

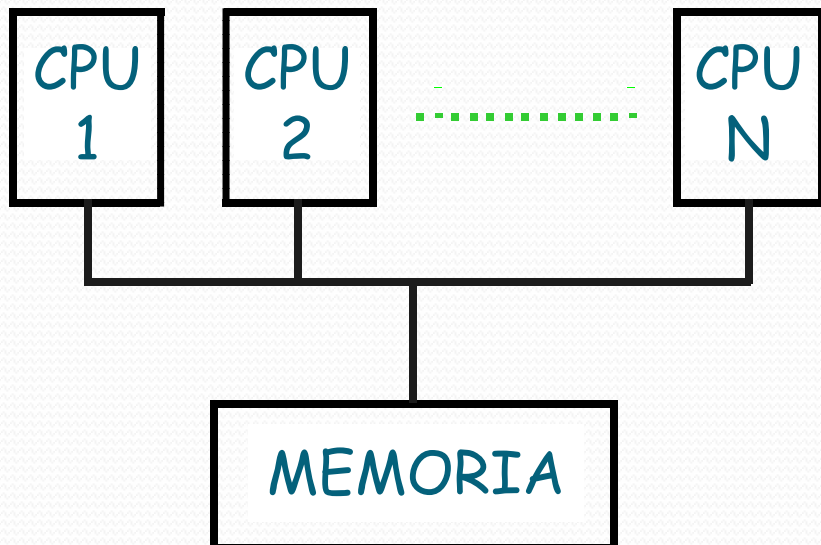
# OpenMp

Introduzione agli strumenti

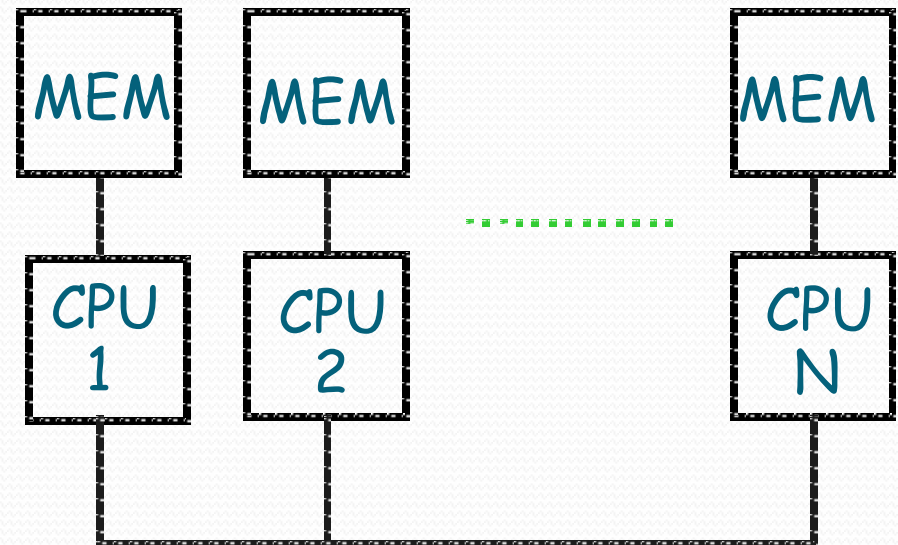
Prof. G. Laccetti

# Shared vs Distributed

Calcolatori MIMD a  
memoria **condivisa**  
(shared-memory)



Calcolatori MIMD a  
memoria **distribuita**  
(distributed-memory)

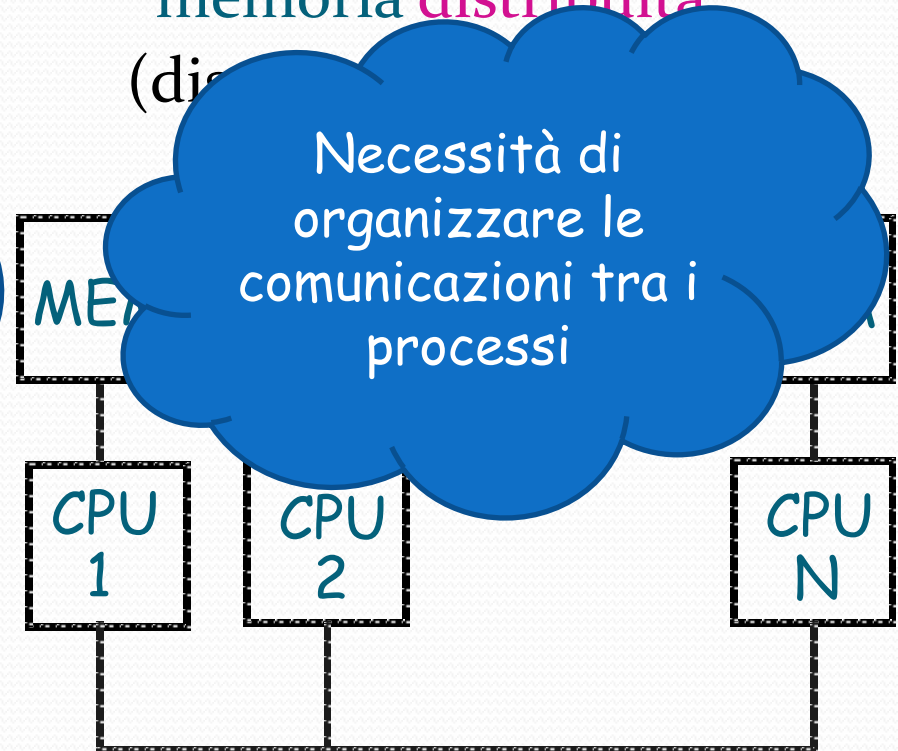


# Shared vs Distributed

Calcolatori MIMD a  
memoria **condivisa**

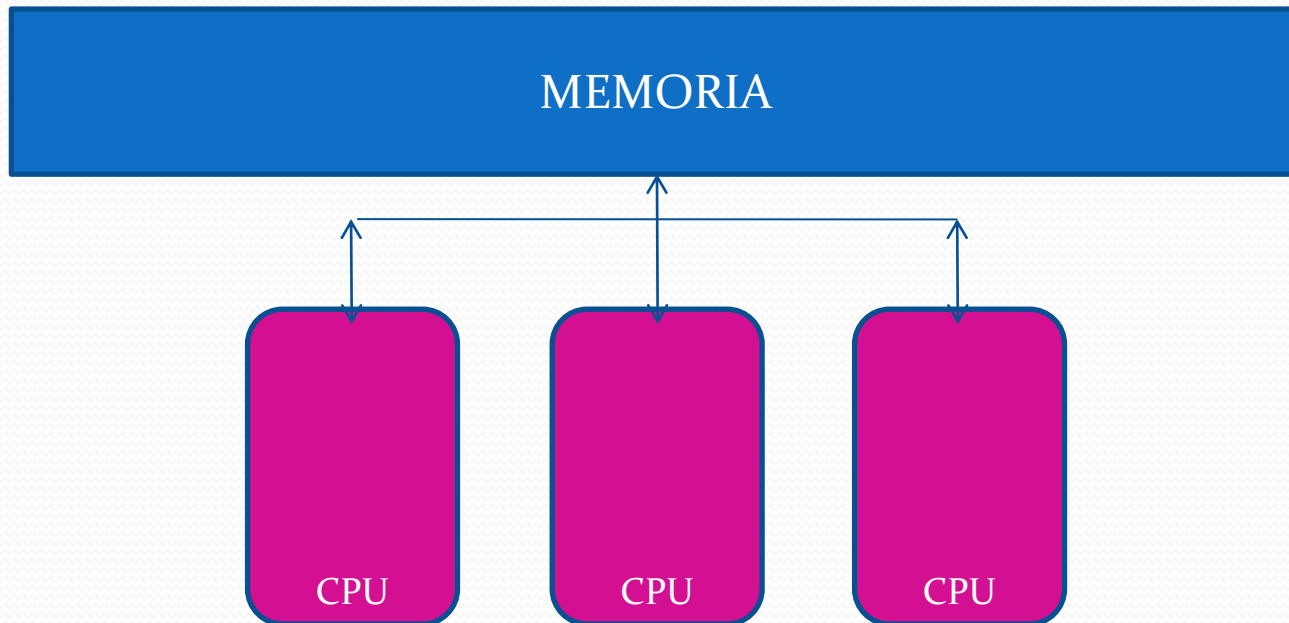


Calcolatori MIMD a  
memoria **distribuita**



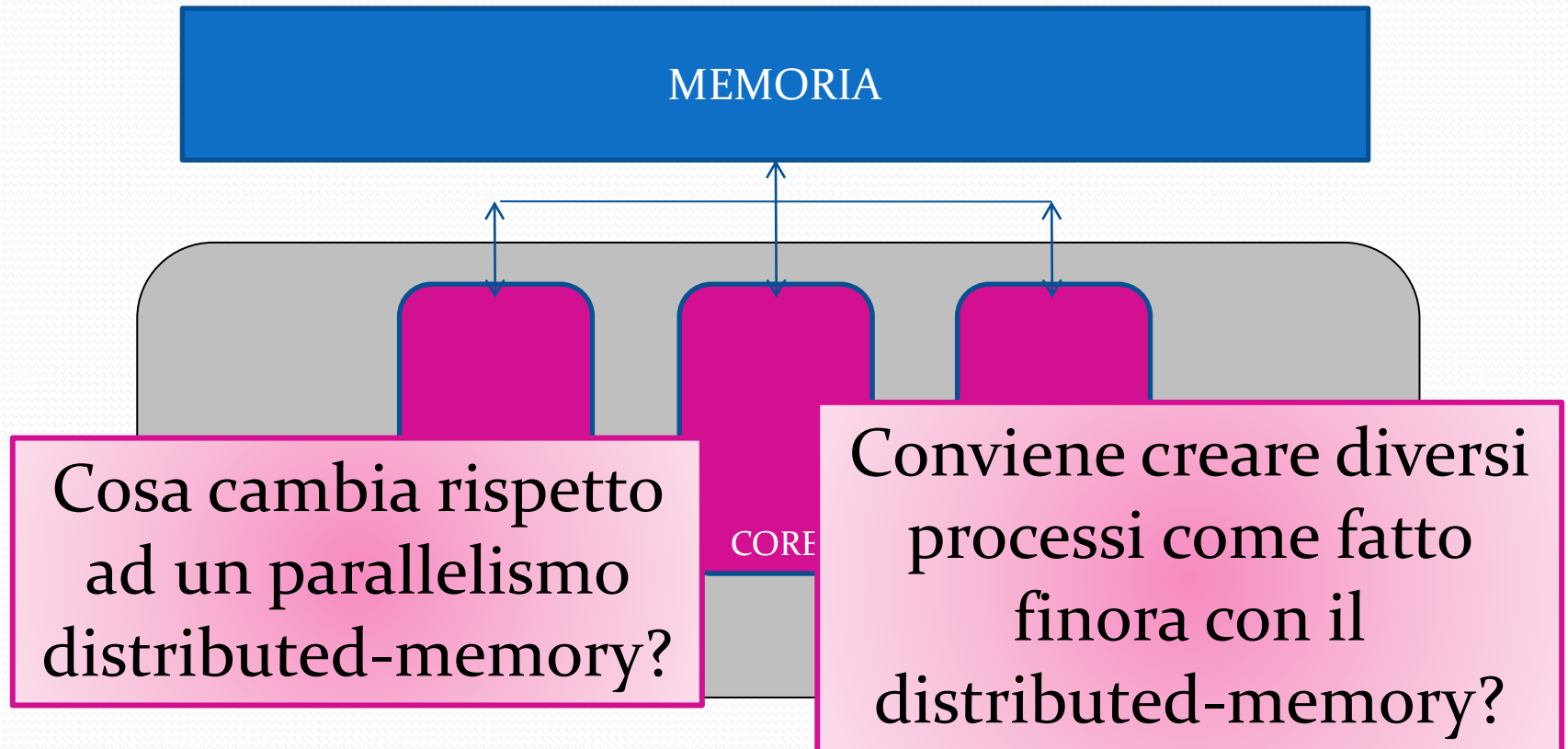
# Shared Memory:

Se il sistema è tale che diverse CPU sono collegate alla stessa memoria fisica, è in genere opportuno un parallelismo shared-memory



# Shared Memory: Multicore

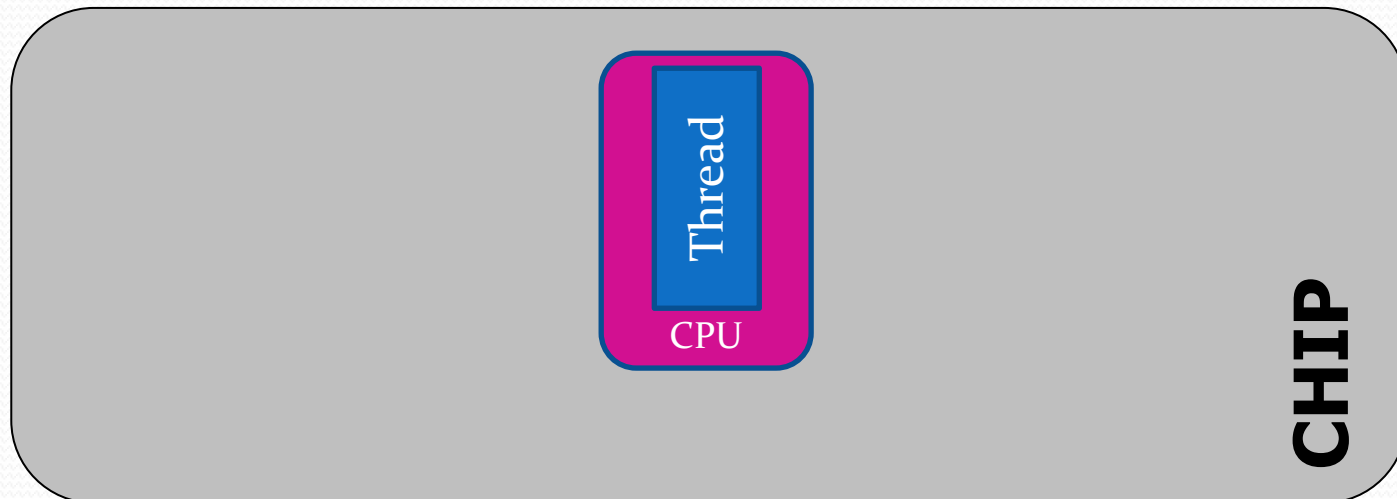
Se si tratta di un sistema *multicore* la situazione non è diversa



# Threads

Per un sistema operativo moderno, l'unità base di utilizzo della CPU è il **thread**

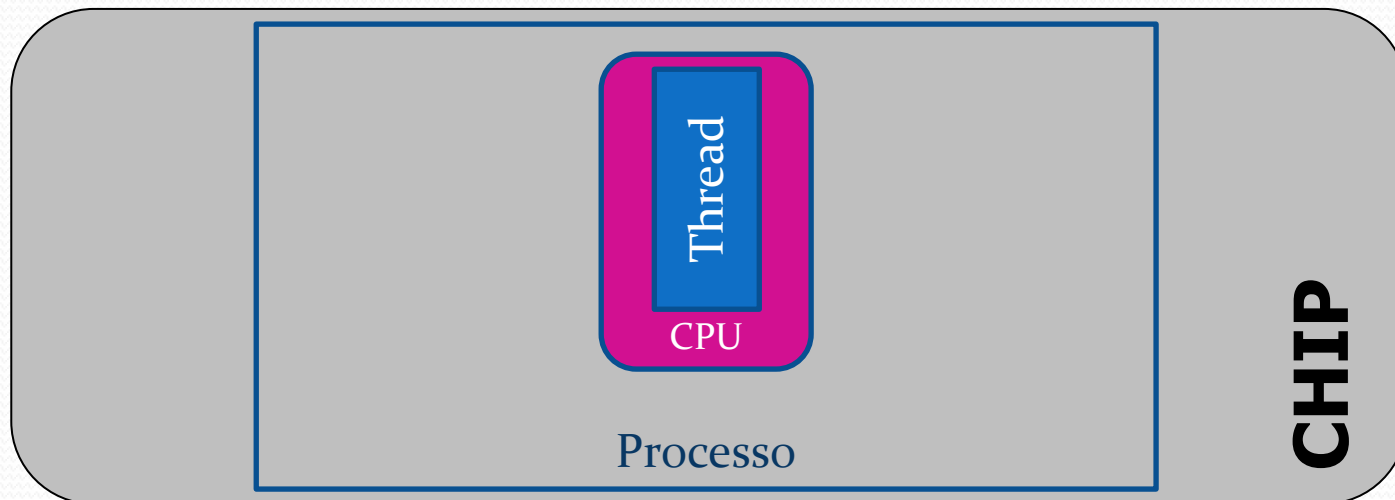
Il thread è quindi un flusso di istruzioni indipendente da altri che deve essere eseguito sequenzialmente su una CPU



# Threads vs Processi

Un **processo** si definisce banalmente come  
“un programma in esecuzione”.

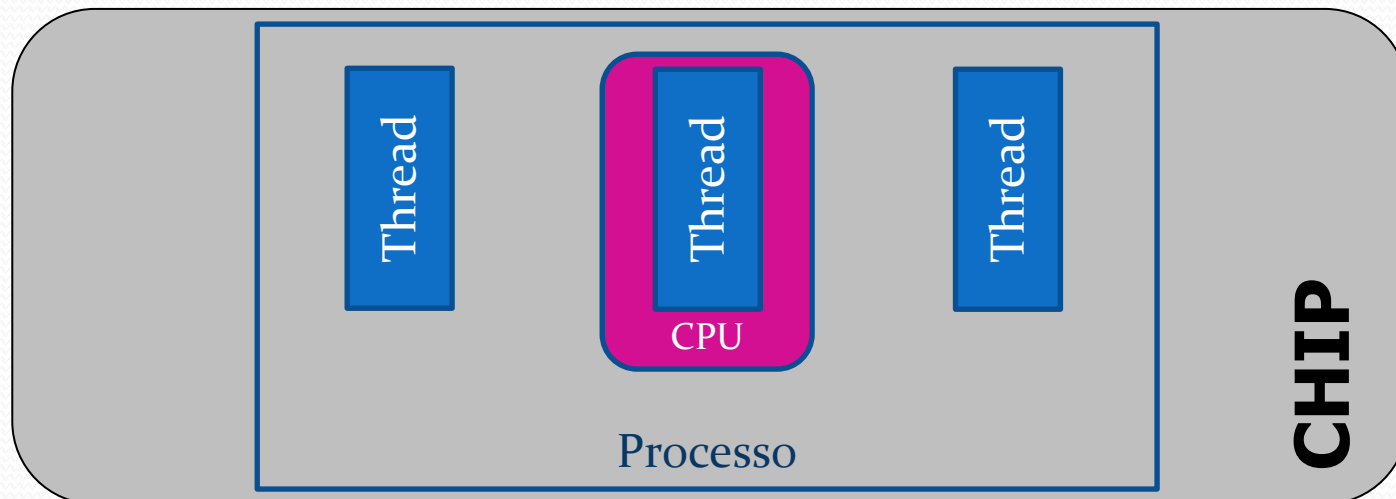
Un **processo** è costituito da almeno un thread ...



# Threads vs Processi

Un **processo** si definisce banalmente come  
“un programma in esecuzione”.

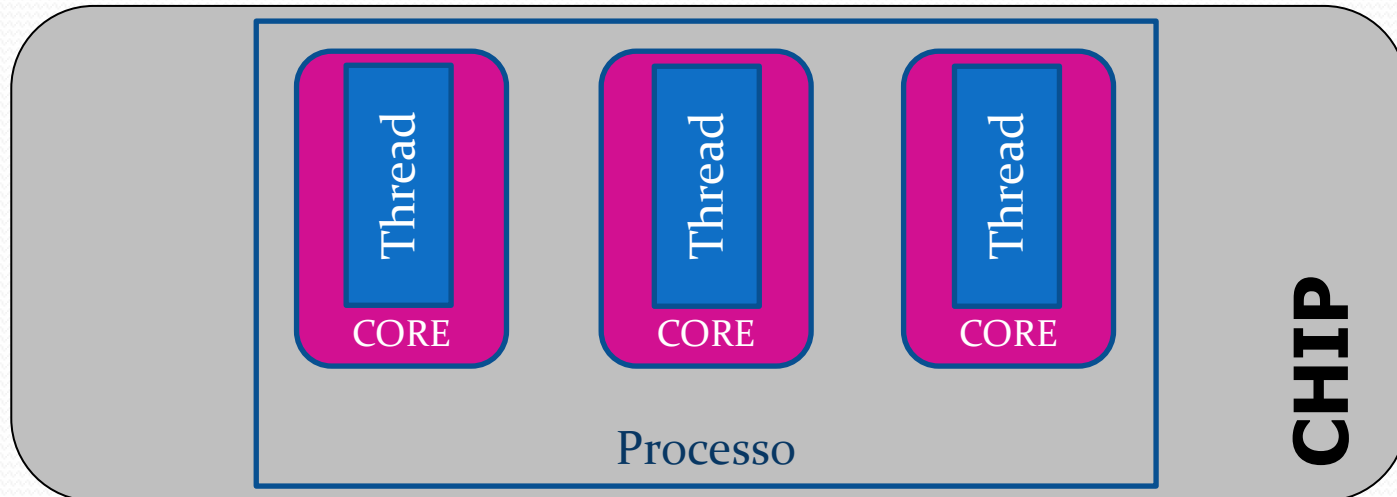
Un **processo** è costituito da almeno un thread ...  
... ma può contenerne più di uno





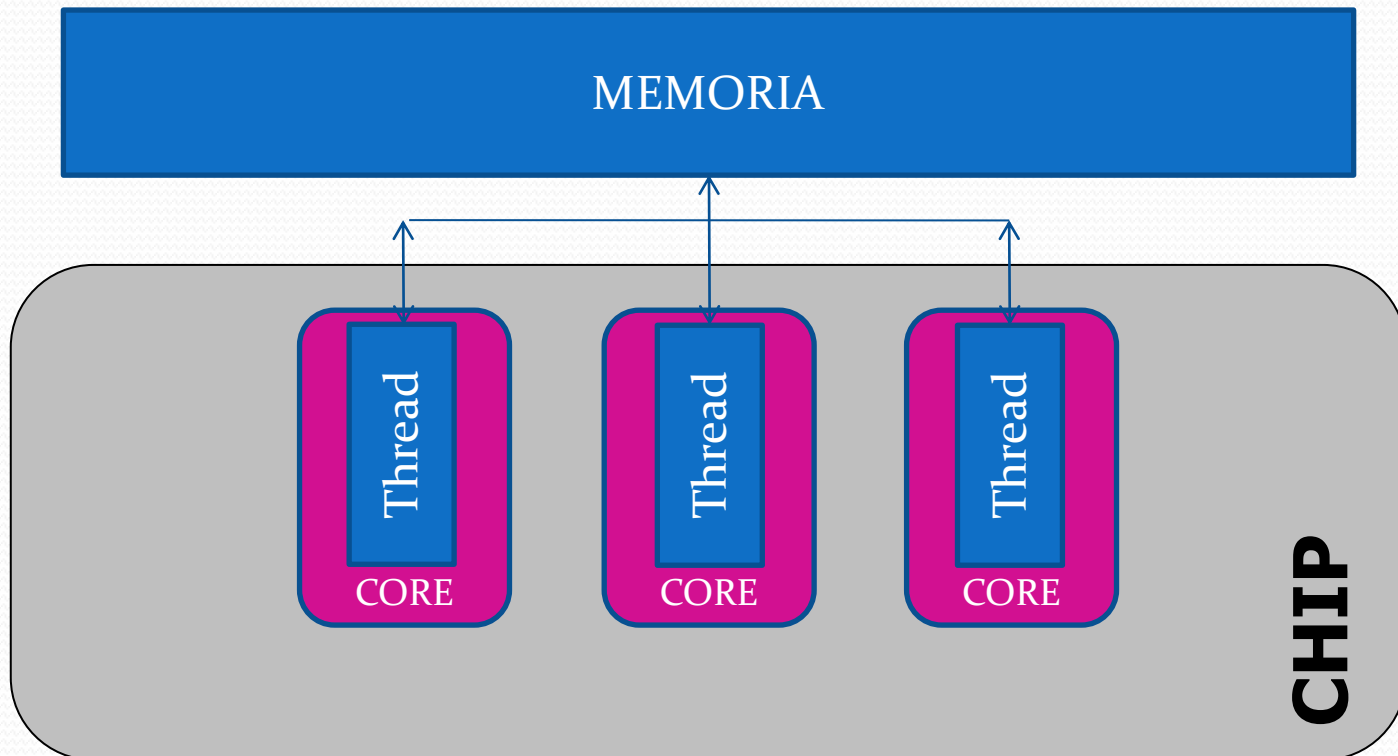
# Threads vs Processi

Thread diversi possono essere eseguiti indipendentemente su cpu/core diversi



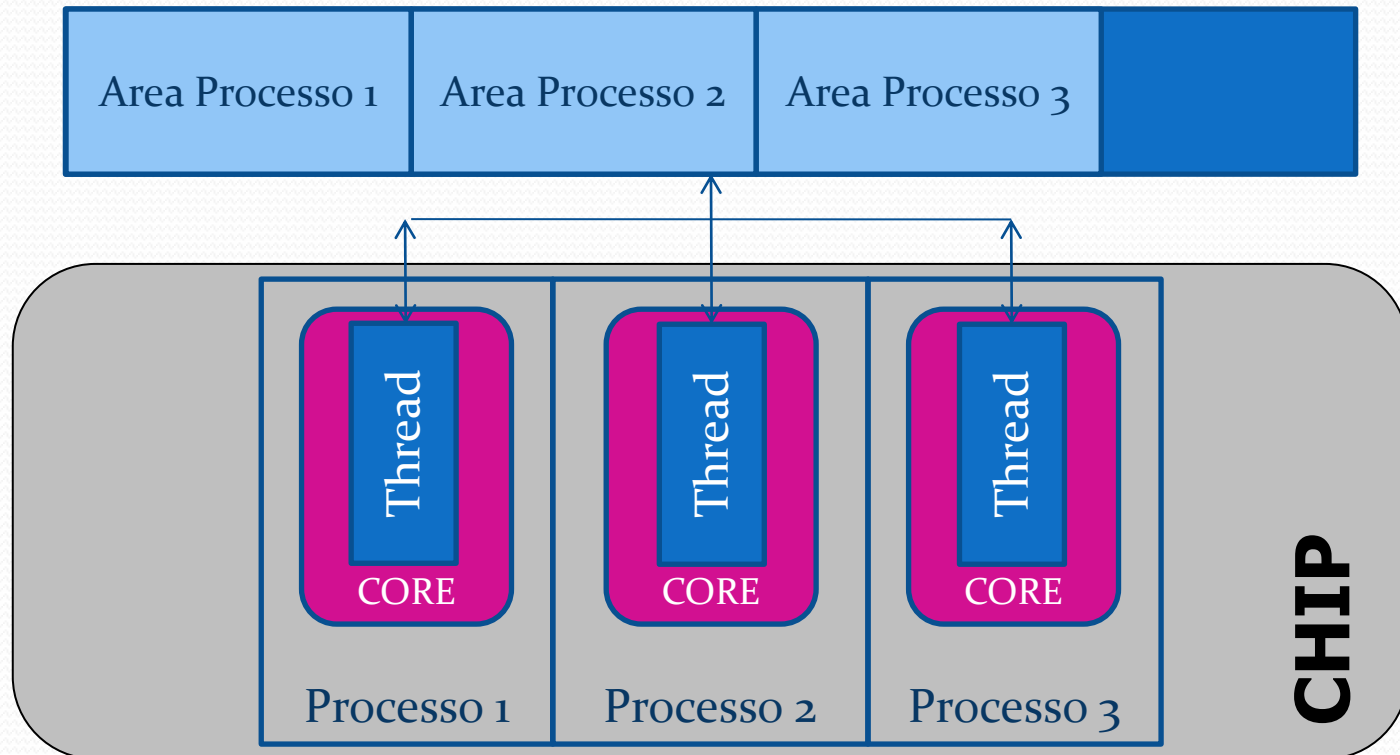
# Threads vs Processi

Anche se i core sono collegati fisicamente alla stessa memoria ...



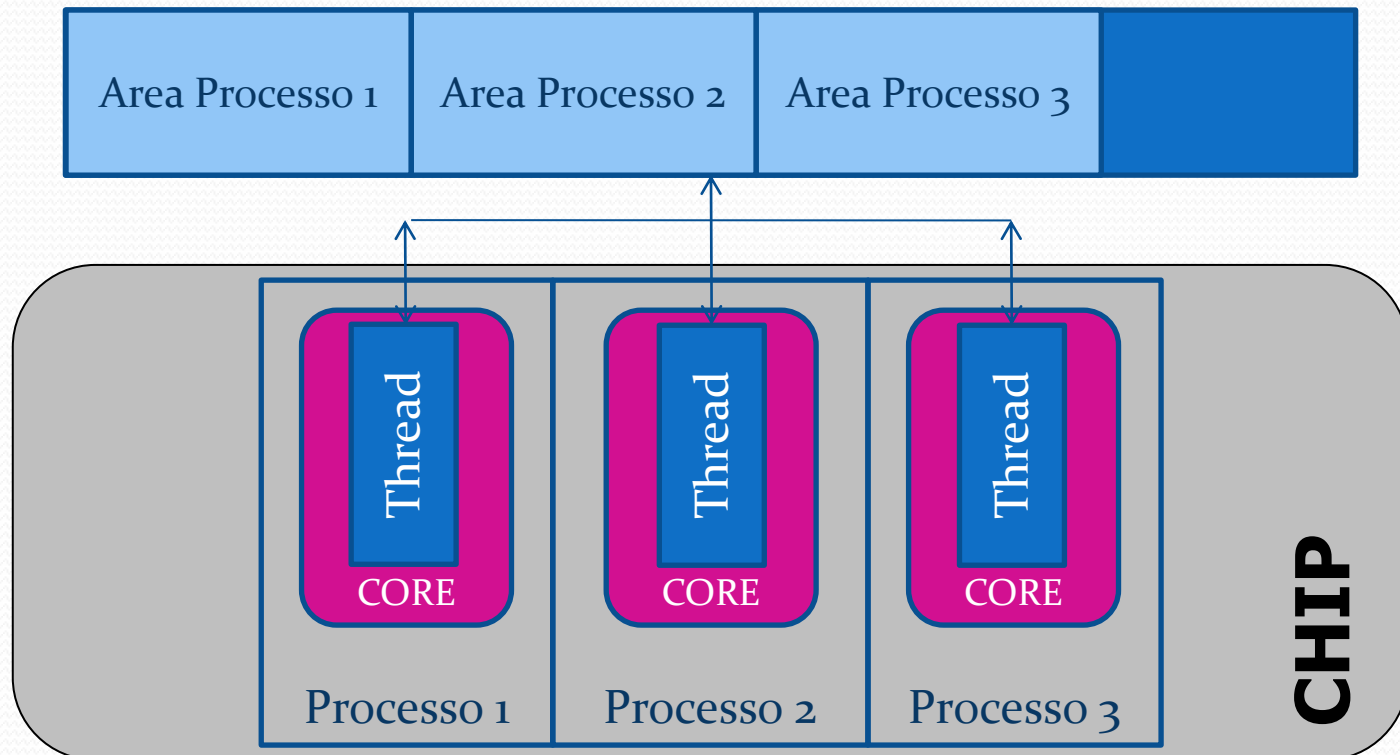
# Threads vs Processi

... processi diversi **non** condividono le stesse aree



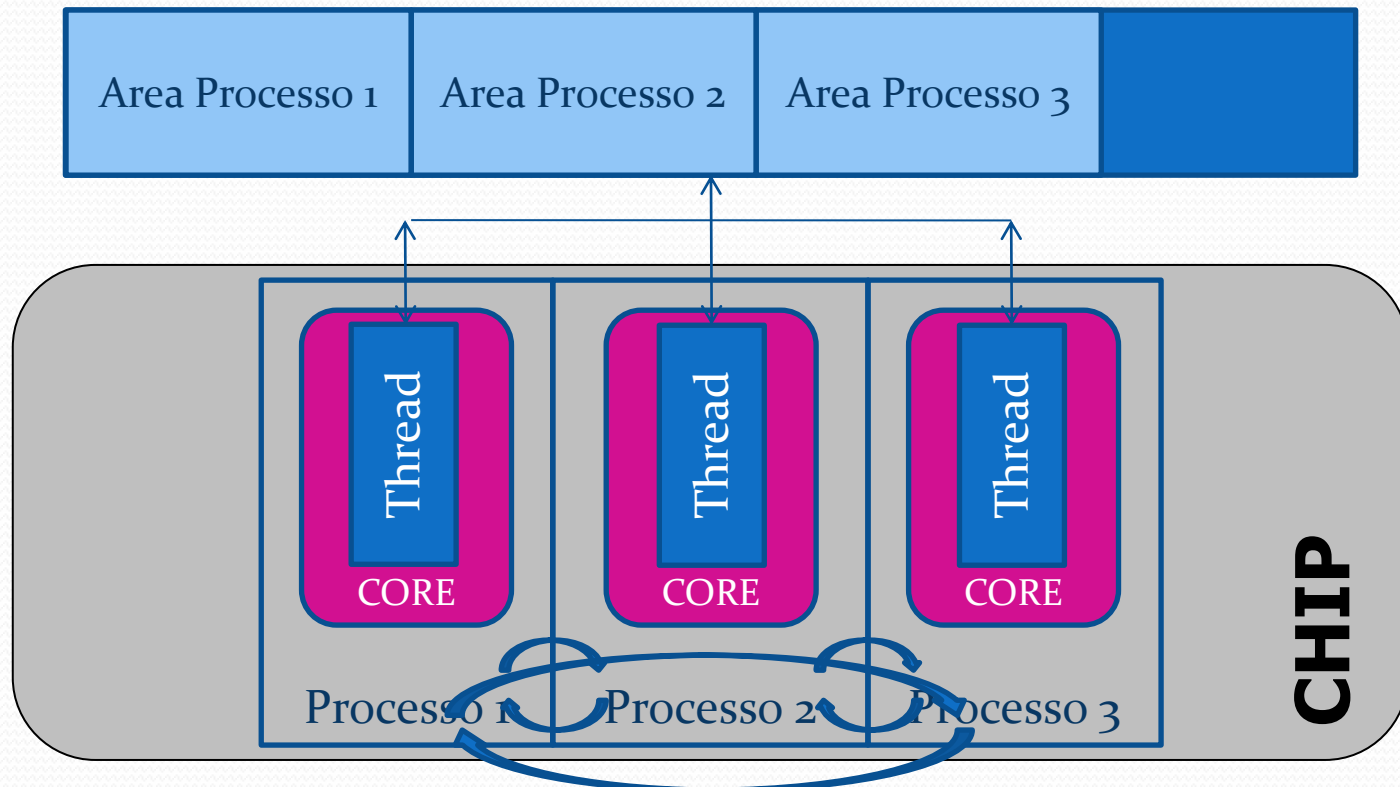
# Threads vs Processi

Es. : perché usino la stessa variabile, questa deve essere allocata **per ognuno**



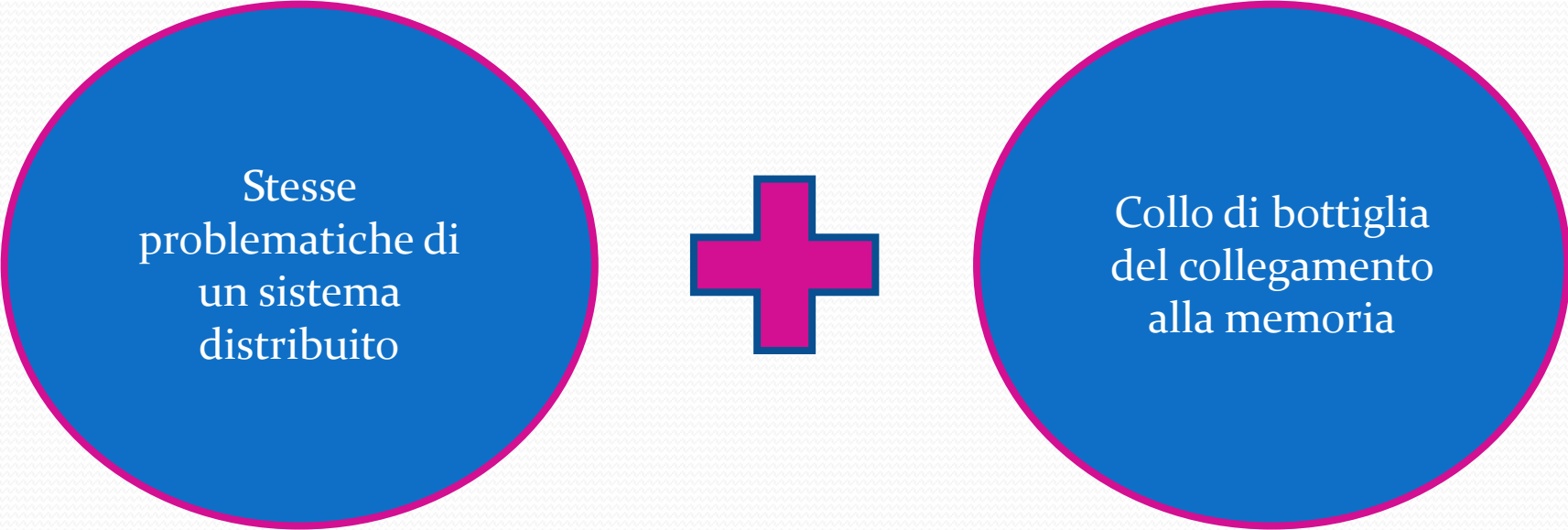
# Threads vs Processi

Perché lavorino insieme bisogna distribuire tra loro i dati e gestire una **comunicazione** esplicita



# Threads vs Processi

Perché lavorino insieme bisogna distribuire tra loro i dati e gestire una **comunicazione** esplicita



Stesse  
problematiche di  
un sistema  
distribuito

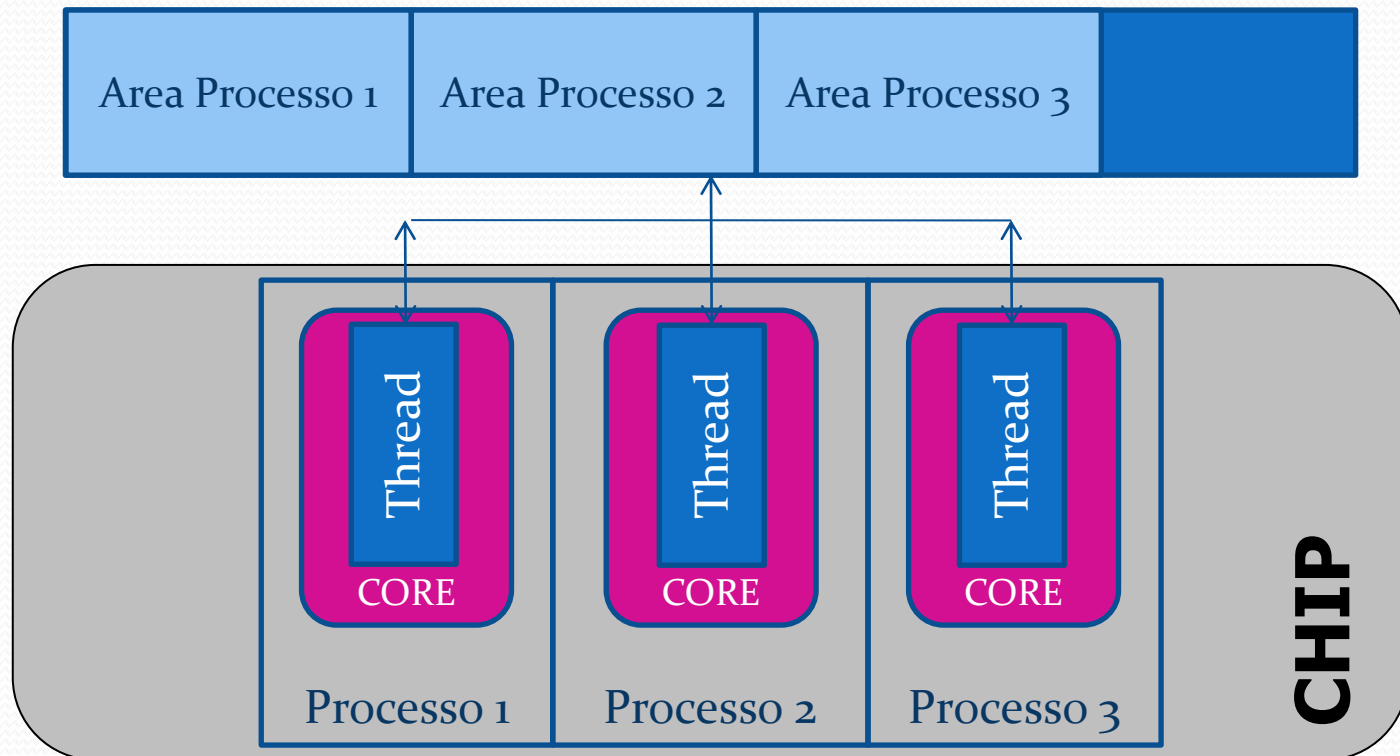


Collo di bottiglia  
del collegamento  
alla memoria

# Threads vs Processi

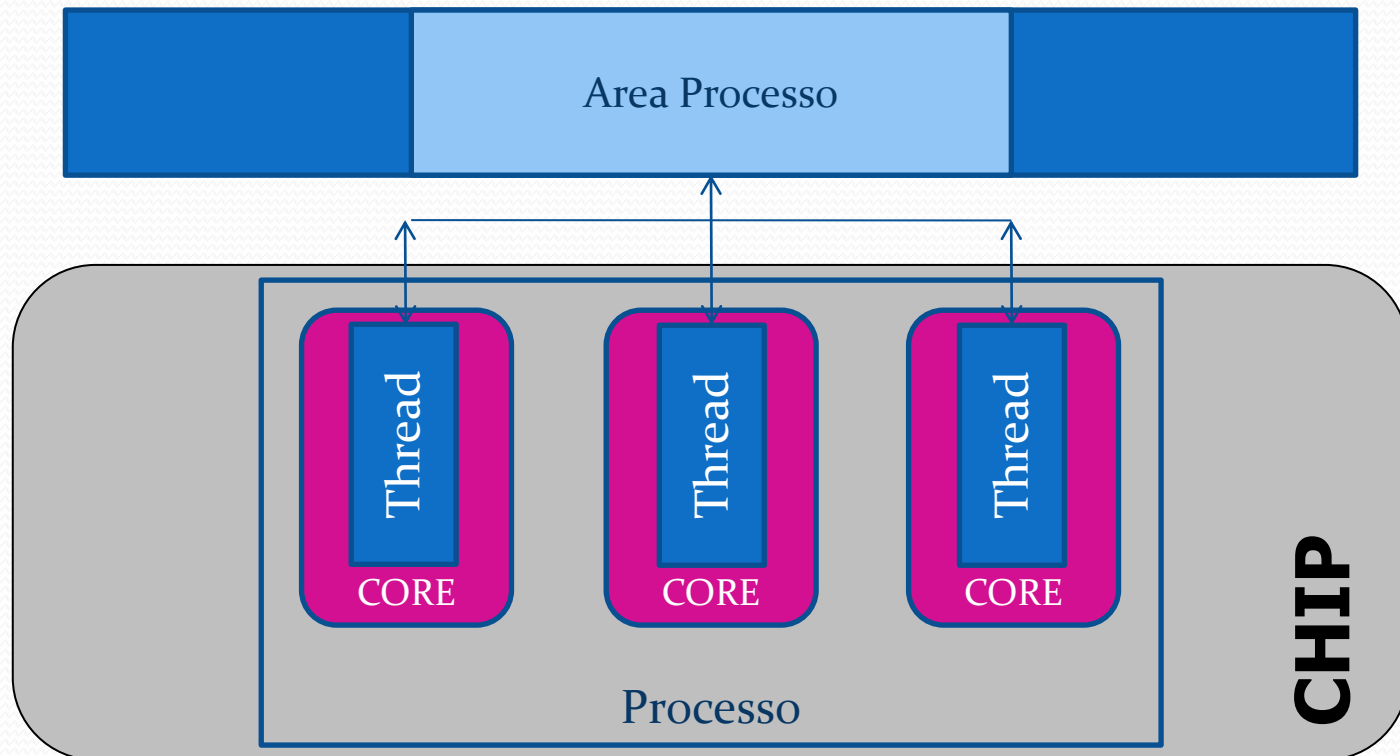
Vantaggio: protezione dei dati

Svantaggio: gestione pesante e poco efficiente



# Threads vs Processi

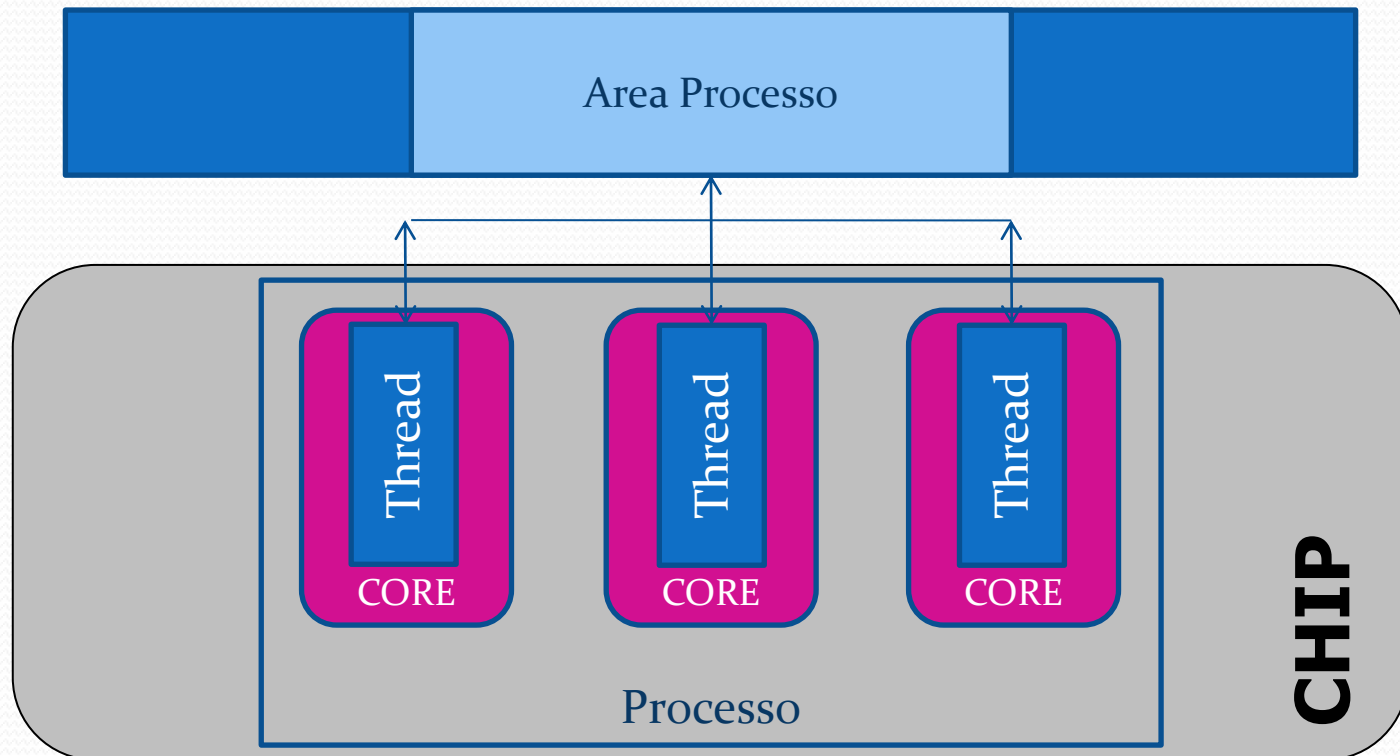
I thread di uno stesso processo **condividono** la stessa area di memoria





# Threads vs Processi

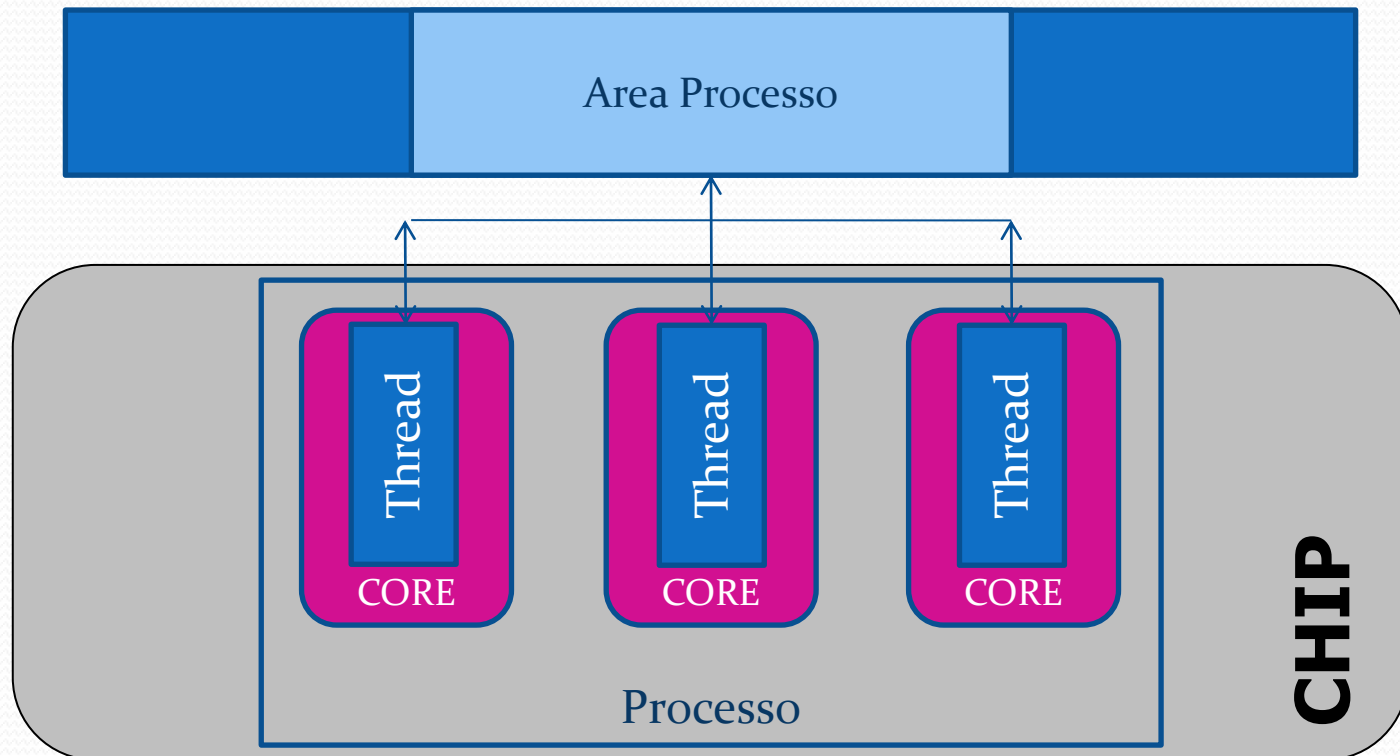
Lavorano insieme in maniera naturale. Si può pensare di dividere il lavoro tra thread di uno stesso processo, invece che tra processi diversi.



# Threads vs Processi

Vantaggio: leggerezza ed efficienza

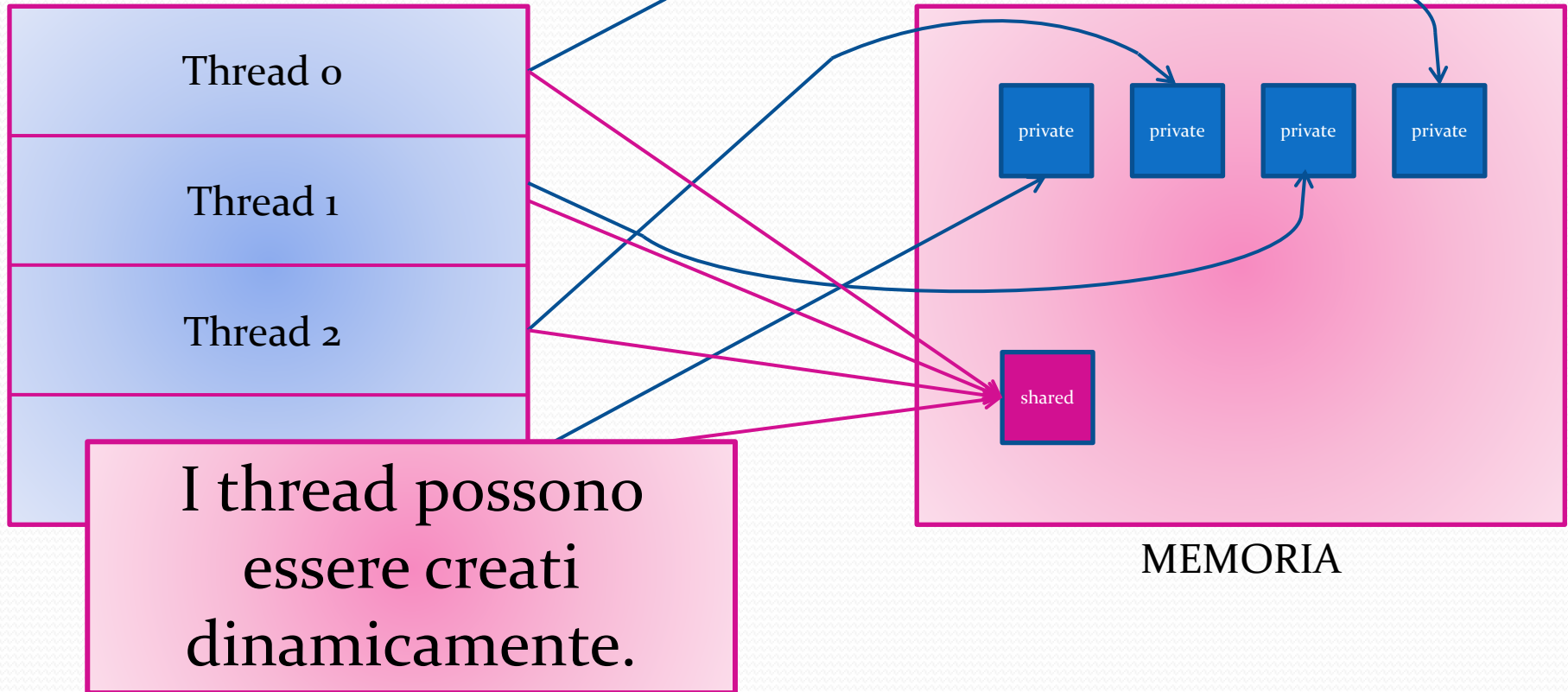
Svantaggio: i dati non sono protetti, è necessaria **sincronizzazione**



# Threads

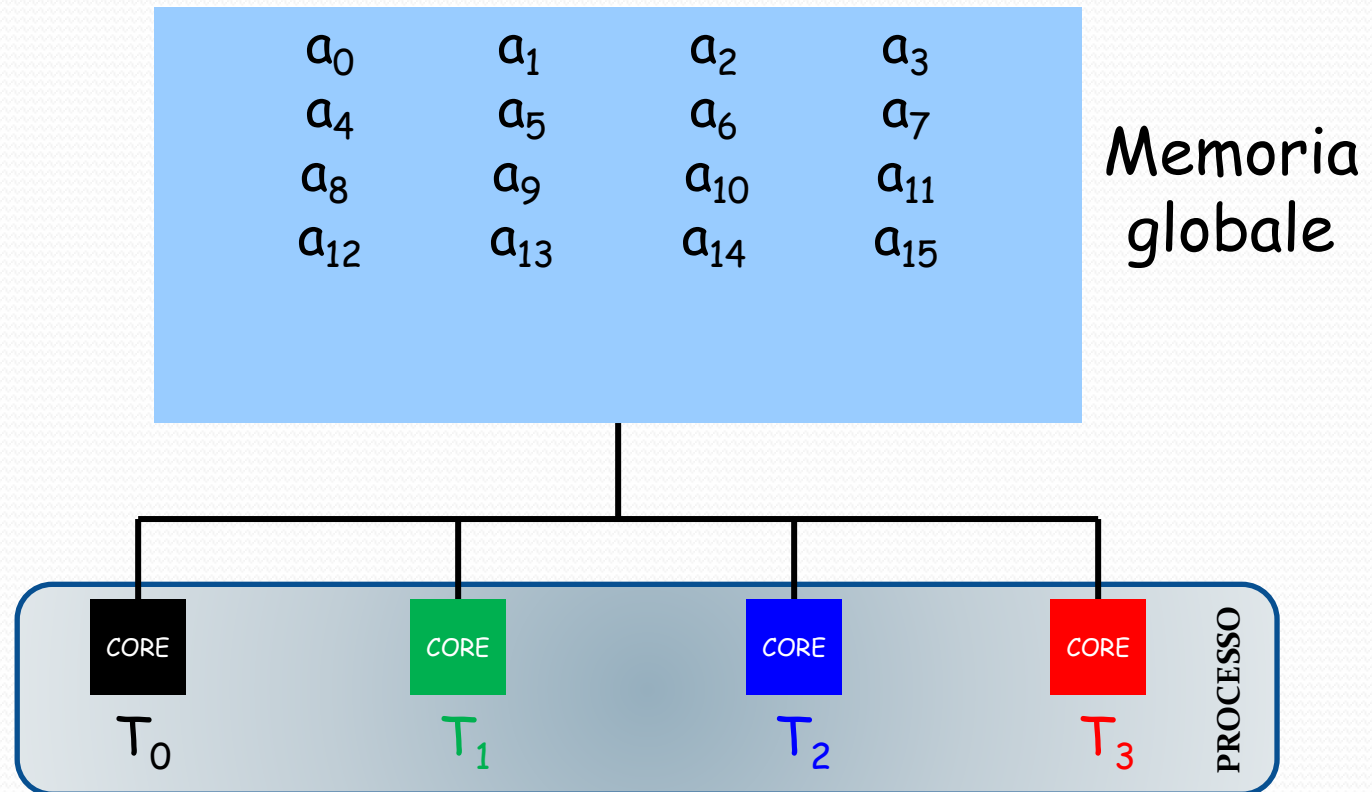
I thread vengono coordinati attraverso la **sincronizzazione** degli accessi alle variabili condivise.

Processo multi-thread



# Esempio: Somma

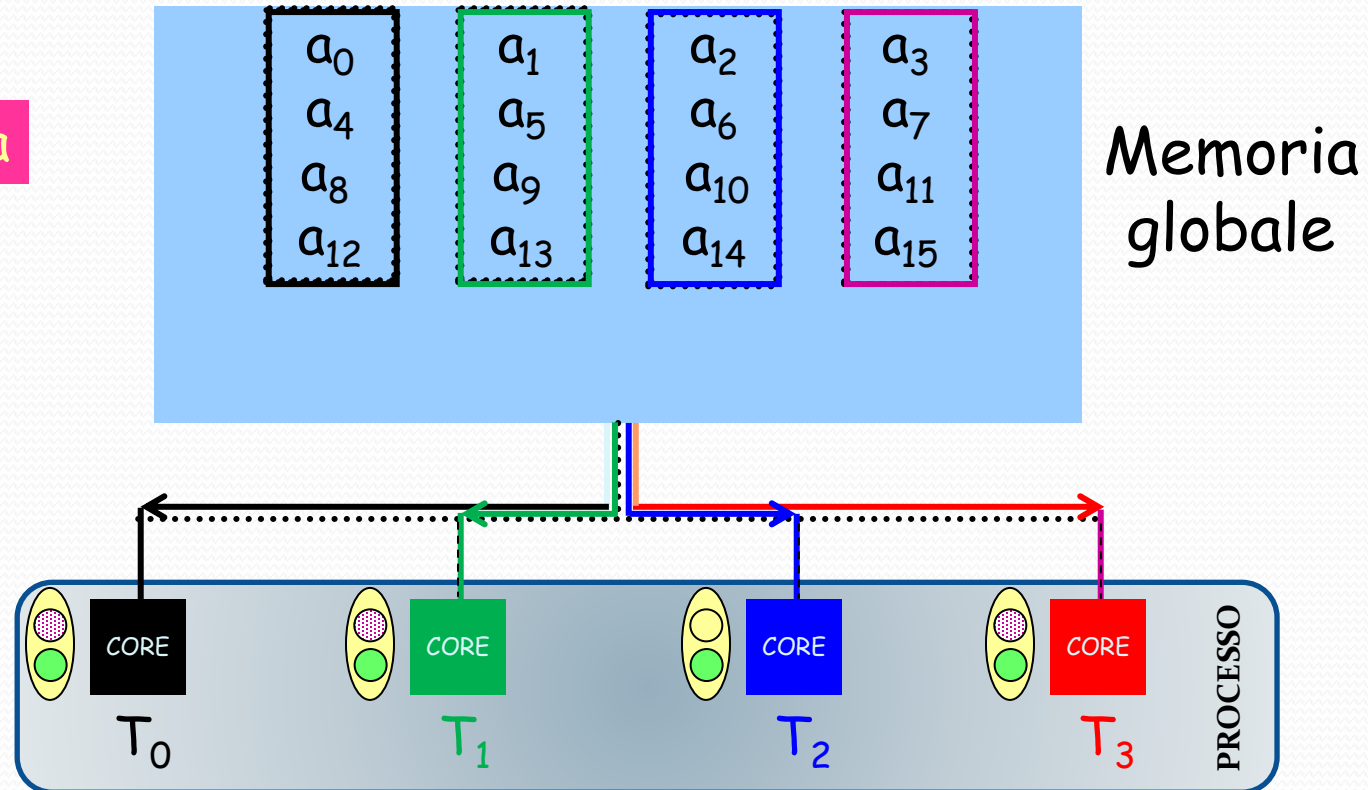
- Esempio:  $N=16$ ,  $p=4$



# Esempio: Somma

- Esempio:  $N=16$ ,  $p=4$

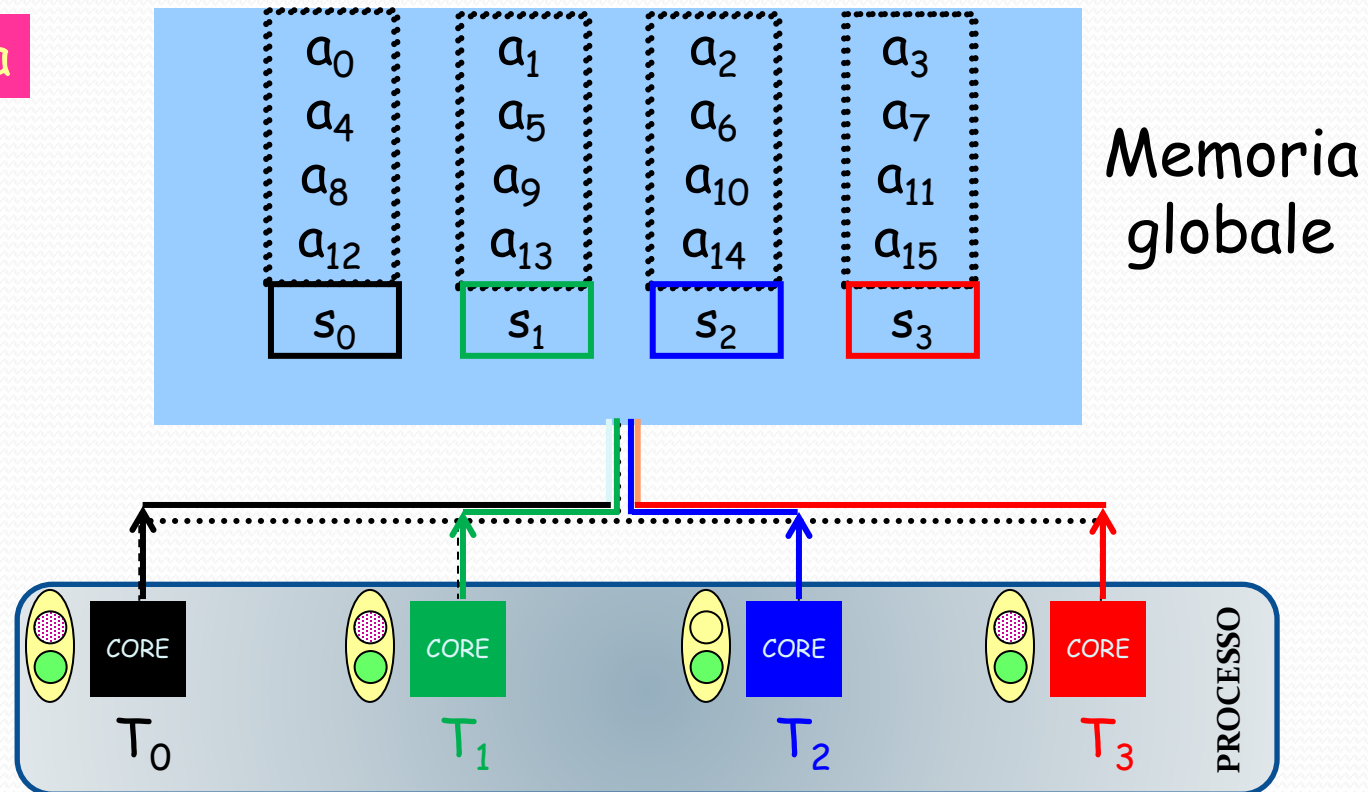
Lettura



# Esempio: Somma

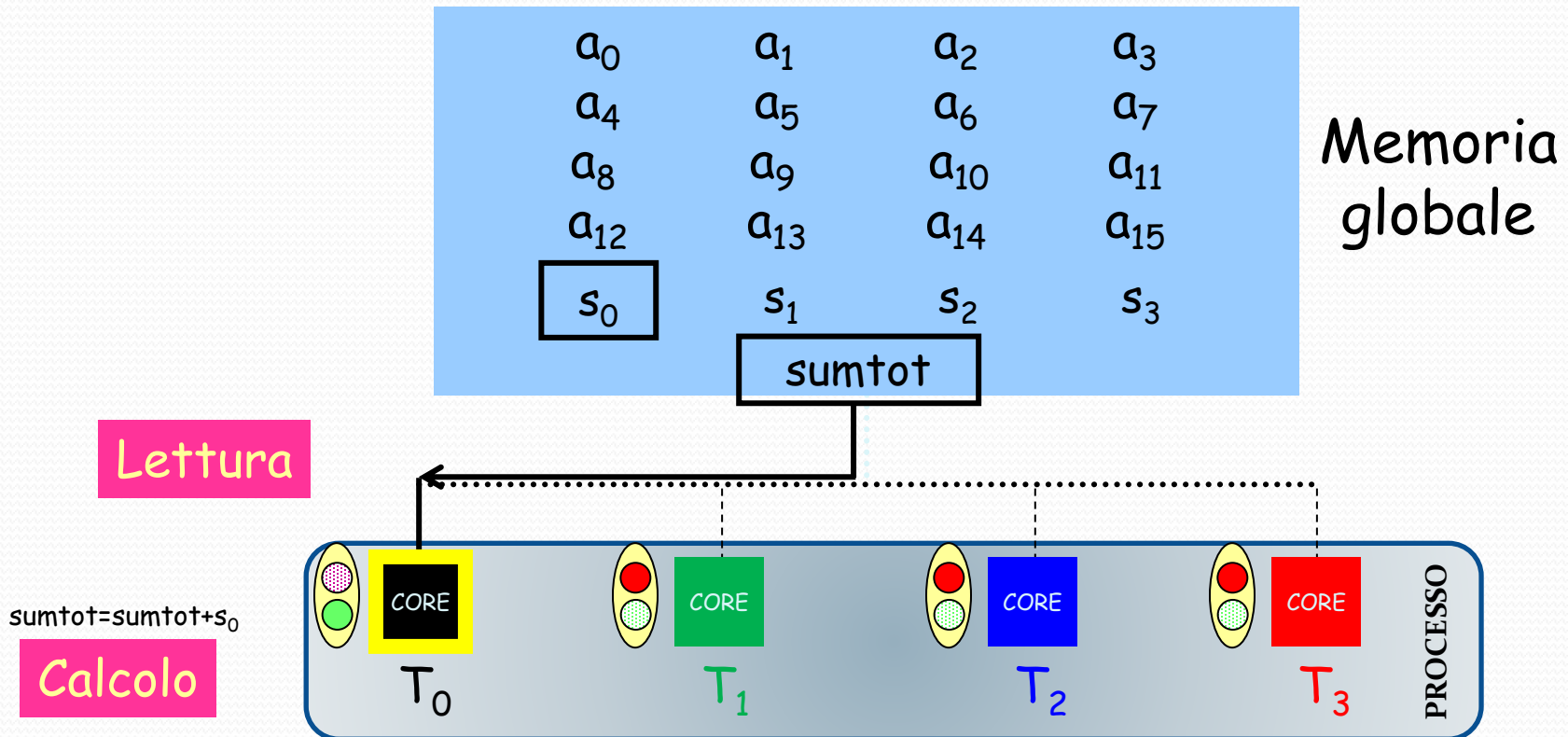
- Esempio:  $N=16$ ,  $p=4$

Scrittura



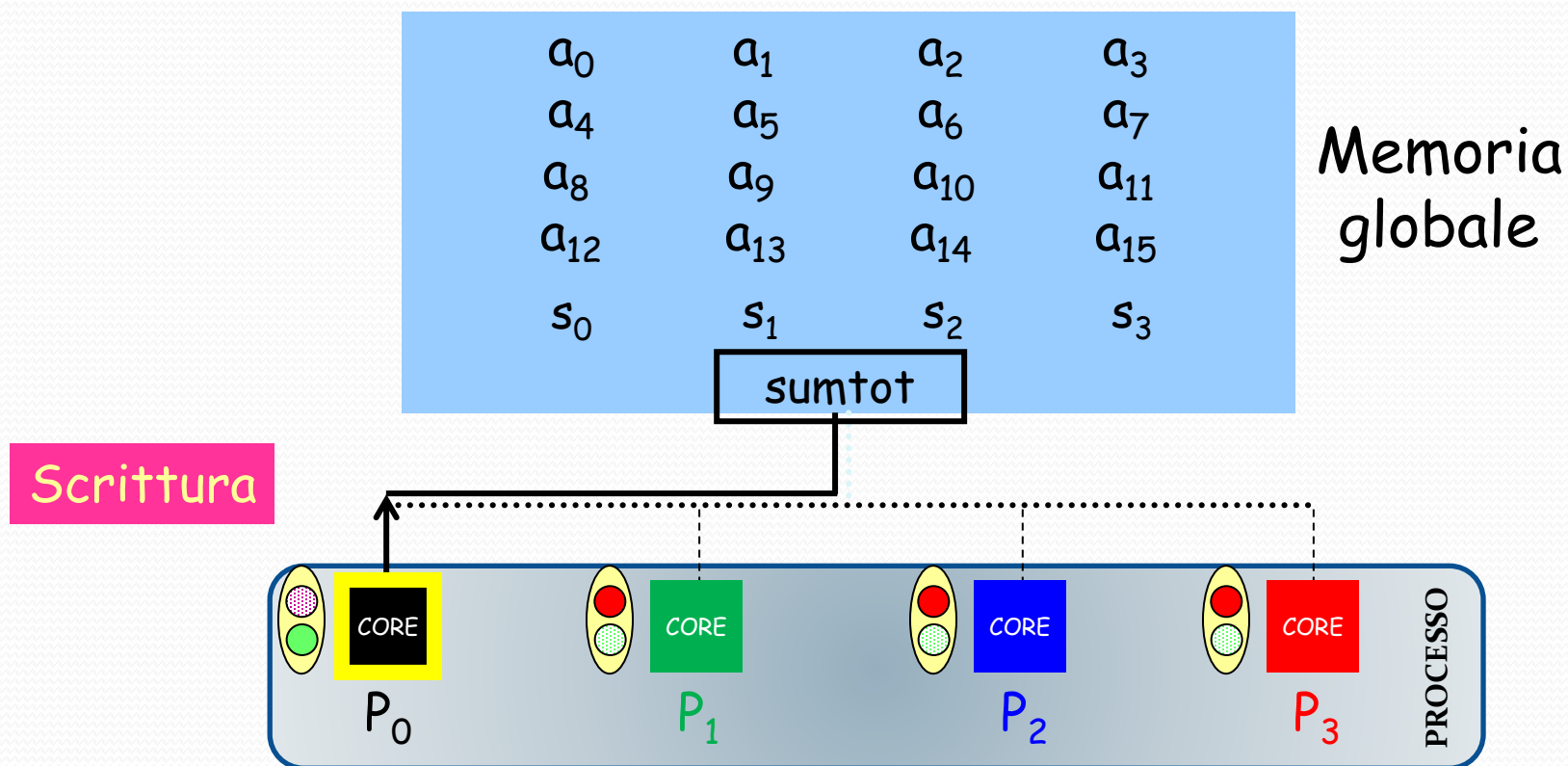
# Esempio: Somma

- Esempio:  $N=16$ ,  $p=4$



# Esempio: Somma

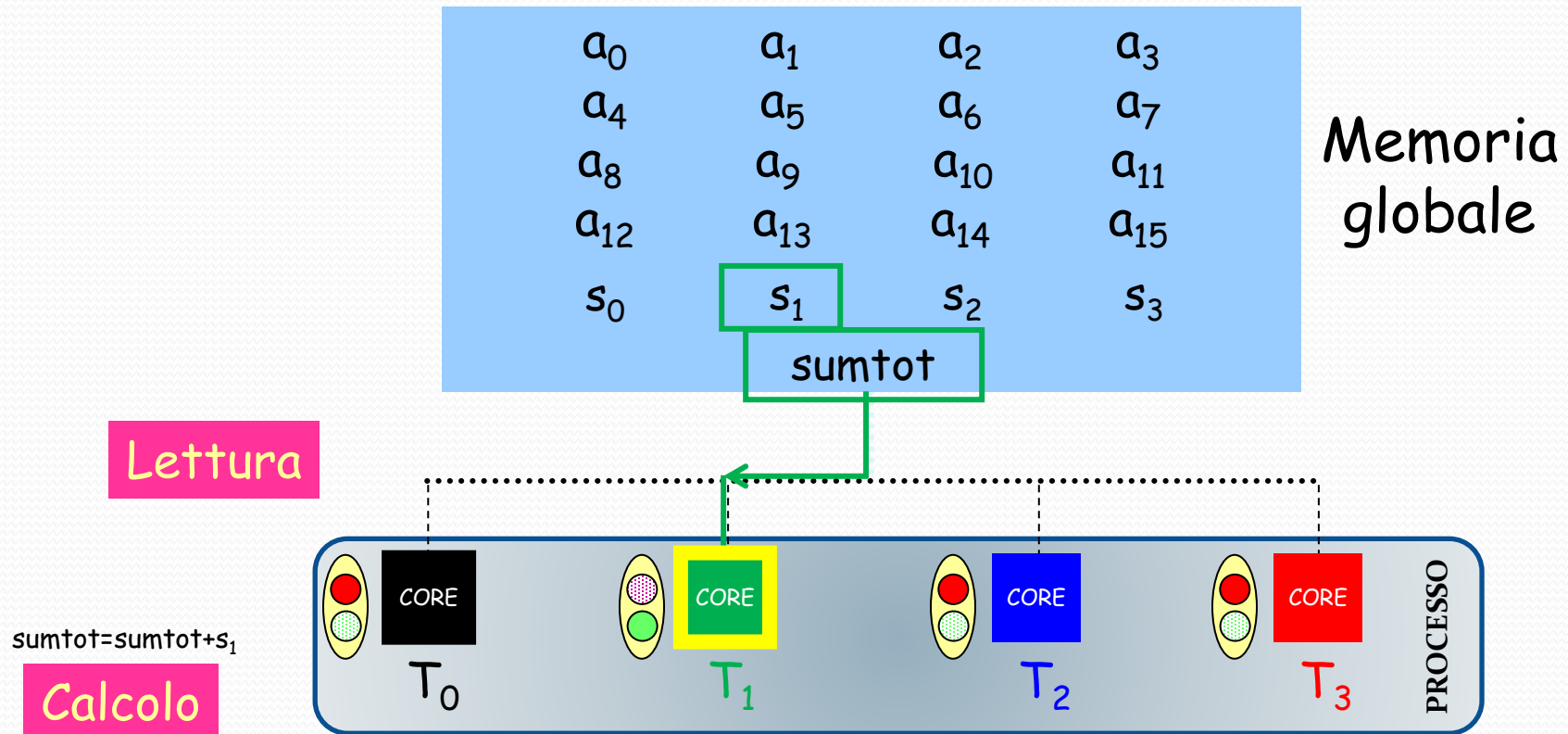
- Esempio:  $N=16$ ,  $p=4$





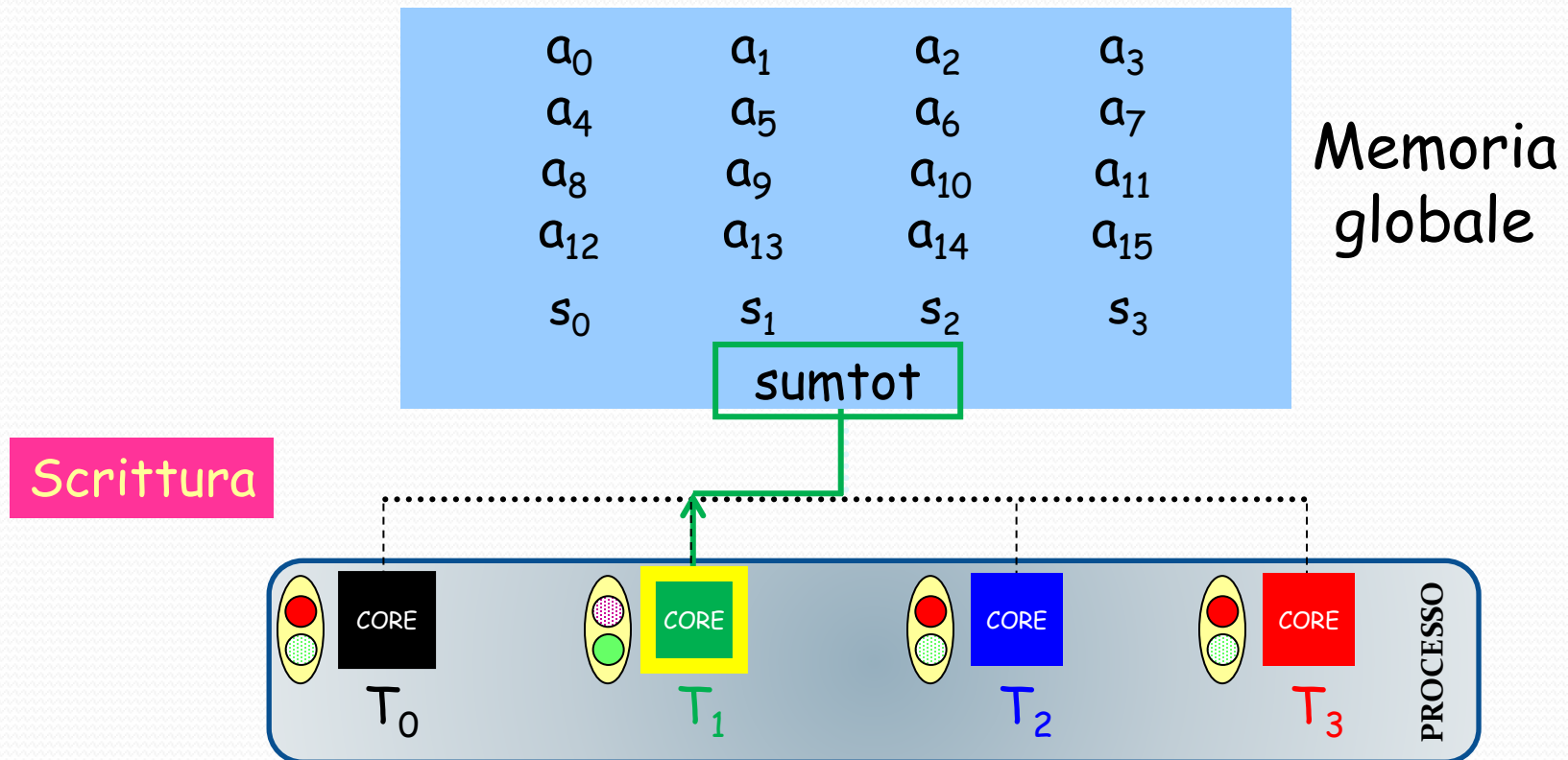
# Esempio: Somma

- Esempio:  $N=16$ ,  $p=4$



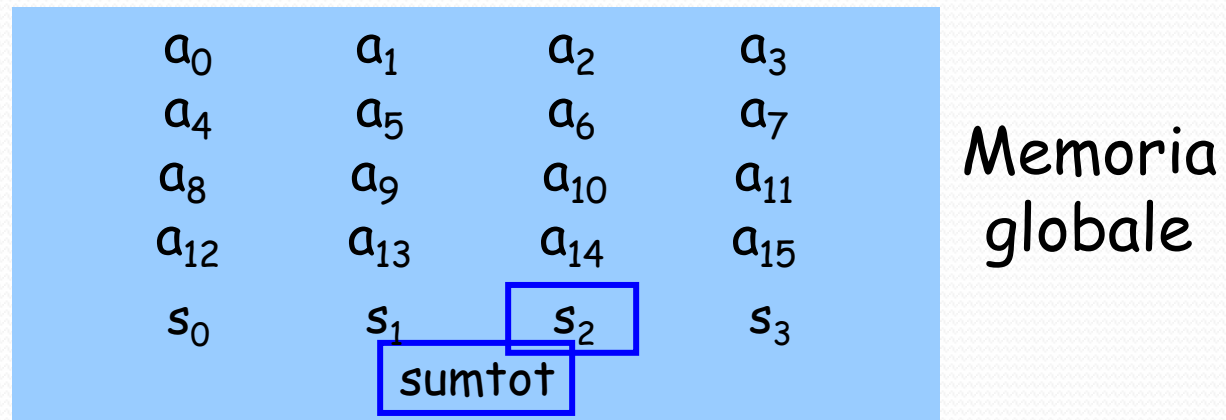
# Esempio: Somma

- Esempio:  $N=16$ ,  $p=4$

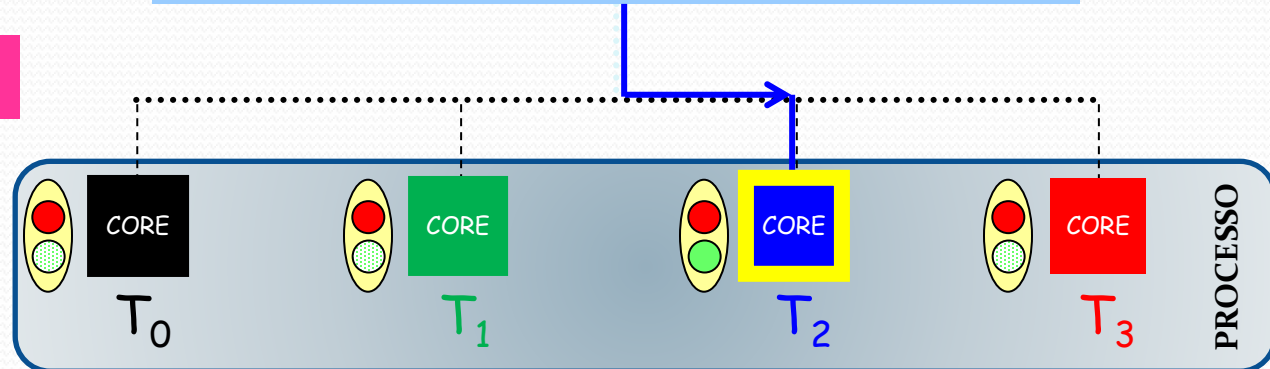


# Esempio: Somma

- Esempio:  $N=16$ ,  $p=4$



Lettura

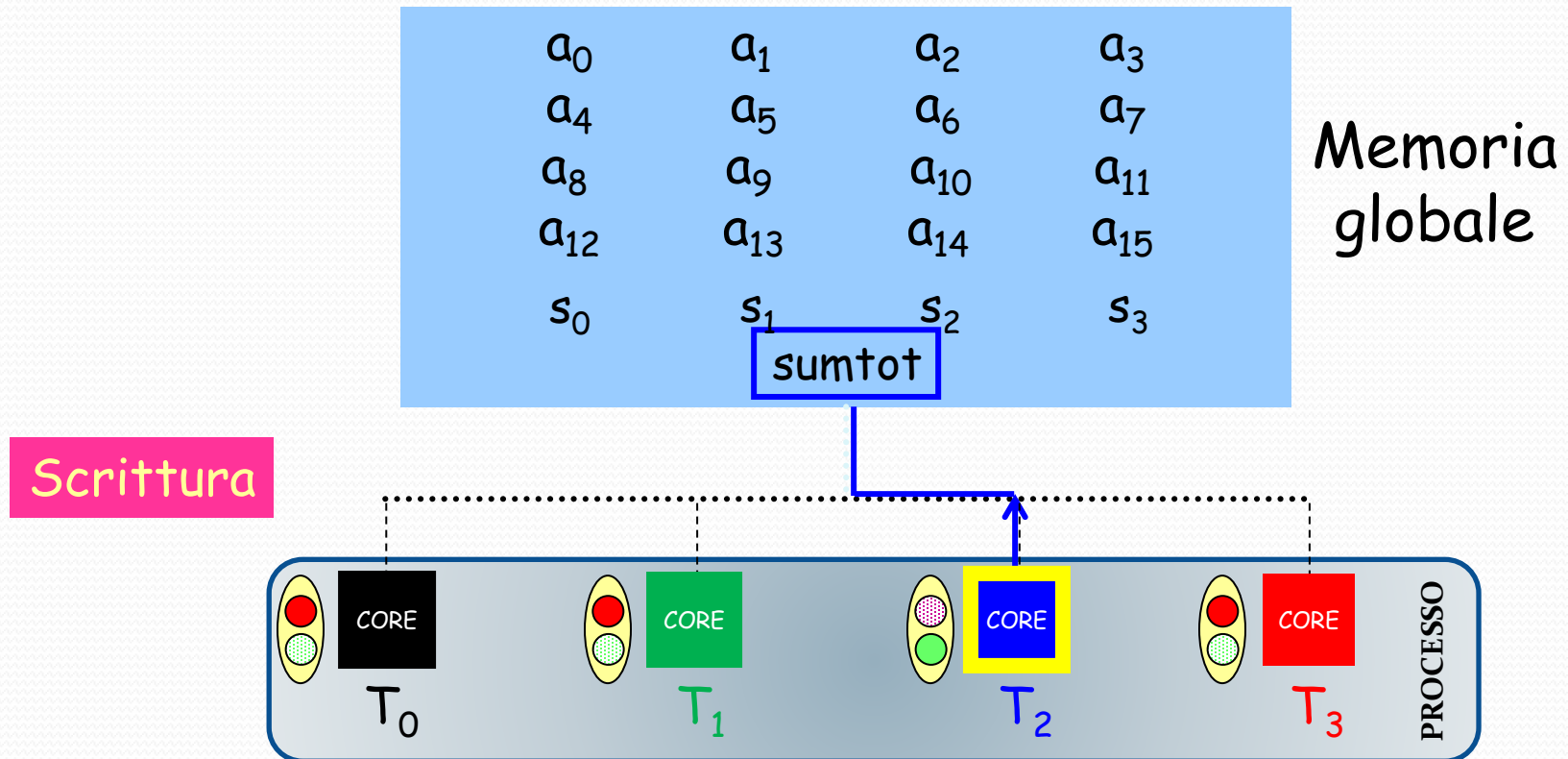


$sumtot = sumtot + s_2$

Calcolo

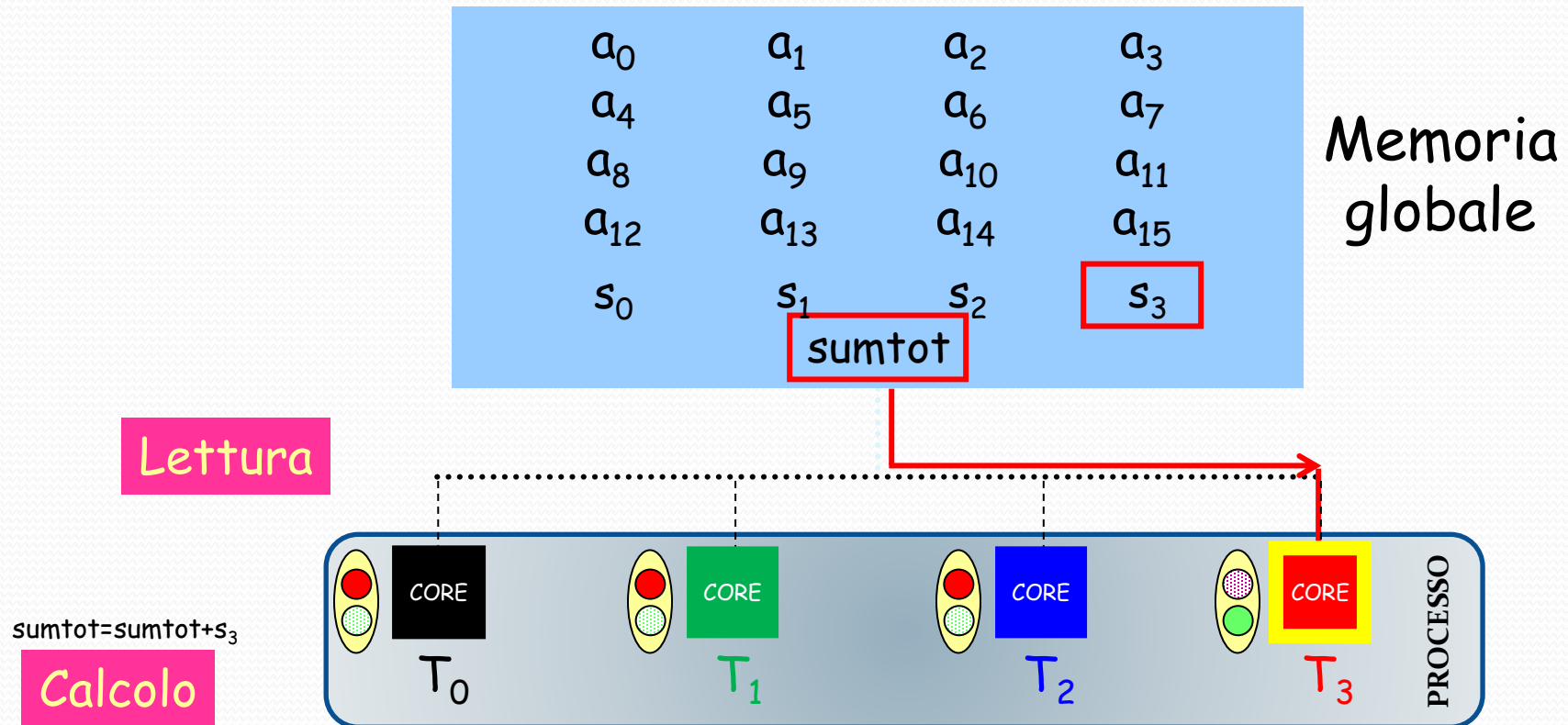
# Esempio: Somma

- Esempio:  $N=16$ ,  $p=4$



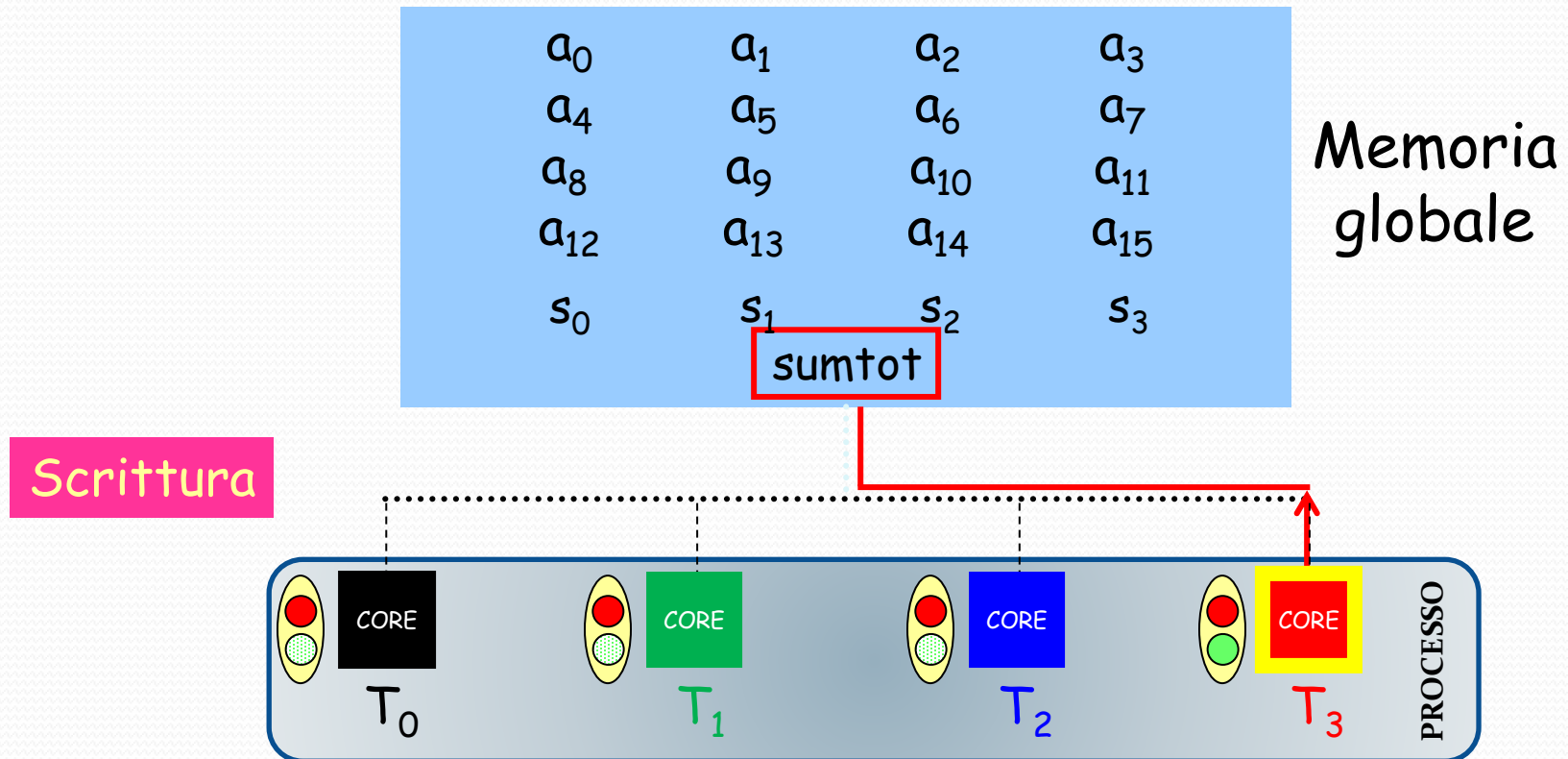
# Esempio: Somma

- Esempio:  $N=16$ ,  $p=4$



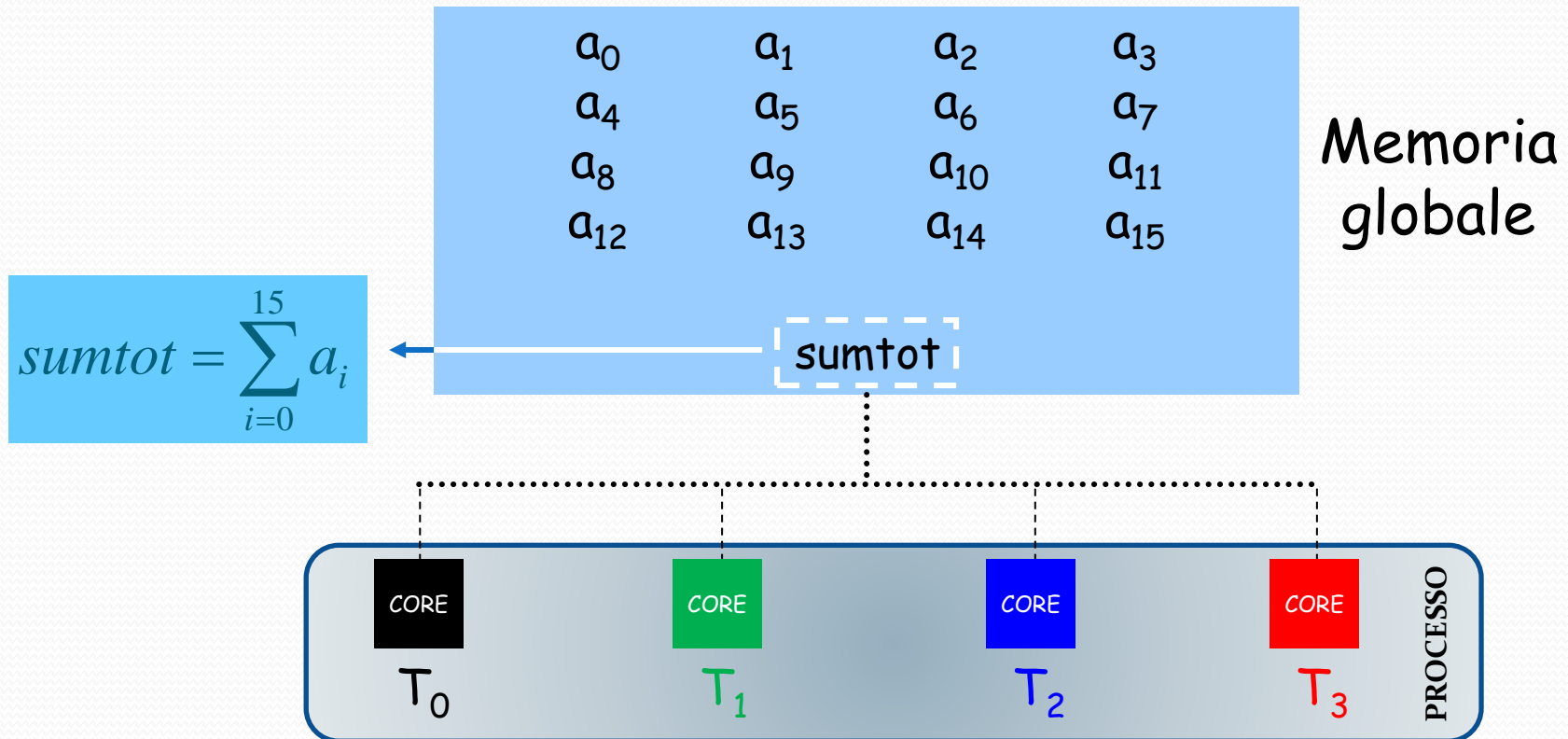
# Esempio: Somma

- Esempio:  $N=16$ ,  $p=4$



# Esempio: Somma

- Esempio:  $N=16$ ,  $p=4$



# Esempio: Somma - Algoritmo

Algoritmo  
sequenziale  
(1 thread)

n elementi da sommare  $a_i$

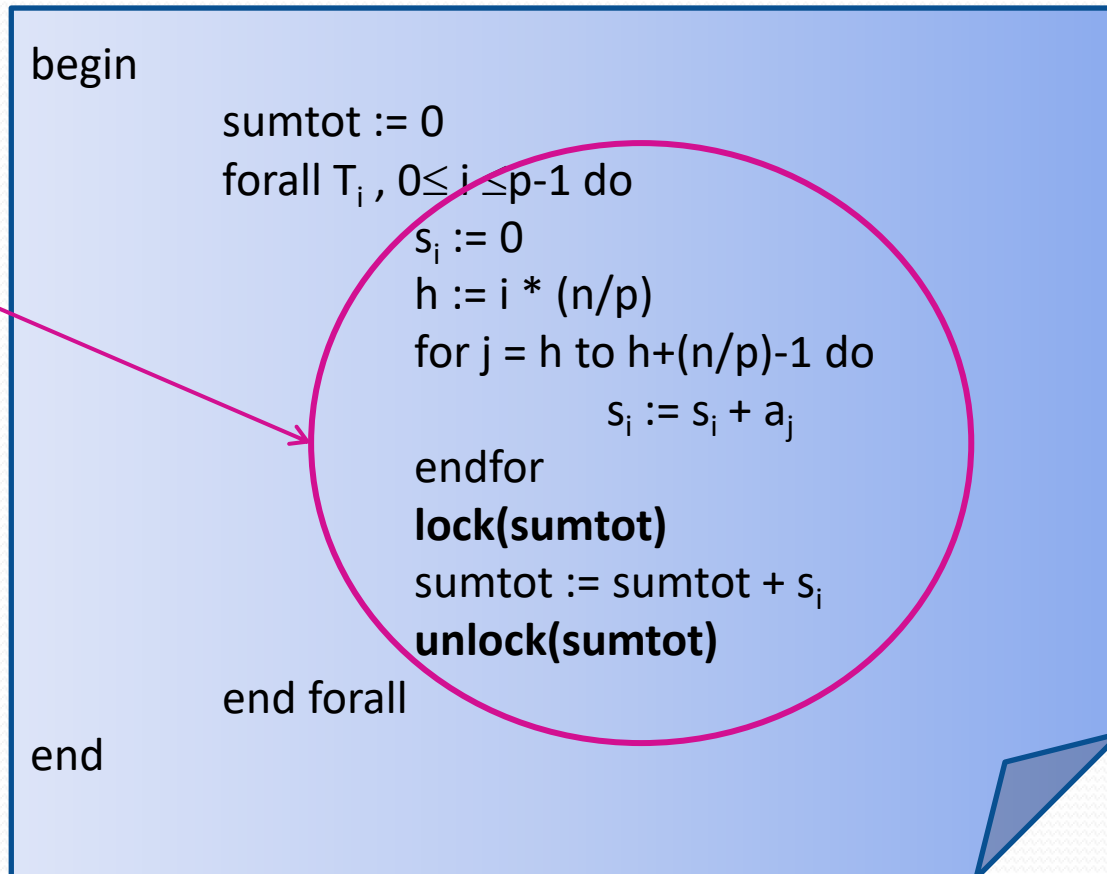
```
begin
    ...
    sumtot := 0
    for i = 0 to n do
        sumtot:= sumtot+  $a_i$ 
    endfor
    ...
end
```



# Esempio: Somma - Algoritmo ( $n=kp$ )

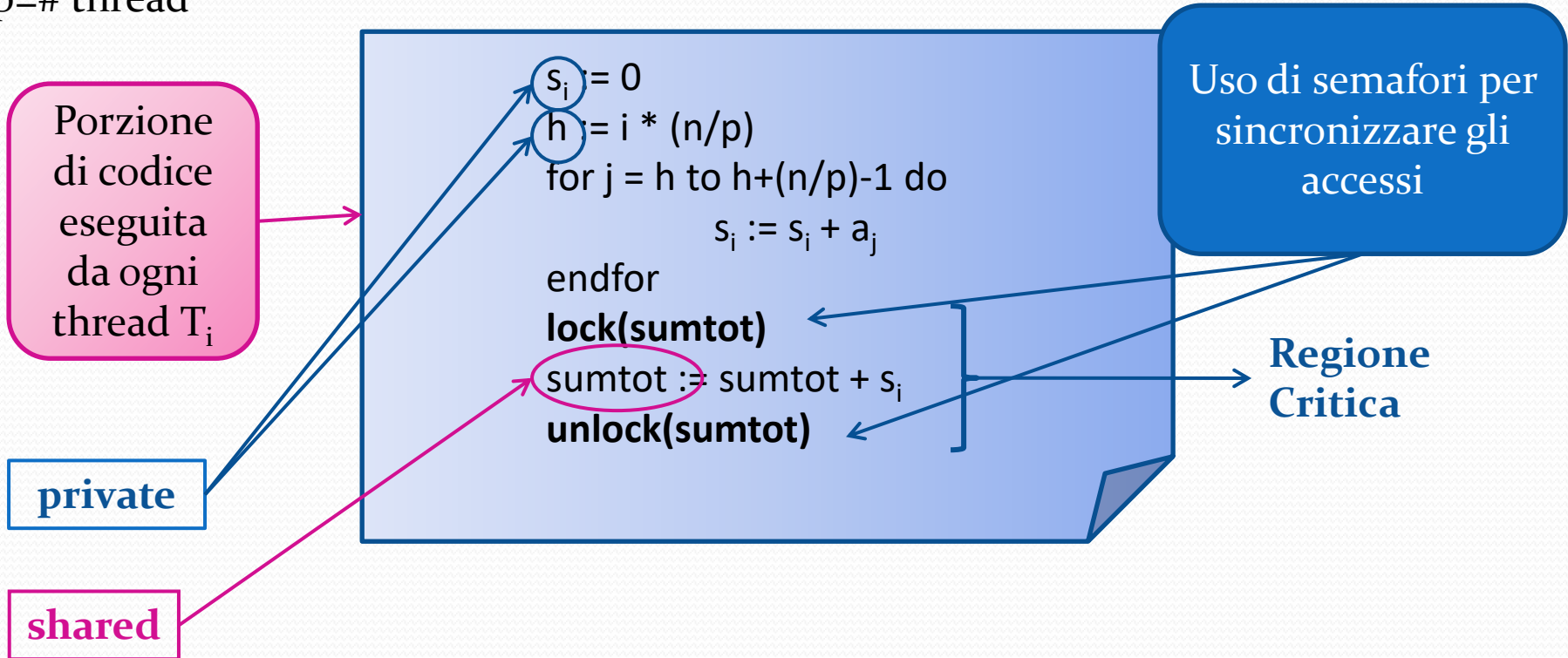
$p = \# \text{ thread}$

Porzione  
di codice  
eseguita  
da ogni  
thread  $T_i$



# Esempio: Somma - Algoritmo ( $n=kp$ )

$p = \# \text{ thread}$



# Esempio: Somma - Algoritmo ( $n=kp$ )

Algoritmo parallelo

```
si := 0
h := i * (n/p)
for j = h to h+(n/p)-1 do
    si := si + aj
endfor
lock(sumtot)
sumtot := sumtot + si
unlock(sumtot)
```

Utilizzando OpenMP:

- l'utilizzo di variabili private d'appoggio
  - la divisione del lavoro tra i thread
  - la collezione del risultato in una variabile shared in modo sincronizzato
- possono essere completamente trasparenti**

Algoritmo parallelo con OpenMP

```
...
sumtot = 0;
#pragma omp parallel for reduction(+:sumtot)
for(i=0;i<n;i++)
{
    sumtot=sumtot+a[i];
}
...
```

# Introduzione ad OpenMp

- **Open** specifications for **M**ulti **P**rocessing
- Application Program Interface (API) per gestire il parallelismo shared-memory multi-threaded
- Consente un approccio ad alto livello, user-friendly
- Portabile: Fortran e C/C++, Unix/Linux e Windows

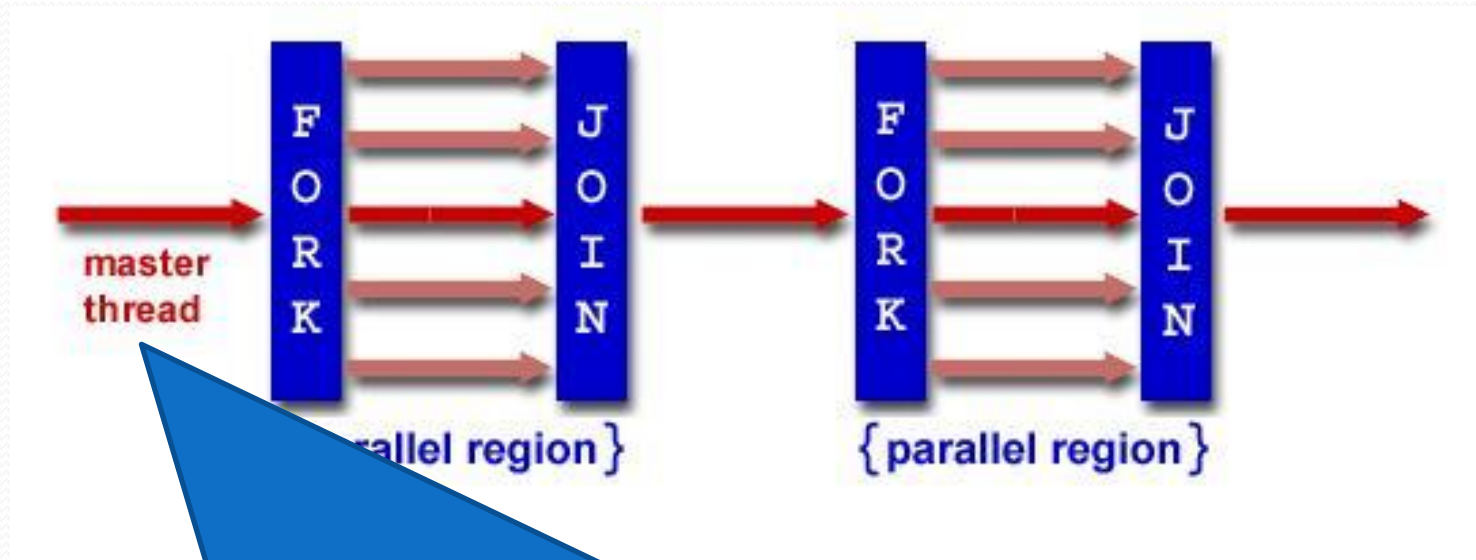
Facile  
trasformare  
un codice  
sequenziale  
in parallelo

Algoritmo parallelo con OpenMP

```
...  
sumtot = 0;  
#pragma omp parallel for reduction (+:sumtot)  
for(i=0;i<n;i++)  
{  
    sumtot=sumtot+a[i];  
}  
...
```

# Introduzione ad OpenMp

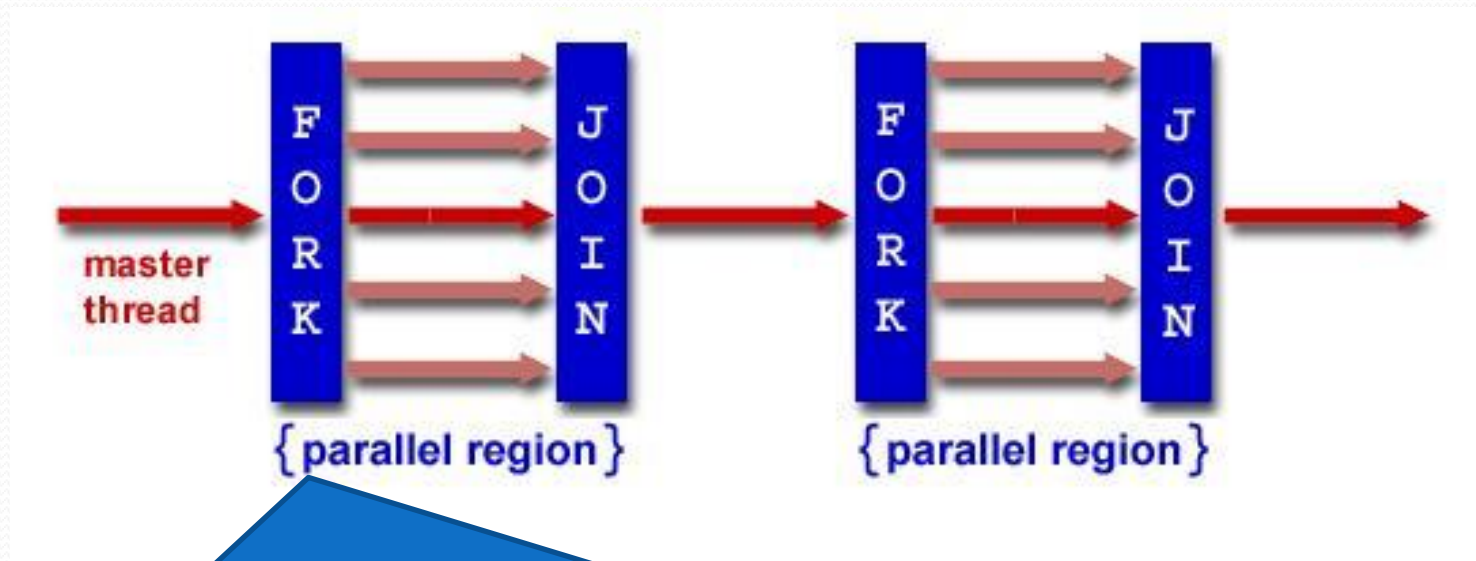
- Il modello d'esecuzione parallela è quello *fork-join*



Tutti i processi cominciano con un solo thread (*master thread*) che esegue in maniera sequenziale

# Introduzione ad OpenMp

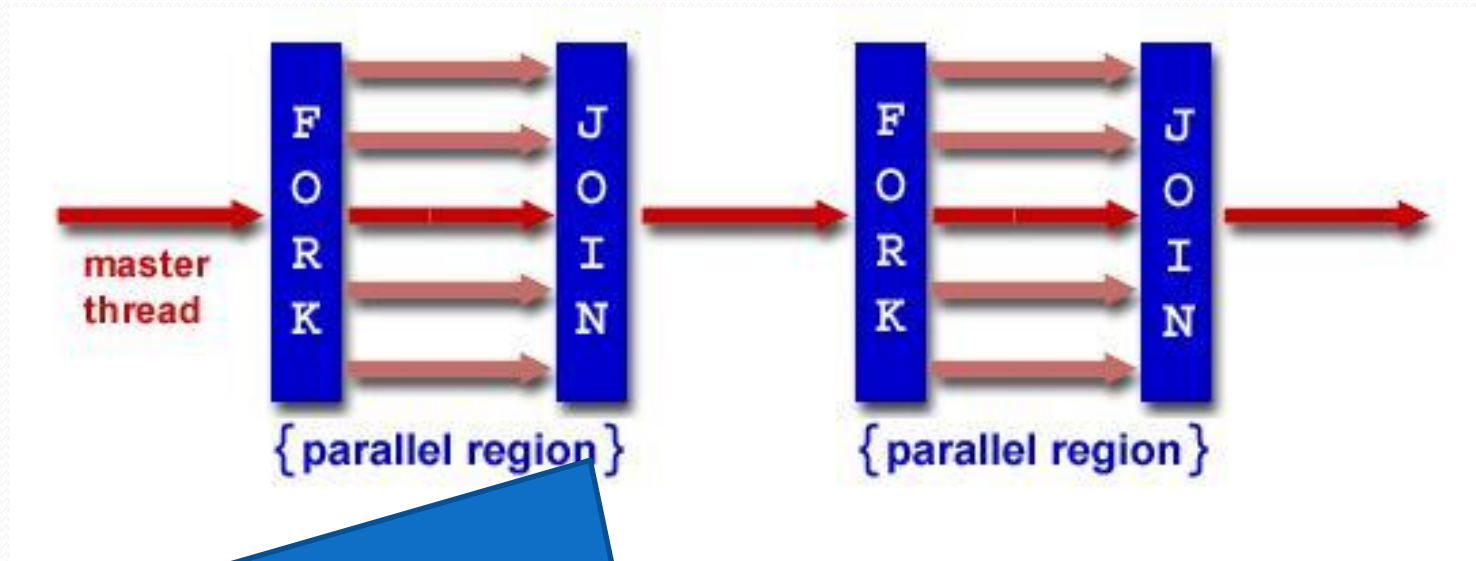
- Il modello d'esecuzione parallela è quello *fork-join*



**Fork:** comincia una *regione parallela*, viene quindi creato un team di thread che procede parallelamente

# Introduzione ad OpenMp

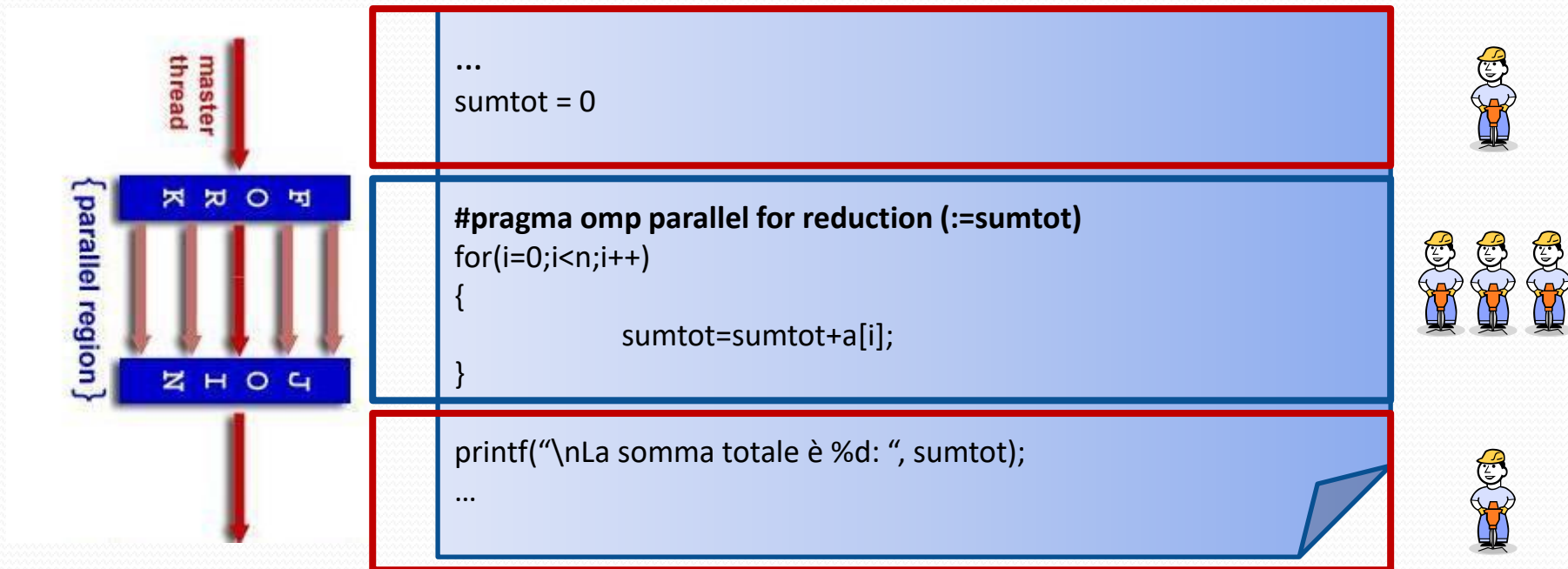
- Il modello d'esecuzione parallela è quello *fork-join*



**Join:** tutti i thread del team hanno terminato le istruzioni della regione parallela, si sincronizzano e terminano, lasciando proseguire solo il master thread.

# Introduzione ad OpenMp

- Il modello d'esecuzione parallela è quello *fork-join*





# Introduzione ad OpenMp

- I thread portano i propri dati in **cache**



Non è garantita automaticamente e costantemente la **consistenza** della memoria



Molta attenzione alla scelta delle variabili  
**condivise/private**

# Introduzione ad OpenMp

- È fondamentalmente composto da:
  - **Direttive per il compilatore**



# Introduzione ad OpenMp

- È fondamentalmente composto da:
  - **Direttive per il compilatore**
    - Completate eventualmente da **clausole** che ne dettagliano il comportamento
  - **Runtime Library Routines**, per intervenire sulle variabili di controllo interne allo standard, a run-time (deve essere incluso il file omp.h).
    - Es. Numero di thread, informazioni sullo scheduling,...
  - **Variabili d'ambiente**, per modificare il valore delle variabili di controllo interne prima dell'esecuzione.

# Struttura generica del codice

```
main ()  
{  
    int var1, var2, var3;  
    ...  
    codice sequenziale  
    ...  
    ...  
    ...  
    ...  
    codice sequenziale  
    ...  
    ...  
    ...  
    ...  
    ...  
    codice sequenziale  
    ...  
}
```

Il primo passo per parallelizzare un codice sequenziale con OpenMp consiste nell'individuare quali istruzioni possono essere eseguite in parallelo

# Struttura generica del codice

```
main ()  
{  
    int var1, var2, var3;  
    ...  
    codice sequenziale  
    ...  
    ...  
    ...  
    ...  
    codice sequenziale  
    ...  
    ...  
    ...  
    ...  
    ...  
    codice sequenziale  
    ...  
}
```

A volte è molto semplice individuare porzioni ben definite di codice da sottoporre ad una direttiva (es. caso della Somma di n numeri)

# Struttura generica del codice

```
main ()  
{  
    int var1, var2, var3;  
    ...  
    codice sequenziale  
    ...  
    ...  
    ...  
    ...  
    codice sequenziale  
    ...  
    ...  
    ...  
}
```

Necessità di studiare una strategia di parallelizzazione come abbiamo imparato a fare nel caso della memoria distribuita

Altre volte, può essere necessario riorganizzare parti di codice per creare gruppi di istruzioni indipendenti, eseguibili in maniera concorrente

# Struttura generica del codice

```
main ()  
{
```

```
    int var1, var2, var3;
```

```
    ...
```

```
    codice sequenziale
```

```
    ...
```

```
    ...
```

```
    ...
```

```
    ...
```

```
    codice sequenziale
```

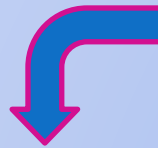
```
    ...
```

```
    ...
```

```
    ...
```

Spesso utile per migliorare le performance

Altre volte, può essere necessario riorganizzare parti di codice per creare gruppi di istruzioni indipendenti, eseguibili in maniera concorrente



# Struttura generica del codice

```
main ()  
{  
    int var1, var2, var3;  
    ...  
    codice sequenziale  
    ...  
    ...  
    ...  
    ...  
    codice sequenziale  
    ...  
    ...  
    ...  
    ...  
    ...  
    codice sequenziale  
    ...  
}
```

Una volta individuate le  
porzioni di codice che  
possono essere  
parallelizzate



# Struttura generica del codice

```
main ()  
{
```

```
    int var1, var2, var3;
```

```
    ...
```

```
    Parte che deve rimanere sequenziale
```

```
    ...
```

```
    ...
```

```
    ...
```

```
    ...
```

```
    ...
```

```
    Sezione che può essere eseguita concorrentemente da più thread
```

```
    ...
```

```
    ...
```

```
    ...
```

```
    Parte che deve rimanere sequenziale
```

```
    ...
```

```
}
```

Una volta individuate le  
porzioni di codice che  
possono essere  
parallelizzate...

# Struttura generica del codice

```
main ()  
{
```

```
    int var1, var2, var3;
```

```
    ...
```

```
    Parte che deve rimanere sequenziale
```

```
    ...
```

```
    ...
```

```
    ...
```

```
    ...
```

```
    ...
```

```
    Sezione che può essere eseguita concorrentemente da più thread
```

```
    ...
```

```
    ...
```

```
    ...
```

```
    Parte che deve rimanere sequenziale
```

```
    ...
```

```
}
```

... si introducono le opportune direttive OpenMp, con le relative clausole

# Struttura generica del codice

```
#include <omp.h>
main ()
{
    int var1, var2, var3;
    ...
    Parte sequenziale
    ...
    Inizio della regione parallela:.
    si genera un team di thread e si specifica lo scope de

    #pragma omp parallel private(var1, var2) shared(var3)
    {
        Sezione parallela eseguita da tutti i thread
        ...
        Tutti i thread confluiscono nel master thread
    }

    Ripresa del codice sequenziale
    ...
}
```

... si introducono le opportune direttive OpenMp, con le relative clausole

# Direttive

- Le **direttive** si applicano al costrutto OpenMp immediatamente seguente

```
#pragma omp directive-name [clause[ [, ]clause] ...] new-line
```

- Il costrutto *parallel* forma un team di thread ed avvia così un'esecuzione parallela

```
#pragma omp parallel [clause[ [, ]clause] ...] new-line
{
    structured-block
}
clause: if(scalar-expression)
num_threads(integer-expression)
default(shared | none)
private(list)
firstprivate(list)
shared(list)
copyin(list)
reduction(operator: list)
```

# Direttive

- Alla fine del blocco di istruzioni è sottintesa una barriera di sincronizzazione: tutti i thread si fermano ad aspettare che tutti gli altri abbiano completato l'esecuzione, prima di ritornare ad una esecuzione sequenziale.
- Tutto quello che segue questa direttiva verrà eseguito da ogni thread

Non è considerato l'ordine delle clausole

```
#pragma omp parallel [clause[ [, ]clause] ...] new-line  
{  
    structured-block  
}  
clause: if(scalar-expression)  
num_threads(integer-expression)  
default(shared | none)  
private(list)  
firstprivate(list)  
shared(list)  
copyin(list)  
reduction(operator: list)
```

Può comparire al più una volta

Può comparire al più una volta, vale limitatamente a questa regione

# Direttive

- Alla fine del blocco di istruzioni è sottintesa una barriera di sincronizzazione: tutti i thread si fermano ad aspettare che tutti gli altri abbiano completato l'esecuzione, prima di ritornare ad una esecuzione sequenziale.
- Tutto quello che segue questa direttiva verrà eseguito da ogni thread
- Dopo aver generato i thread, è necessario stabilire anche la distribuzione del lavoro tra i thread del team, per evitare ridondanze inutili e/o dannose

# Direttive

- Ci sono tre tipi di costrutti detti *WorkSharing* perché si occupano della distribuzione del lavoro al team di thread: **for**, **sections**, **single**
- Anche all'uscita da un costrutto work-sharing è sottintesa una barriera di sincronizzazione, se non diversamente specificato dal programmatore

# Direttive

- Il costrutto *for* specifica che le iterazioni del ciclo contenuto devono essere distribuite tra i thread del team

Perché  
“for”



```
#pragma omp for [clause[,] clause] ... ] new-line
{
  for-loops
}
clause: private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
schedule(kind[, chunk_size])
collapse(n)
ordered
nowait
```

Solo per  
questo  
costrutto

Esclusione  
della barriera  
in uscita

Perché non  
“while”





# Direttive

- Il costrutto *sections* conterrà un insieme di costrutti *section* ognuno dei quali verrà eseguito da un thread del team



Le diverse  
sezioni  
devono  
poter essere  
eseguite in  
ordine  
arbitrario

```
#pragma omp sections [clause[[,] clause] ...] new-line
{
  [#pragma omp section new-line]
  structured-block
  [#pragma omp section new-line]
  structured-block ]
  ...
}
clause: private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
nowait
```



Rischio di  
sbilanciamento  
del carico

# Direttive

- Il costrutto *single* specifica che il blocco di istruzioni successivo verrà eseguito da un solo thread QUALSIASI del team

```
#pragma omp single [clause[,] clause] ...] new-line
{
    structured-block
}
clause: private(list)
firstprivate(list)
copyprivate(list)
nowait
```

Gli altri thread attendono che questo termini la sua porzione di codice

- I costrutti WorkSharing possono essere combinati con il costrutto *parallel*, e le clausole ammesse sono l'unione di quelle ammesse per ognuno.

```
#pragma omp parallel for [clause[,] clause] ...] new-line
{
    for-loop
}
```

# Direttive

- Il costrutto *master* specifica che il blocco di istruzioni successivo verrà eseguito dal solo master thread

```
#pragma omp master  
{  
    structured-block  
}
```

Non sono sottintese  
barriere di  
sincronizzazione né  
all'ingresso né all'uscita  
del costrutto!

# Direttive

- Il costrutto *critical* forza l'esecuzione del blocco successivo ad un thread alla volta: è utile per gestire le **regioni critiche**

```
#pragma omp critical [(name)] new-line  
{  
    structured-block  
}
```

Si può assegnare un NOME alla regione che sarà *globale* al programma

- Il costrutto *barrier* forza i thread di uno stesso task ad attendere il completamento di tutte le istruzioni precedenti da parte di tutti gli altri

```
#pragma omp barrier new-line
```

Al momento del barrier (implicito o esplicito) si crea una vista consistente dei dati dei thread

- Ci sono altri costrutti!

# Clausole

- Non tutte le clausole sono valide per tutte le direttive
- Quasi tutte accettano una lista di argomenti separati da virgole
- Questi argomenti devono comparire nel costrutto a cui viene applicata la clausola

# Clausole

- *default(shared|none)*: controlla gli attributi di data-sharing delle variabili in un costrutto

## MODELLO S-M

In genere tutti i dati  
sono condivisi da tutti i  
thread

Se non specificata  
questa clausola, il  
compilatore stabilisce  
quali variabili devono  
essere private per ogni  
thread

# Clausole

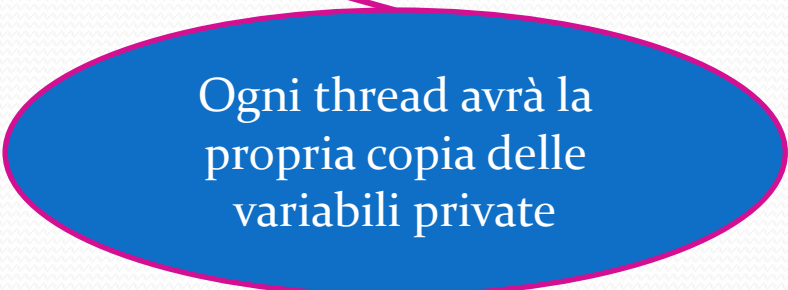
- *default(shared|none)*: controlla gli attributi di data-sharing delle variabili in un costrutto

**shared**: tutte le variabili saranno considerate condivise

**none**: deciderà tutto il programmatore

# Clausole

- *default(shared|none)*: controlla gli attributi di data-sharing delle variabili in un costrutto
- *shared(list)*: gli argomenti contenuti in *list* sono condivisi tra i thread del team
- *private(list)*: gli argomenti contenuti in *list* sono privati per ogni thread che li utilizza



Ogni thread avrà la propria copia delle variabili private



# Clausole

- *default(shared|none)*: controlla gli attributi di data-sharing delle variabili in un costrutto
- *shared(list)*: gli argomenti contenuti in *list* sono condivisi tra i thread del team
- *private(list)*: gli argomenti contenuti in *list* sono privati per ogni thread che li utilizza

Le variabili private hanno un valore indefinito all'entrata e all'uscita della regione parallela

Ogni thread avrà la propria copia delle variabili private

# Clausole

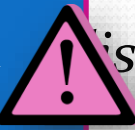
- *default(shared|none)*: controlla gli attributi di data-sharing delle variabili in un costrutto
- *shared(list)*: gli argomenti contenuti in *list* sono condivisi tra i thread del team
- *private(list)*: gli argomenti contenuti in *list* sono privati per ogni thread che li utilizza
- *firstprivate(list)*: gli argomenti contenuti in *list* sono privati per i thread e vengono inizializzati con il valore che avevano gli originali al momento in cui è stato incontrato il costrutto in questione

All'ingresso le copie private sono pre-inizializzate con il valore che ha la variabile con lo stesso nome prima di incontrare la regione parallela

# Clausole

- *default(shared|none)*: controlla gli attributi di data-sharing delle variabili in un costrutto
- *shared(list)*: gli argomenti contenuti in *list* sono condivisi tra i thread del team
- *private(list)*: gli argomenti contenuti in *list* sono privati per ogni thread
- *firstprivate(list)*: All'uscita le variabili manterranno come valore l' "ultimo" della sezione parallela (se fosse stata eseguita in sequenziale) privati per i thread originali e quelli nuovi sono gli originali. Il costrutto in questione è stato modificato in
- *lastprivate(list)*: gli argomenti contenuti in *list* sono privati per i thread e quelli originali verranno aggiornati al termine della regione parallela

# Clausole

- *default(shared|none)*: controlla gli attributi di data-sharing delle variabili in un costrutto
- *shared(list)*: gli argomenti contenuti in *list* sono condivisi tra i thread. L'ordine di esecuzione dei thread non è specificato ... quindi ATTENZIONE con i valori f.p.! Risultati numericamente non determinati. 
- *private(list)*: gli argomenti contenuti in *list* sono privati per ogni thread. Ogni thread avrà una copia privata delle variabili in *list*, e al termine del costrutto la variabile sarà condivisa.
- *lastprivate(list)*: gli argomenti contenuti in *list* sono privati per i thread originali. Ogni thread avrà una copia privata delle variabili in *list*, e al termine del costrutto la variabile sarà condivisa.
- *reduction(operator:list)*: gli argomenti contenuti in *list* verranno combinati utilizzando l'operatore associativo specificato.

# Clausole

Si possono migliorare  
le performance, ma...  
**ATTENZIONE!**



- ***nowait***: elimina la barriera implicita alla fine del costrutto (non della regione parallela). I thread non aspettano gli altri e continuano.
- ***schedule(kind[,chunk\_size])***: specifica il modo (*kind*) di distribuire le iterazioni del ciclo seguente; *chunk\_size* è il numero (>0) di iterazioni contigue da assegnare allo stesso thread, mentre *kind* può essere:
  - **static**: chunk assegnati secondo uno scheduling round-robin
  - **dynamic**: chunk assegnati su richiesta. Quando un thread termina il proprio, ne chiede un altro.
  - **guided**: variante del dynamic.
  - **runtime**: decisione presa a runtime attraverso la variabile d'ambiente OMP\_SCHEDULE.

# Runtime Library Routines

- **omp\_set\_num\_threads(*scalar-integer-expression* )**: definisce il numero di thread da utilizzare
- **omp\_get\_max\_threads()**: restituisce il numero massimo di thread disponibili per la prossima regione parallela
- **omp\_set\_dynamic(*scalar-integer-expression* )**: permesso (o) o meno (1) al sistema di riadattare il numero di thread utilizzati. Ritorna il valore attuale.
- **omp\_get\_thread\_num()**: restituisce l'id del thread
- **omp\_get\_num\_procs()**: restituisce il numero di processori disponibili per il programma al momento della chiamata
- **omp\_get\_num\_threads()**: restituisce il numero di thread del team

# Variabili d'ambiente

- **OMP\_NUM\_THREADS**: numero di thread che verranno utilizzati nell'esecuzione/i successiva/e
  - `OMP_NUM_THREADS(integer)`
  - `sh/bash: export OMP_NUM_THREADS=integer`
- **OMP\_DYNAMIC**: permesso (true) o meno (false) al sistema di riadattare il numero di thread utilizzati
- **OMP\_SCHEDULE**: stabilisce lo scheduling di default da applicare nei costrutti *for*
  - `OMP_SCHEDULE(type [,chunk])`
  - `sh/bash: export OMP_SCHEDULE="type [,chunk]"`
  - Tipi possibili: *static*, *dynamic*, *guided*.

# Esempio: Hello World

Libreria

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

Creazione di un  
team di thread

```
    #pragma omp parallel
```

```
    printf("Hello from thread %d, nthreads %d\n", omp_get_thread_num(), omp_get_num_threads());
    return 0;
```

```
}
```

Library function  
per conoscere l'id  
del thread  
chiamante

Library function  
per conoscere il  
numero di thread  
attivi



# Compilare ed eseguire

- Potete utilizzare le stesse macchine del cluster, di cui ogni nodo è utilizzabile come una macchina ad 8 core.

**Nome server:** ui-studenti.scope.unina.it

**Nome utente:** dato a lezione

**Password:** data a lezione

- SEMPRE ATTRAVERSO IL SISTEMA PBS!!!!

# Compilare ed eseguire

- OpenMp viene implementato da molti compilatori, tra questi il gcc (v. 4 e superiori)
- Per compilare basta
  - aggiungere al comando di compilazione l'opzione `-fopenmp`
  - fare link alla libreria omp, con l'opzione `-lgomp`

```
gcc -fopenmp -lgomp -o nome-eseguibile nome-codice.c
```

# Compilare ed eseguire

- OpenMp viene implementato da vari fornitori, tra questi il glibc
  - Per compilare con OpenMp:
    - usare il compilatore con l'opzione `-fopenmp`
    - fare attenzione a non usare librerie che non supportino OpenMp
- Questo comando va  
SEMPRE dato attraverso  
uno script PBS!!

# Compilare ed eseguire

- Una volta compilato il codice, basta lanciare l'eseguibile come di consueto

```
./nome-eseguibile
```

- Si possono modificare prima delle variabili d'ambiente proprie dello standard OpenMp, come accennato.

```
export OMP_NUM_THREADS=4
```

```
export OMP_SCHEDULE="dynamic,2"
```

# Compilare ed eseguire

- Una volta compilato il codice, è possibile come di consueto

- Si può eseguire il codice in parallelo attraverso uno script PBS!!

Questi comandi vanno  
SEMPRE dati attraverso  
uno script PBS!!

# Prendere il tempo

- Il modo consigliato per verificare il tempo di esecuzione del programma è utilizzare la funzione

`int gettimeofday(timeval *tp, NULL)`

contenuta in `<sys/time.h>` (da includere!)

- Questa funzione deve essere chiamata subito prima e subito dopo la regione parallela.

# Prendere il tempo

## Esempio

```
timeval time;  
double inizio,fine;  
...  
gettimeofday(&time, NULL);  
inizio=time.tv_sec+(time.tv_usec/1000000.0);
```

## REGIONE PARALLELA

```
gettimeofday(&time, NULL);  
fine=time.tv_sec+(time.tv_usec/1000000.0);  
...  
printf("tempo impiegato: %e\n", fine-inizio);
```

# Riferimenti bibliografici

## **Using OpenMp**

*B. Chapman, G. Jost, R. van der Pas*

The MIT Press

<http://openmp.org/>

<https://computing.llnl.gov/tutorials/openMP/>