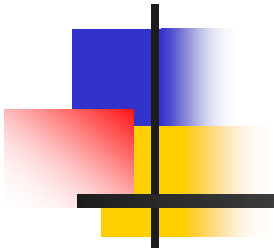


Calcolo Parallelo e Distribuito

openMP:

clausola reduction - costrutto WS: FOR – costrutto CRITICAL

Docente: Prof. L. Marcellino

Tutor: Prof. P. De Luca

Somma II strategia

Siete riusciti a usare…

reduction(operator: list)

?

Oggi vediamo insieme qualche nozione in più che vi aiuterà!

Direttive

Si usano per creare un team e stabilire quali istruzioni devono essere eseguite in parallelo e come queste devono essere distribuite tra i thread del team creato

La **direttiva**

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

prevede:

- il costrutto ***parallel***, che forma un team di thread ed avvia così un'esecuzione parallela

```
#pragma omp parallel [clause], [clause] ...
{

}
clause:
num_threads(integer-expression)
default(shared | none)
private(list)
firstprivate(list)
shared(list)
copyin(list)
reduction(operator: list)
```

Direttive

La **direttiva**

Si usano per creare un team e stabilire quali istruzioni devono essere eseguite in parallelo e come queste devono essere distribuite tra i thread del team creato

#pragma omp ***costrutto*** *[clause], [clause] ... new-line*

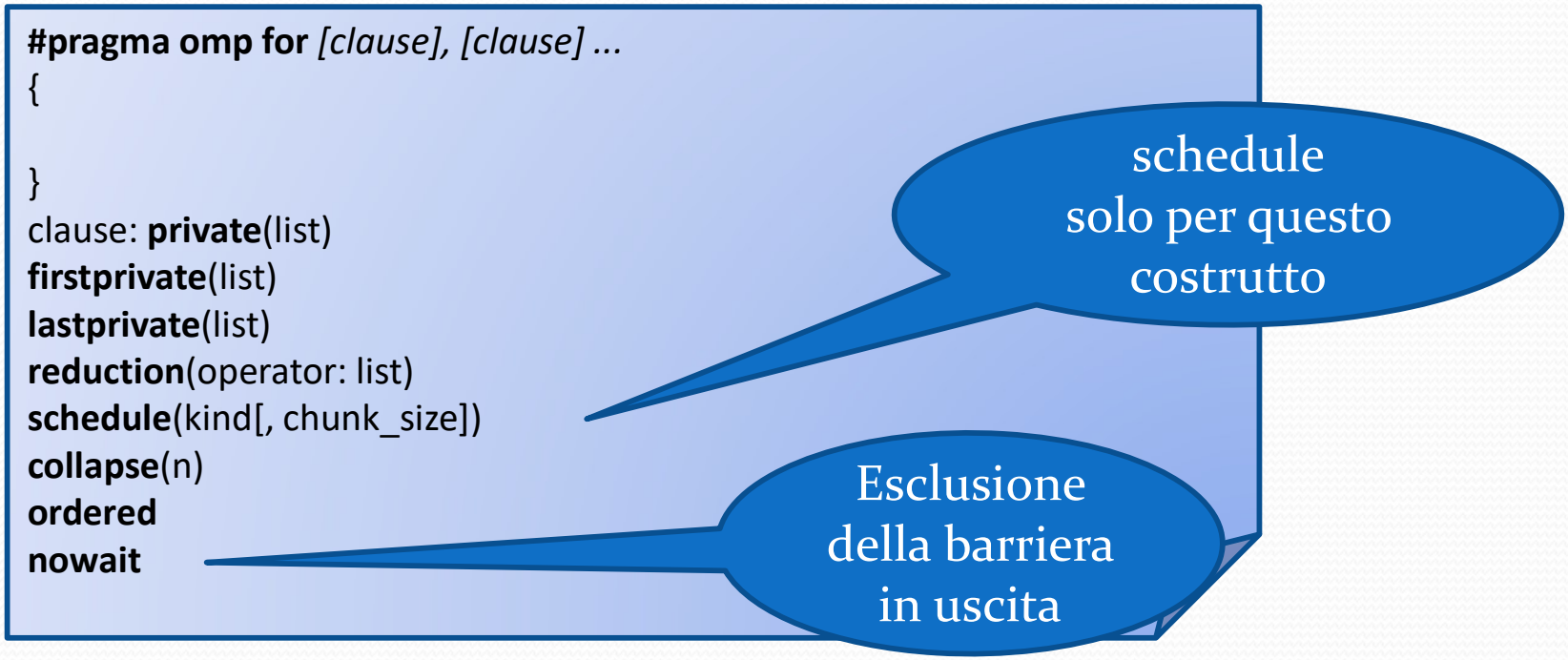
oltre al costrutto **parallel** prevede anche:

- **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 debbano essere distribuite tra i thread del team (secondo un ordine che non specificherò in dettaglio, come ho fatto con la somma; ma userò le potenzialità di OpenMP!)



Clausole

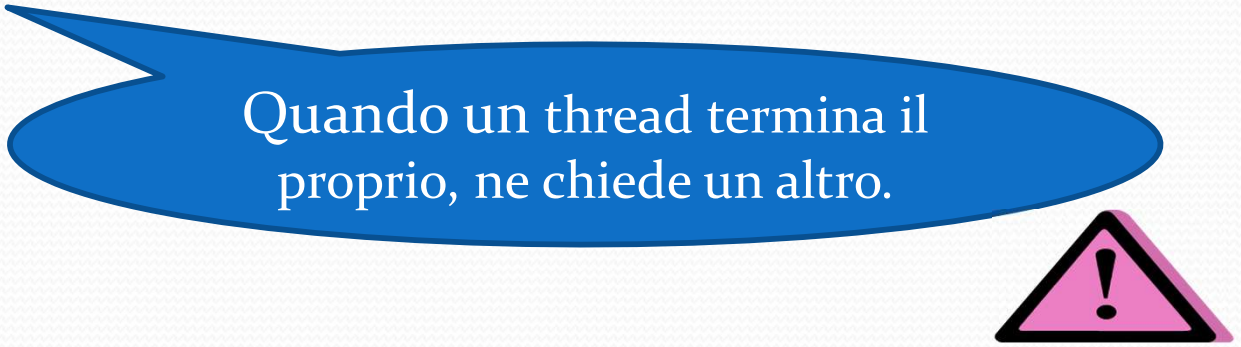
Si possono migliorare le performance, ma... ATTENZIONE!

- *nowait*: elimina la barriera implicita alla fine del costrutto. I thread non aspettano che tutti abbiano finito, ma continuano a lavorare.

Clausole

- *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/ guided** : chunk assegnati su richiesta.



  - **runtime**: decisione presa a runtime attraverso la variabile d'ambiente OMP_SCHEDULE.

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 da applicare nei *costrutti for*
  - OMP_SCHEDULE(*type [,chunk])*
  - sh/bash: export OMP_SCHEDULE="*type [,chunk]*"
  - Tipi possibili: *static*, *dynamic*.

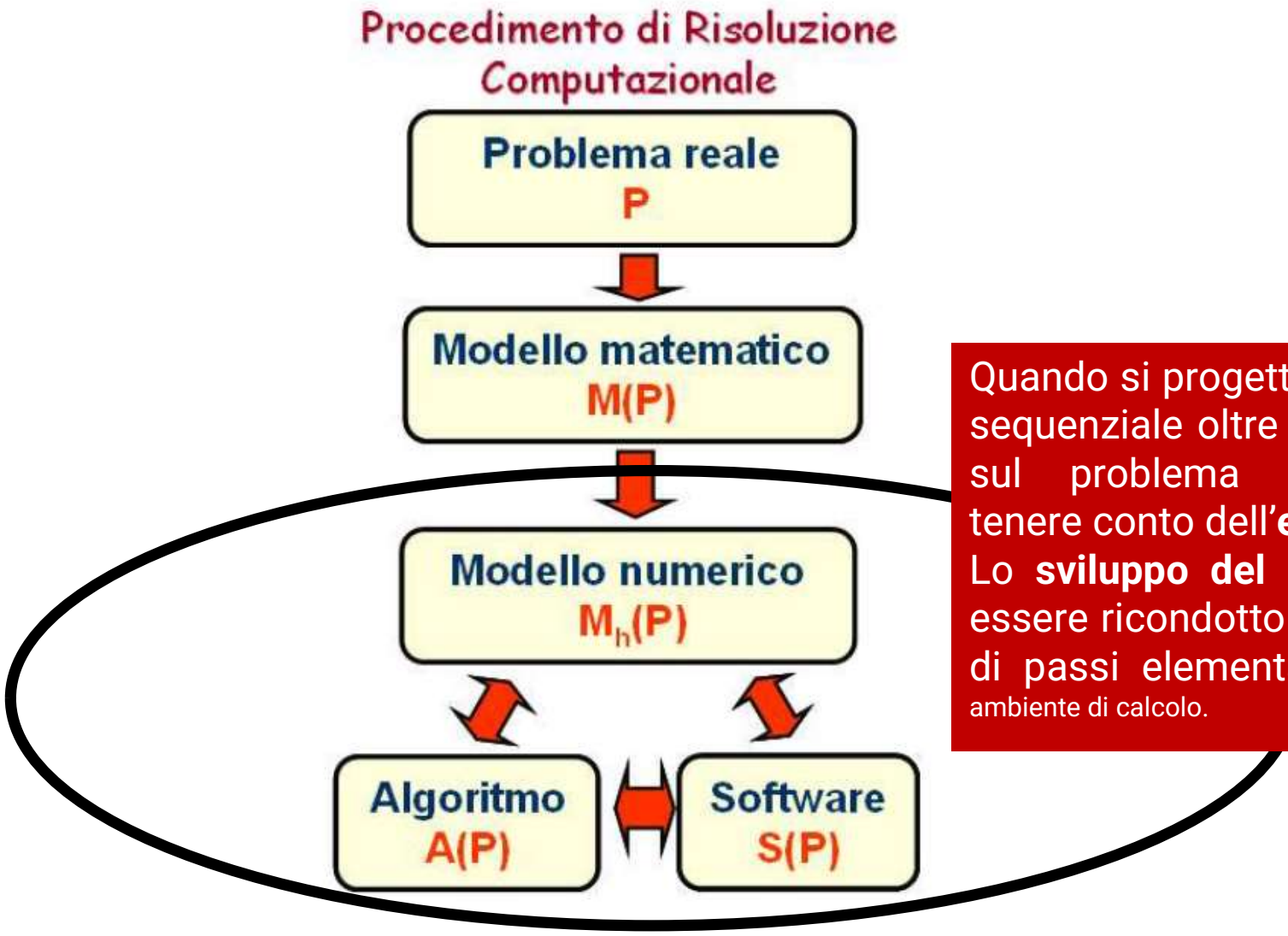
Clausole

- *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 thr
- *firs* privati per i thr no gli orig utto in questione
- *lastprivate(list)*: gli argomenti contenuti in *list* sono privati per i thread e quelli originali verranno aggiornati al termine della regione parallela

All'uscita le variabili manterranno come valore l'*ultimo* della sezione parallela

Attenzione: *firstprivate* si usa con **parallel**, invece *lastprivate* si usa solo con un altro costrutto il **for** (ora possiamo capire perché!)

Modellizzazione di problemi su larga scala



Quando si progetta un algoritmo sequenziale oltre a concentrarsi sul problema è importante tenere conto dell'**esecutore**.
Lo **sviluppo del software** deve essere ricondotto ad un insieme di passi elementari, per lo specifico ambiente di calcolo.

Esercizio: calcolo di π

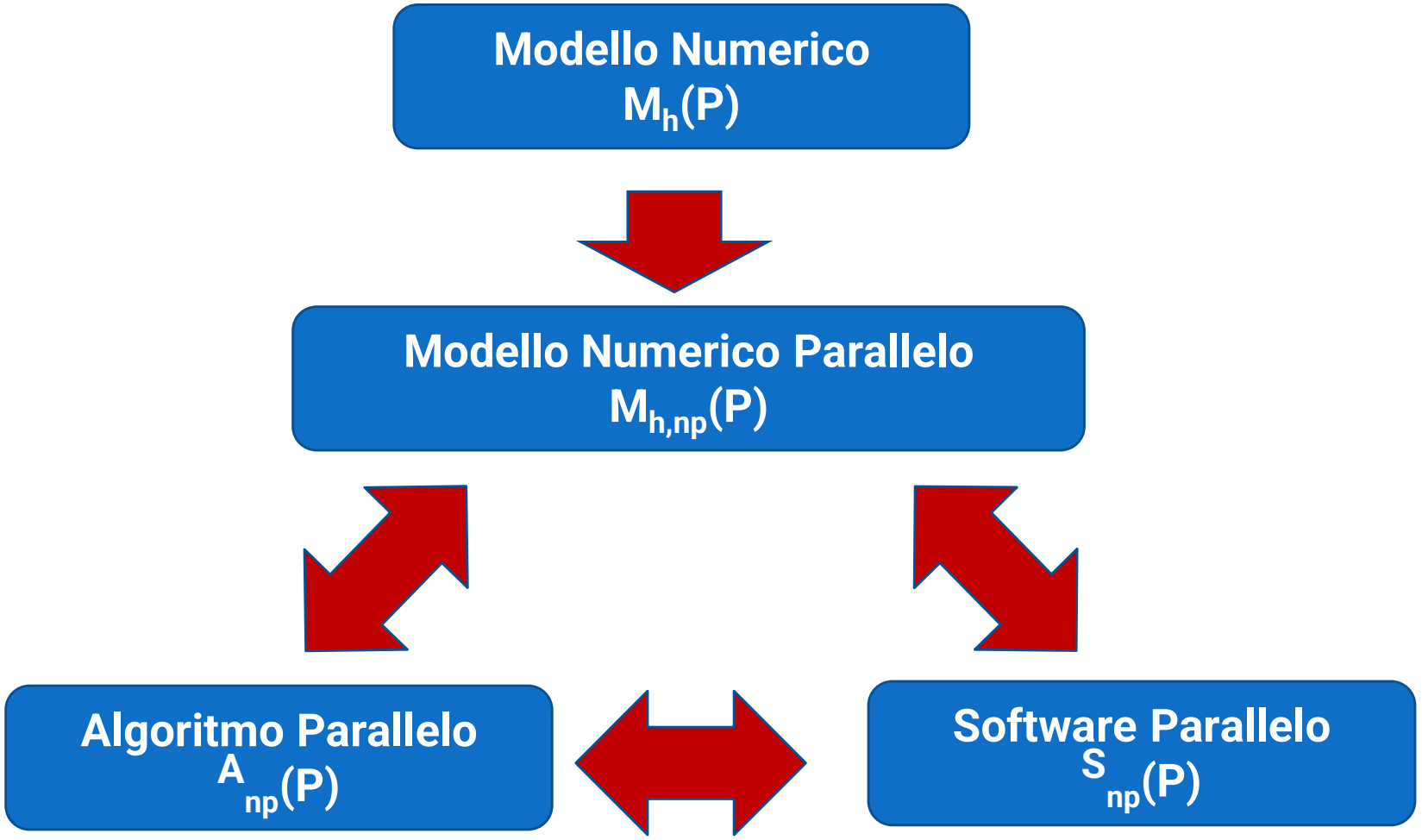
Tra tutti i modi per calcolare il numero π, si può considerare il seguente integrale:

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

Numericamente:

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

algoritmi e software paralleli



Anche in questo caso, per progettare un algoritmo parallelo, è fondamentale non perdere mai di vista quali siano le capacità intrinseche dell'architettura parallela a disposizione o necessaria.

Esempio: calcolo π

```
#include <stdio.h>
#define N 100000000
int main(int argc, char **argv)
{
   long int i, n = N;
   double x, dx, f, sum, pi;
   printf("numero di intervalli: %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("PI %.24f\n", pi);

return 0;
}
```

Esempio: π

- Compiliamo ed eseguiamo

```
gcc –o pi.o pi.c
```

```
./pi
```

- Ora proviamo a parallelizzare usando il costrutto **FOR** (combinato a parallel)

```
#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;
```

Esempio: π

- Bisogna stare attenti solo a quali variabili sono shared e quali private
- Combinando il costrutto **parallel** con il costrutto **for** tutte le istruzione da fare devono essere comprese tra le parentesi del for interno!
- E' possibile anche personalizzare lo scheduling

```
#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;
```

Esempio: π pi_par_v1

```
#include <stdio.h>
#include<omp.h>
#define N 100000000
int main(int argc, char **argv)
{
   long int i, n = N;
   double x, dx, f, sum, pi;
   printf("numbero di intervalli: %ld\n", n);
   sum = 0.0;

   dx = 1.0 / (double) n;
   #pragma omp parallel for private(x,f,i) shared(dx,sum,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("PI %.24f\n", pi);

return 0;
}
```

Esempio: π

- Compiliamo ed eseguiamo

```
gcc –fopenmp –o pi_par_v1.o pi_par_v1.c
```

```
export OMP_NUM_THREADS=numero
```

```
./pi_par_v1
```

- Che succede?
- facendo variare il numero di thread si ottengono risultati differenti ed errati
- tutti i thread leggono e modificano **sum** senza coordinarsi
- ma è necessario che **sum** sia shared per ottenere il risultato dopo il loop parallelo
- E' nessario, quindi che i thread aggiornino **sum** uno alla volta

il costrutto **CRITICAL**

Direttive

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

```
#pragma omp critical
```

Non fa parte dei costrutti work-sharing!

Anche perché simula un sequenziale

Esempio: π pi_par_v2

```
#include <stdio.h>
#include<omp.h>
#define N 100000000
int main(int argc, char **argv)
{
   long int i, n = N;
   double x, dx, f, sum, pi;
   printf("numbero di intervalli: %ld\n", n);
   sum = 0.0;

   dx = 1.0 / (double) n;
   #pragma omp parallel for private(x,f,i) shared(dx,sum,n)
   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;
    printf("PI %.24f\n", pi);

return 0;
}
```

Esempio: π

- Compiliamo ed eseguiamo

```
gcc –fopenmp –o pi_par_v2.o pi_par_v2.c
```

```
export OMP_NUM_THREADS=numero
```

```
./pi_par_v2
```

- Che succede?
- facendo variare il numero di thread e la dimensione N, i risultati sono corretti, ma il tempo d'esecuzione aumenta!
- tutti i thread, ad ogni iterazione, sono in contesa per l'accesso alla sezione critica (**1 strategia**)
- C'è un modo più efficiente (**2 strategia**)

Esempio: π pi_par_v3

```
#include <stdio.h>
#include<omp.h>
#define N 100000000
int main(int argc, char **argv)
{
   long int i, n = N;
   double x, dx, f, sum, pi;
   printf("numbero di intervalli: %ld\n", n);
   sum = 0.0;

   dx = 1.0 / (double) n;
   #pragma omp parallel for private(x,f,i) shared(dx,n) reduction(+:sum)
//    attenzione ho dovuto togliere sum dai parametri shared
   for (i = 1; i <= n; i++)
   {
       x = dx * ((double) (i - 0.5));
       f = 4.0 / (1.0 + x*x);
       //  commento:    #pragma omp critical
       sum = sum + f;
    }

    pi = dx*sum;
    printf("PI %.24f\n", pi);

return 0;
}
```

Esempio: π

- Compiliamo ed eseguiamo

```
gcc -fopenmp –o pi_par_v3.o pi_par_v3.c
```

```
export OMP_NUM_THREADS=numero
```

```
./pi_par_v3
```

L'algoritmo è molto più veloce, provate da soli a prendere i tempi e fate un po' di confronti con osservazioni!

Somma II strategia

Ora forse riuscirete ad implementare…

reduction(operator: list)