf]

Bibliografia

- Peter Pacheco, An Introduction to Parallel Programming, Morgan Kaufmann, 2011
- J.Patterson, D.Hennessy:Computer Architecture: A Quantitative Approach, Elsevier, 2017.
- https://www.mpi-forum.org
- https://www.openmp.org/
- https://www.rookiehpc.com/index.php → documentazione e tutorial