

# 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)
```

# 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 delle variabili

    #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
    ...
}
```

# 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

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

```
#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
```

# Direttive

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

```
#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
```

# 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
```

# Direttive

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

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

# 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

# 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 (0) 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



# 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
```

# 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);
```

```
#!/bin/bash

#####
#          #
# The PBS directives #
#          #
#####

#PBS -q studenti
#PBS -l nodes=1:ppn=8      # si riservano così 8 processori su un nodo. Se ne servono meno, si può mettere un numero minore.
#PBS -N somma
#PBS -o somma.out
#PBS -e somma.err
#####

# -q coda su cui va eseguito il job #
# -l numero di nodi richiesti #
# -N nome job(stesso del file pbs) #
# -o, -e nome files contenente l'output #

#####
#          #
#      qualche informazione sul job #
#          #

#####

echo 'Job is running on node(s): '
cat $PBS_NODEFILE

PBS_O_WORKDIR=$PBS_O_HOME/ProgettoSommaOpenMP
echo -----
```

```
PBS_O_WORKDIR=$PBS_O_HOME/ProgettoSommaOpenMP
```

```
echo -----
```

```
echo PBS: qsub is running on $PBS_O_HOST
```

```
echo PBS: originating queue is $PBS_O_QUEUE
```

```
echo PBS: executing queue is $PBS_QUEUE
```

```
echo PBS: working directory is $PBS_O_WORKDIR
```

```
echo PBS: execution mode is $PBS_ENVIRONMENT
```

```
echo PBS: job identifier is $PBS_JOBID
```

```
echo PBS: job name is $PBS_JOBNAME
```

```
echo PBS: node file is $PBS_NODEFILE
```

```
echo PBS: current home directory is $PBS_O_HOME
```

```
echo PBS: PATH = $PBS_O_PATH
```

```
echo -----
```

```
1  
export OMP_NUM_THREADS=2          # numero di thread generati di default durante le regioni parallele  
  
export PSC_OMP_AFFINITY=TRUE      # per legare i thread a particolari processori  
  
echo "Compilo..."  
  
gcc -fopenmp -lgomp -o $PBS_O_WORKDIR/somma $PBS_O_WORKDIR/sommaOpenMP.c  
  
# nell'esempio il primo argomento è il numero di thread da utilizzare e il secondo la dimensione n  
# il numero di thread deve essere minore o uguale del numero di processori riservati sul nodo  
  
echo "Eseguo..."  
  
$PBS_O_WORKDIR/somma 4 500000000
```

# Parallel and Distributed Computing

a.a. 2021-2022

Prof. Giuliano Laccetti  
Università degli Studi di Napoli Federico II

slides basate sul testo

A. Murli «Lezioni di Calcolo Parallelo»

e

su appunti di lezioni di Calcolo Parallelo e Distribuito tenute dal prof. A. Murli

## CORSO

- ⇒ Lezioni ed esercitazioni in aula
- ⇒ Esercitazioni in laboratorio

## ESAMI

- ⇒ Prova scritta in aula +  
elaborati consegnati durante il corso / prova pratica in laboratorio  
prova orale

# Modalità

# Riferimenti Bibliografici – 1

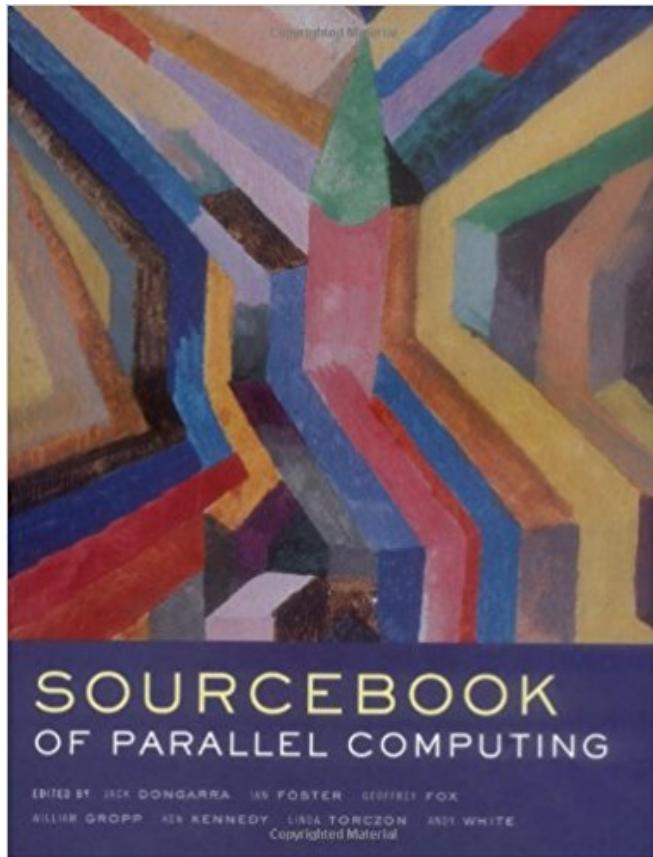


## **Lezioni di Calcolo Parallello**

Almerico Murli

**Liguori Editore**

# Riferimenti Bibliografici – 2



## Sourcebook of Parallel Computing

Jack Dongarra, Ian Foster, Geoffrey C. Fox, William Gropp, Ken Kennedy, Linda Torczon, Andy White

Morgan Kaufmann Edition

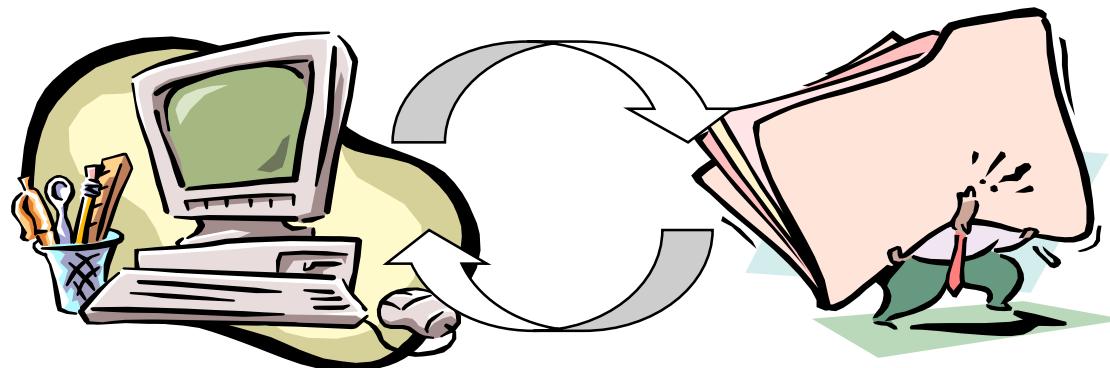
# Introduzione al corso

» Perché Calcolo Parallelo?

# Obiettivo del corso

Fornire idee, strumenti software e metodologie  
alla base della  
risoluzione computazionale di un problema  
mediante  
**calcolatore parallelo**

Attività di  
laboratorio



Lezioni  
in aula

# L'attività di laboratorio ....

Progetto, sviluppo, analisi e implementazione  
di algoritmi in ambienti ad  
**alte prestazioni**





Che cos'è il calcolo ad alte prestazioni ?

# Supercomputing

The real purpose of having a  
**super** computer  
is to solve problems

# Supercomputing

The real purpose of having a  
**super** computer  
is to solve problems

**Supercomputing** is  
To solve a problem by using a supercomputer

# Supercomputing

Il termine “**supercalcolatore**” si riferisce ad un sistema che fornisce le prestazioni più elevate (in quel dato momento).

(’80s)

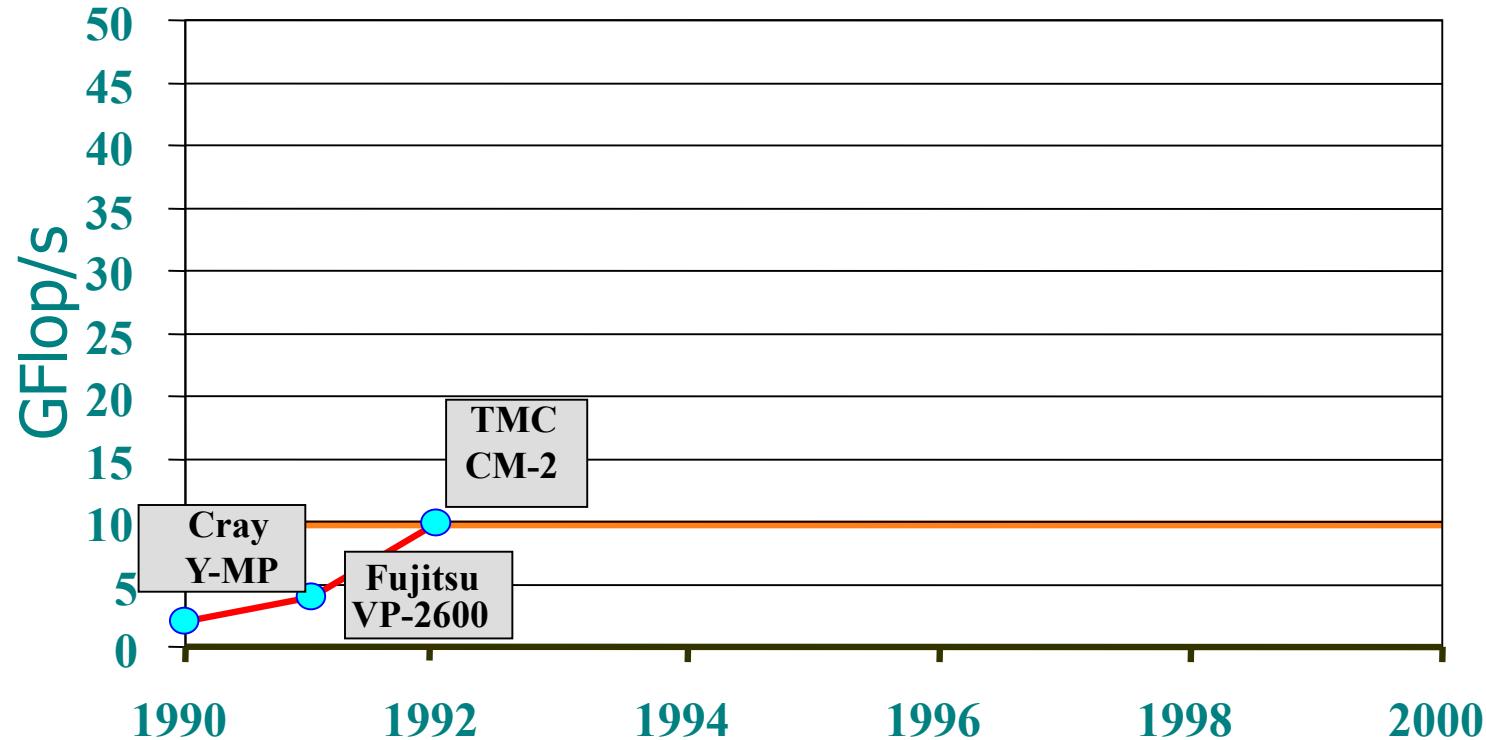
La prestazione è misurata dal tempo necessario per risolvere una particolare applicazione  
**(application-dependent)**

# Supercomputing

Dal 1993

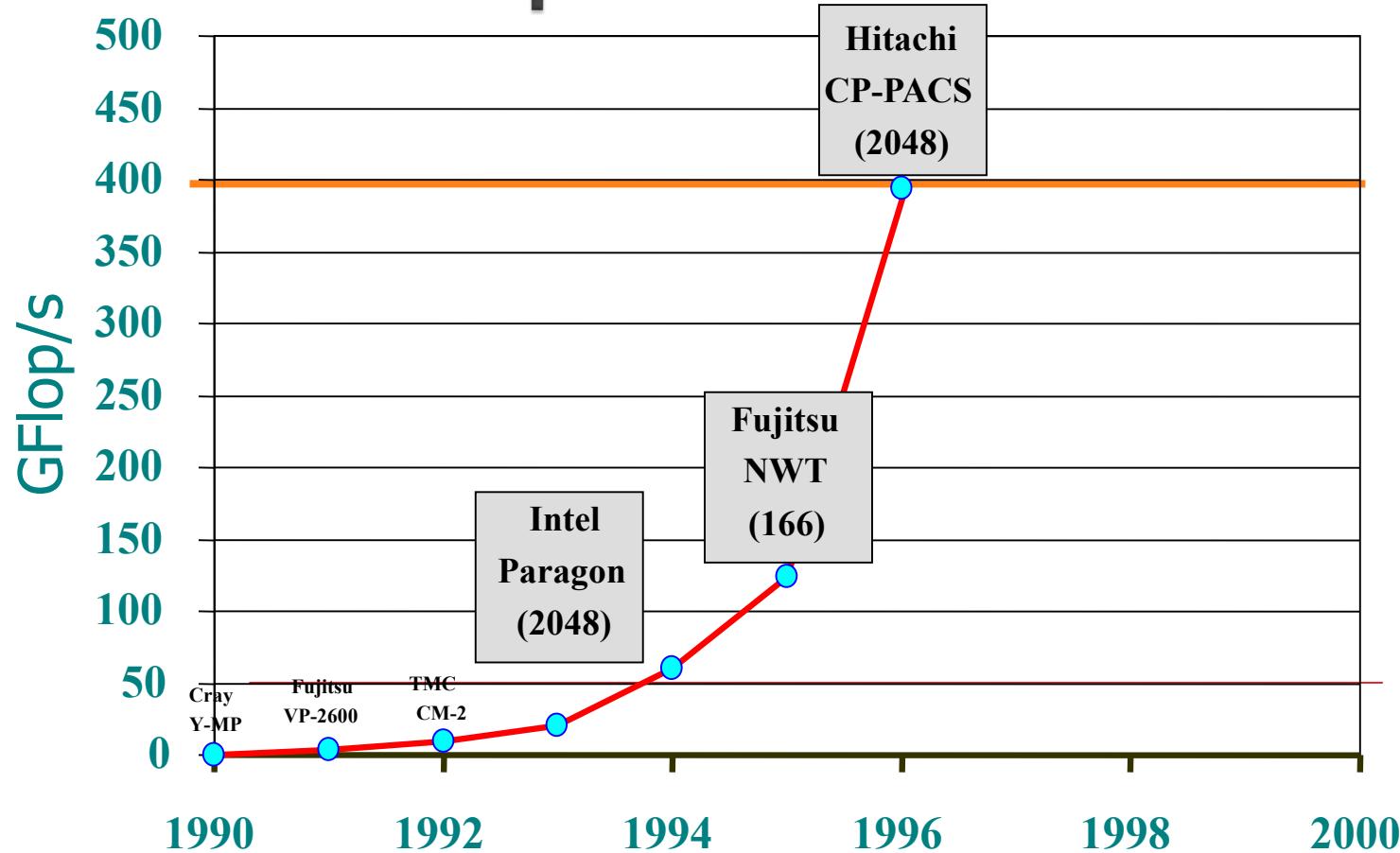
- ▶ TOP 500: lista dei calcolatori più veloci nel mondo
- ▶ **Rmax**: performance misurata dai benchmark di LINPACK per la risoluzione di  $AX=B$

# I calcolatori più veloci



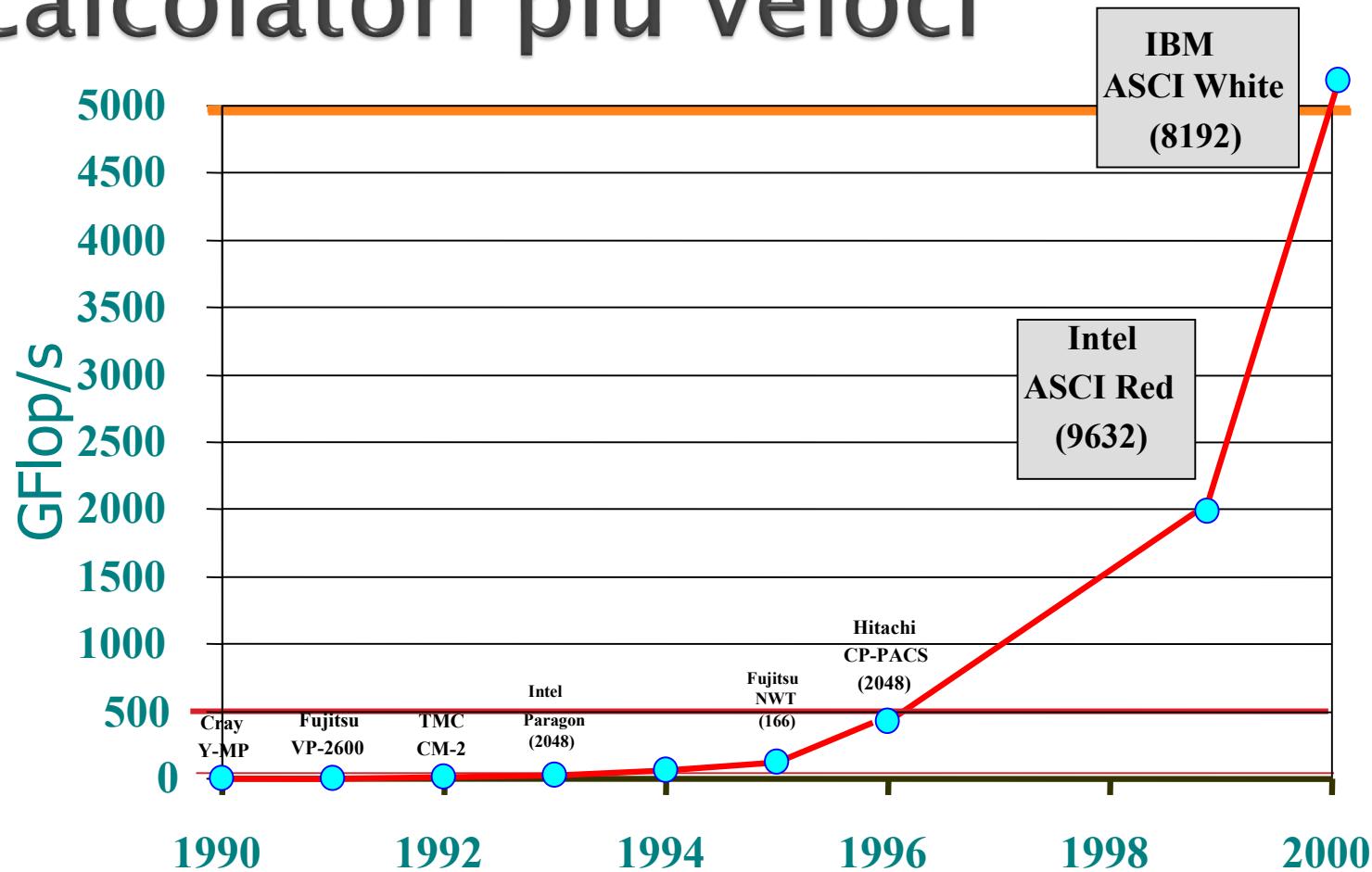
Un calcolo che nel 1980 durava 1 anno (CRAY 1, 100 Mflop)  
può essere effettuato nel 1992 (CM-2, 10 Gflop)  
in circa 87 ore

# I calcolatori più veloci



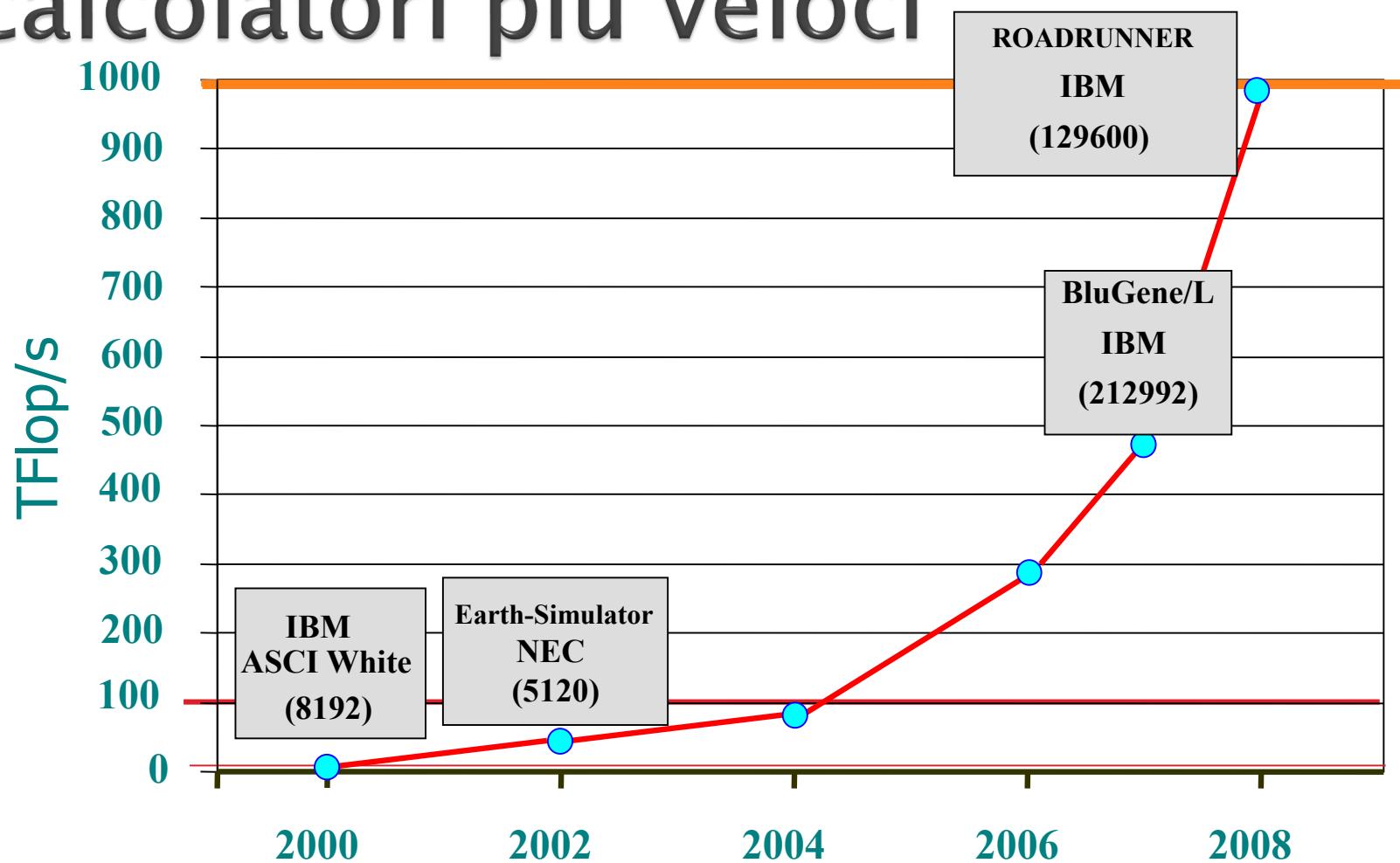
Un calcolo che nel 1980 durava 1 anno (CRAY 1, 100 Mflop)  
può essere effettuato nel 1996 (Hitachi, 400 GFlop)  
in circa 2 ore!

# I calcolatori più veloci



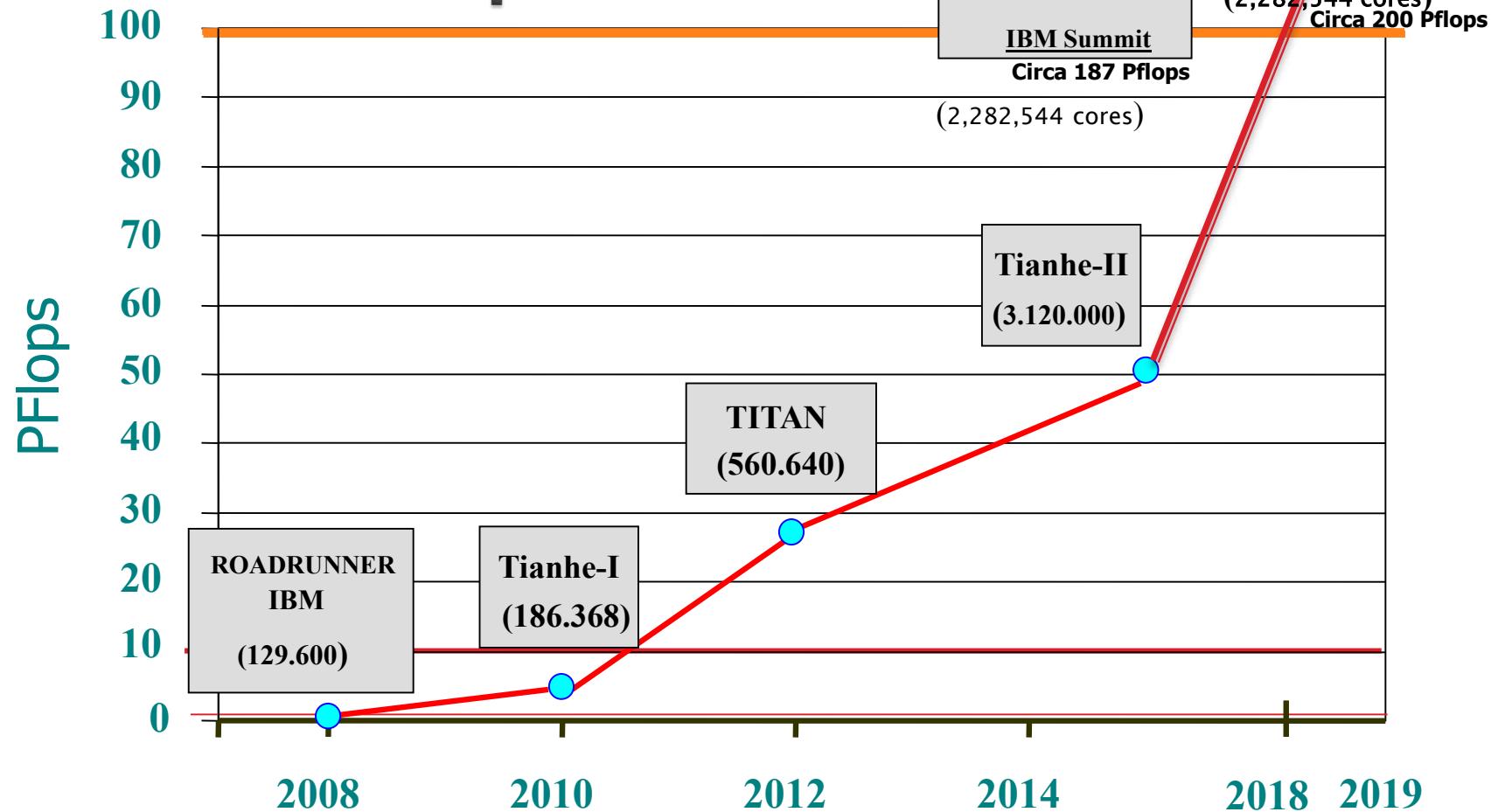
Un calcolo che nel 1980 durava 1 anno (CRAY 1, 100 Mflop)  
può essere effettuato nel 2000 (ASCI White, 7 Tflop)  
in circa 7 minuti!

# I calcolatori più veloci



Un calcolo che nel 1980 durava 1 anno (CRAY 1, 100 Mflop)  
può essere effettuato nel 2008 (ROADRUNNER, 1 Pflop)  
in meno di 3 secondi!

# I calcolatori più veloci



Un calcolo che nel 1980 durava 1 anno (CRAY 1, 100 Mflops)  
può essere effettuato nel 2019 (IBM Summit, Oak Ridge Lab, 200 Pflops)  
in circa 15 millesimi di secondo!

FLOPS

## Teraflops

Tianhe-2 (MilkyWay-2), China, 3.120.000 cores,  
peak performance 54.902 Teraflops

FUGAKU, Fujitsu, 7,630,848 cores, peak  
performance 537.212 Tflops, [RIKEN](#)  
[Center for Computational Science](#), Japan

SUMMIT, IBM, 2,414,592 cores, peak  
performance 200.794 Tflops, [Oak Ridge](#)  
[Nat. Lab, DOE USA](#)

Giugno 2021 442mila Tflops  
Rmax

Giugno 2019

Sunway TaihuLight, China, [National Supercomp. Center](#),  
Wuxi, 10.649.600 cores, peak perf. 125.436 TFlops

2015

2016

Year

2017

2018

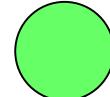
2019 ...

**Un calcolo che nel 1980 durava 1 anno (CRAY 1, 100 Mflops) può essere effettuato nel 2020 (Fugaku Fujitsu, Riken Center, Giappone, circa 500 Pflops) in circa 6 millesimi di secondo!**

# Supercomputing

## Calcolo ad “alte prestazioni”

- Anni '70 – '80:  $10^6$  flops (MFLOPs)
  - ✓ Calcolatori sequenziali-scalari (CDC 7600, IBM 360)
- Anni '81 – '90:  $10^9$  flops (GFLOPs)
  - ✓ Calcolatori vettoriali (CRAY 1, CRAY X-MP)
- Anni '91 – 2000:  $10^{12}$  flops (TFLOPs)
  - ✓ Calcolatori a parallelismo massiccio (CRAY T3D, ASCI White)
- Anni 2001 – 2010:  $10^{15}$  flops (PFLOPs)
  - ✓ Calcolatori a parallelismo massiccio (Roadrunner)
- Anni 2011 – 2016:  $10^{17}$  flops (187 PFLOPs – Summit, Oak Ridge Lab)
  - ✓ Calcolatori a parallelismo massiccio (Tianhe-2; Sunway Taihu, Summit)
- Anni 2017–202x:
  - ✓  $10^{18}$  flops (EFLOPs) ? (exaflop/s)



Ma ... il calcolo ad alte prestazioni  
è davvero necessario?



**In autostrada: velocita',  
affidabilita', sicurezza,....**

**dipende dall'uso!**

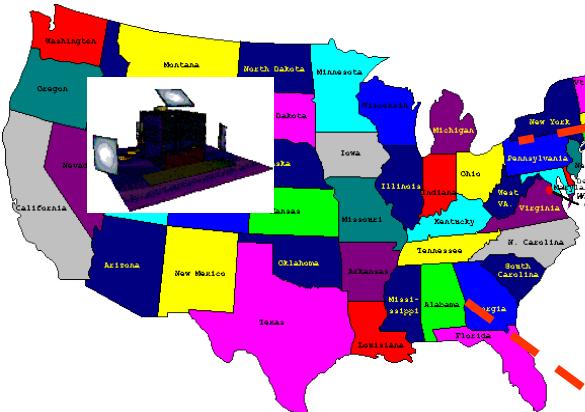
**In citta':**  
traffico, parcheggi,  
consumo...





Difesa del territorio?

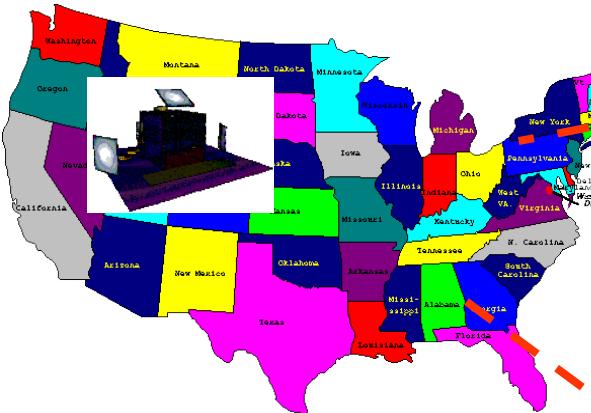
Nella Guerra del Golfo, il Calcolo ad Alte Prestazioni, ha guidato Patriot contro gli Scud Iracheni





Difesa del  
territorio?

Nella Guerra del Golfo, il Calcolo ad  
Alte Prestazioni, ha guidato Patriot  
contro gli Scud Iracheni



Il missile deve essere distrutto  
prima che arrivi sull'obiettivo

Entro 2 minuti dal lancio del missile nemico!



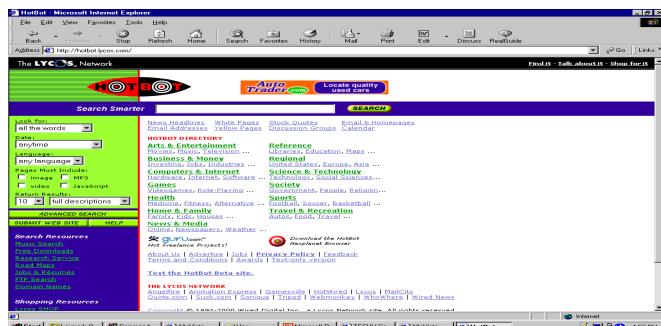
# Altre applicazioni

altavista: SEARCH



## ▶ Ricerca su Internet

- Ogni giorno circa un milione di persone interroga su Internet un motore di ricerca



Google™

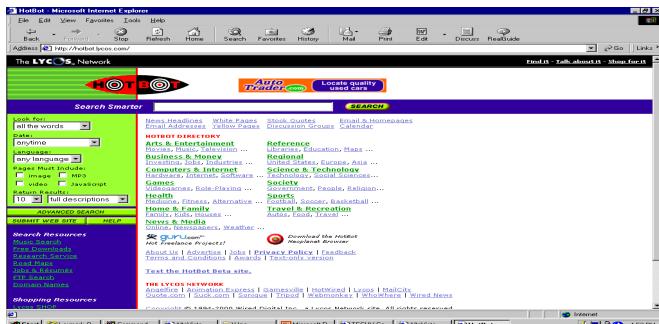
# Altre applicazioni

altavista: SEARCH



## ▶ Ricerca su Internet

- Ogni giorno circa milioni e milioni di persone interrogano su Internet un motore di ricerca
  - Quanto si è disposti ad aspettare?



Google™

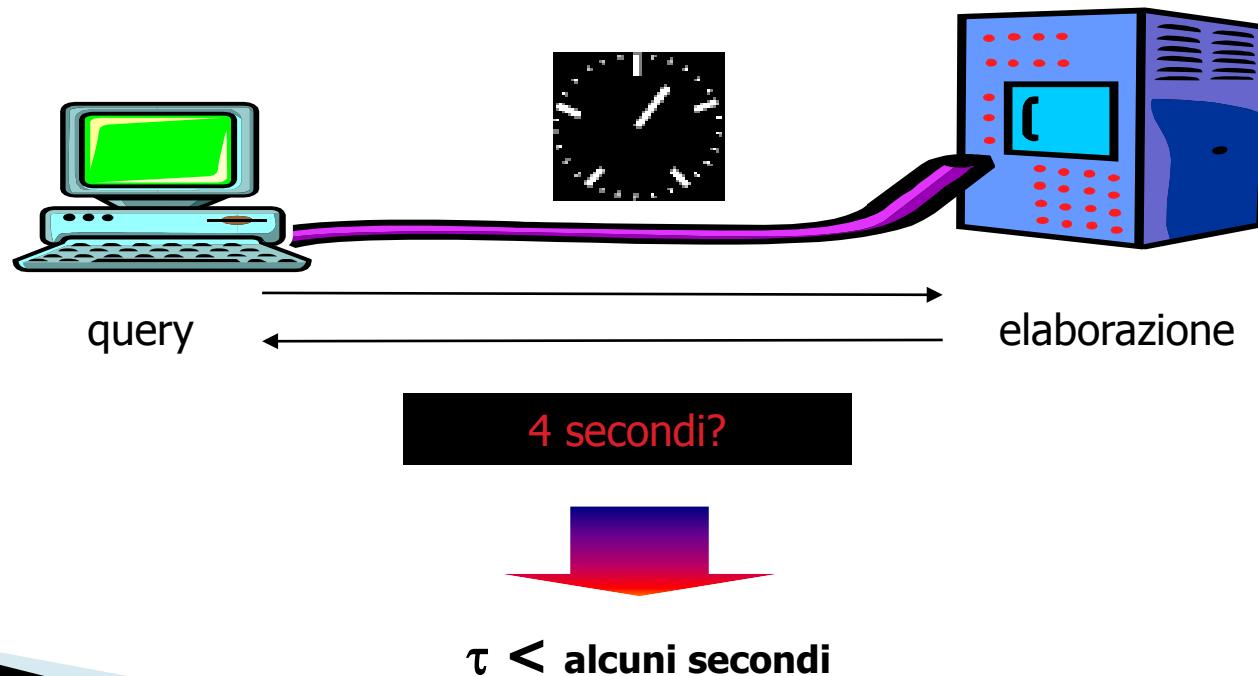
# Altre applicazioni

**alta**vista: SEARCH



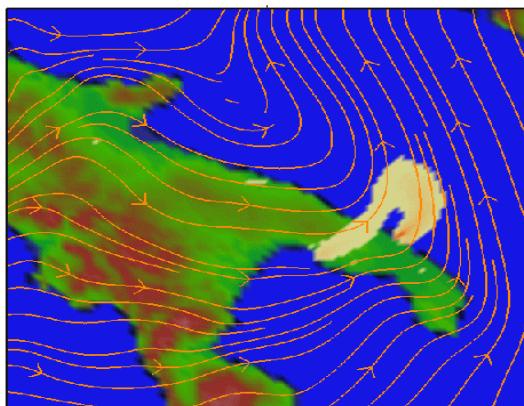
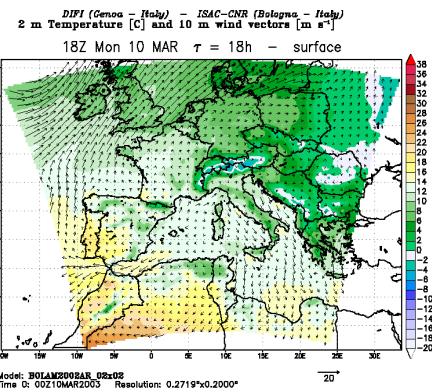
## ▶ Ricerca su Internet

- Ogni giorno circa un milione di persone interroga su Internet un motore di ricerca
    - Quanto si è disposti ad aspettare?



# Altre applicazioni

## ▶ Previsioni metereologiche



- superficie da monitorare  
**20 milioni di Km<sup>2</sup>**
- altezza sul livello del mare  
**20 Km**
- discretizzazione dello spazio 3D  
mediante cubi di lato 100 m  
**in 1 km<sup>3</sup>: $10 \times 10 \times 10 = 10^3$  cubi**

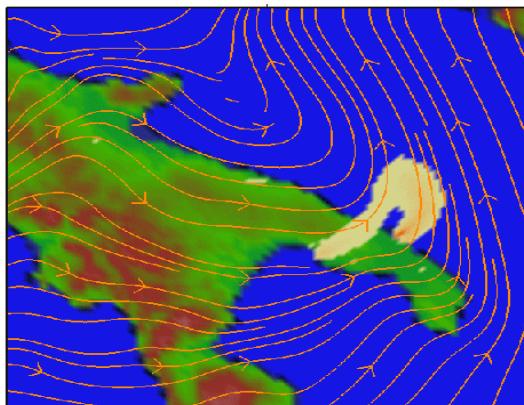
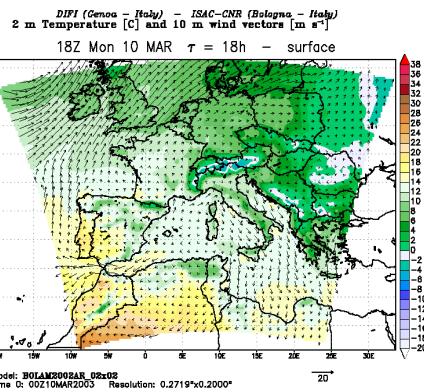
$$20 \times 10^6 \text{ km}^2 \times 20 \text{ km} = 4 \times 10^8 \text{ km}^3$$



$$\mathbf{4 \times 10^8 \times 10^3 = 4 \times 10^{11} \text{ cubi}}$$

# Altre applicazioni

## ▶ Previsioni metereologiche



- superficie da monitorare  
**20 milioni di Km<sup>2</sup>**
- altezza sul livello del mare  
**20 Km**
- discretizzazione dello spazio 3D  
mediante cubi di lato 100 m  
**in 1 km<sup>3</sup>: $10 \times 10 \times 10 = 10^3$  cubi**

$$20 \times 10^6 \text{ km}^2 \times 20 \text{ km} = 4 \times 10^8 \text{ km}^3$$



$$\mathbf{4 \times 10^8 \times 10^3 = 4 \times 10^{11} \text{ cubi}}$$

Il modello deve fornire le previsioni per i prossimi 2 giorni.

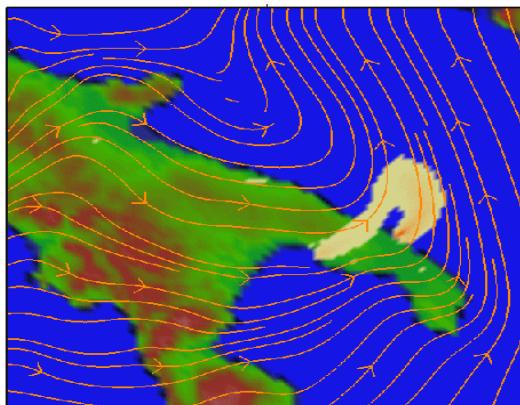
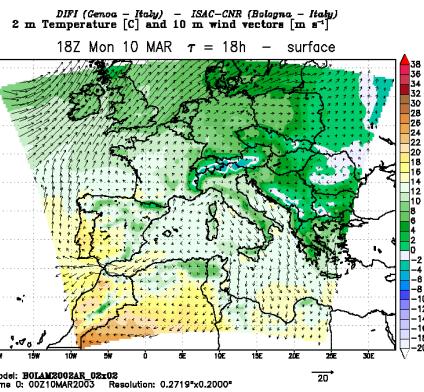
Supponiamo che l'algoritmo effettui, per ogni ora di simulazione,  $10^2$  flop per cubo

- per l'intera superficie sono richiesti  $4 \times 10^{11} \times 10^2$  flop
  - Per **2 giorni**:  $4 \times 10^{11} \times 10^2 \times 48$  flop

$$\mathbf{2 \times 10^{15} \text{ operazioni}}$$

# Altre applicazioni

## ▶ Previsioni metereologiche



- superficie da monitorare  
**20 milioni di Km<sup>2</sup>**
- altezza sul livello del mare  
**20 Km**
- discretizzazione dello spazio 3D  
mediante cubi di lato 100 m  
**in 1 km<sup>3</sup>: $10 \times 10 \times 10 = 10^3$  cubi**

$$20 \times 10^6 \text{ km}^2 \times 20 \text{ km} = 4 \times 10^8 \text{ km}^3$$



$$\mathbf{4 \times 10^8 \times 10^3 = 4 \times 10^{11} \text{ cubi}}$$

Il modello deve fornire le previsioni per i prossimi 2 giorni.

Supponiamo che l'algoritmo effettui, per ogni ora di simulazione,  $10^2$  flop per cubo

➤ per l'intera superficie sono richiesti  $4 \times 10^{11} \times 10^2$  flop

➤ Per **2 giorni**:  $4 \times 10^{11} \times 10^2 \times 48$  flop ➔

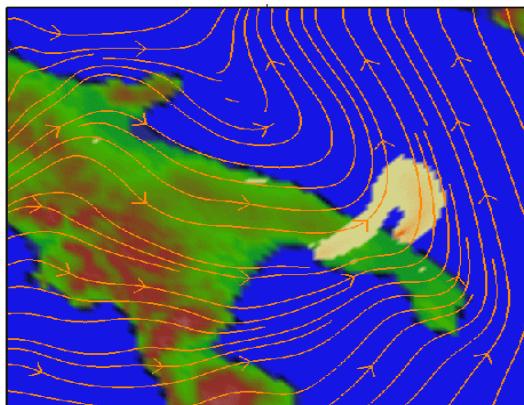
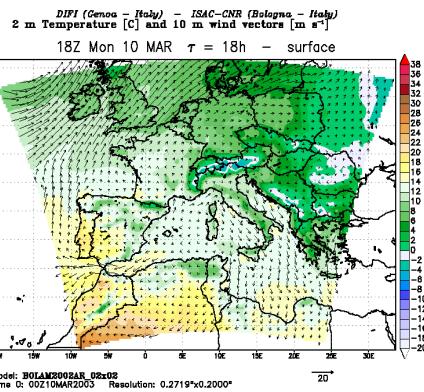
Con un PC ➔

$\frac{2 \times 10^{15} \text{ operazioni}}{1 \times 10^{10} \text{ flop/s}}$

$$= 2 \times 10^5 \text{ sec} = \mathbf{2 \text{ giorni}}$$

# Altre applicazioni

## ▶ Previsioni metereologiche



- superficie da monitorare  
**20 milioni di Km<sup>2</sup>**
- altezza sul livello del mare  
**20 Km**
- discretizzazione dello spazio 3D  
mediante cubi di lato 100 m  
**in 1 km<sup>3</sup>:10 x 10 x 10 = 10<sup>3</sup> cubi**

$$20 \times 10^6 \text{ km}^2 \times 20 \text{ km} = 4 \times 10^8 \text{ km}^3$$



$$\mathbf{4 \times 10^8 \times 10^3 = 4 \times 10^{11} \text{ cubi}}$$

Il modello deve fornire le previsioni per i prossimi 2 giorni.

Supponiamo che l'algoritmo effettui, per ogni ora di simulazione,  $10^2$  flop per cubo

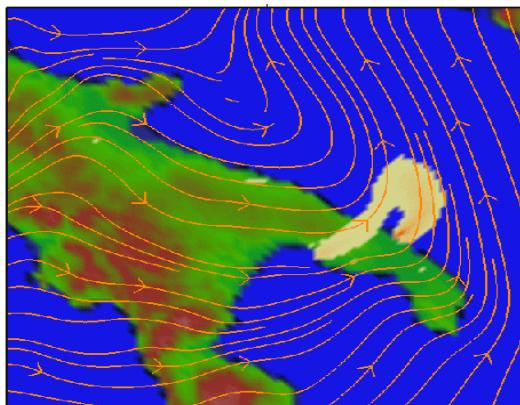
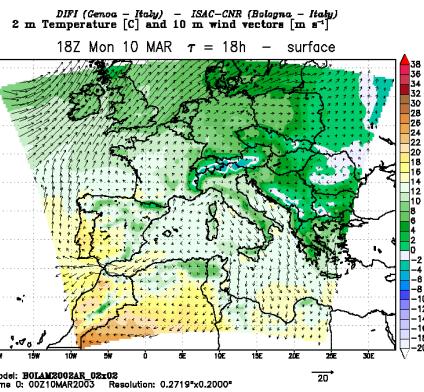
- per l'intera superficie sono richiesti  $4 \times 10^{11} \times 10^2$  flop
  - Per **2 giorni**:  $4 \times 10^{11} \times 10^2 \times 48$  flop

$$\mathbf{2 \times 10^{15} \text{ operazioni}}$$

**30 minuti**

# Altre applicazioni

## ▶ Previsioni metereologiche



- superficie da monitorare  
**20 milioni di Km<sup>2</sup>**
- altezza sul livello del mare  
**20 Km**
- discretizzazione dello spazio 3D  
mediante cubi di lato 100 m  
**in 1 km<sup>3</sup>: $10 \times 10 \times 10 = 10^3$  cubi**

$$20 \times 10^6 \text{ km}^2 \times 20 \text{ km} = 4 \times 10^8 \text{ km}^3$$



$$\mathbf{4 \times 10^8 \times 10^3 = 4 \times 10^{11} \text{ cubi}}$$

Il modello deve fornire le previsioni per i prossimi 2 giorni.

Supponiamo che l'algoritmo effettui, per ogni ora di simulazione,  $10^2$  flop per cubo

➤ per l'intera superficie sono richiesti  $4 \times 10^{11} \times 10^2$  flop

➤ Per **2 giorni**:  $4 \times 10^{11} \times 10^2 \times 48$  flop ➔

$2 \times 10^{15}$  operazioni

E' necessario 1 TFLOPs !!!!! ➔

$1 \times 10^{12}$  flop/s

=  $1.8 \times 10^3$  sec = **30 minuti**

# Come ridurre i tempi di risposta (*turnaround time*) di una simulazione?

## ▶ In generale

- una rappresentazione semplificata del tempo richiesto dalla risoluzione numerica di un problema è:

$$\tau = k \cdot T(n) \cdot \mu$$

$T(N)$ = complessità di tempo dell'algoritmo

Dipendenza dall'algoritmo

$\mu$ =tempo di esecuzione di 1 op. f.p.

Dipendenza dal calcolatore

# Soluzione A

## ▶ In generale

- una rappresentazione semplificata del tempo richiesto dalla risoluzione numerica di un problema è:

$$\tau = k \cdot T(n) \cdot \mu$$

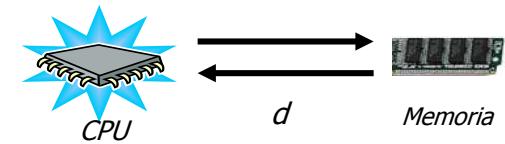
$\mu$ =tempo di esecuzione di 1 op. f.p.

Dipendenza dal calcolatore

Si può ridurre migliorando la tecnologia

Velocità della luce  $c=3 \times 10^8$  m/s

$$d = \mu c$$



# Soluzione A

## In generale

- una rappresentazione semplificata del tempo richiesto dalla risoluzione numerica di un problema è:

$$\tau = k \cdot T(n) \cdot \mu$$

$T(N)$ = complessità di tempo dell'algoritmo

Dipendenza dall'algoritmo

$\mu$ =tempo di esecuzione di 1 op. f.p.

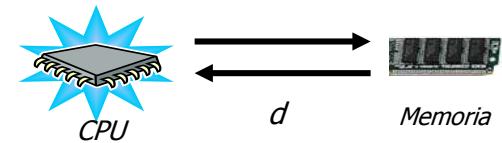
Dipendenza dal calcolatore

Si può ridurre migliorando la tecnologia

Limiti tecnologici  
(packaging e raffreddamento)

Velocità della luce  $c=3 \times 10^8$  m/s

$$d = \mu c$$



# Soluzione A

## ▶ In generale

- una rappresentazione semplificata del tempo richiesto dalla risoluzione numerica di un problema è:

$$\tau = k \cdot T(n) \cdot \mu$$

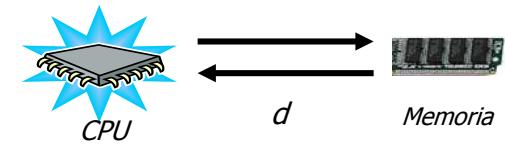
$\mu$ =tempo di esecuzione di 1 op. f.p.

Dipendenza dal calcolatore

Si può ridurre migliorando la tecnologia

Velocità della luce  $c=3 \times 10^8$  m/s

$$d = \mu c$$



2002  
STOP!!!

# Soluzione B

## ▶ In generale

- una rappresentazione semplificata del tempo richiesto dalla risoluzione numerica di un problema è:

$$\tau = k \cdot T(n) \cdot \mu$$

$T(N)$ = complessità di tempo  
dell'algoritmo

Dipendenza dall'algoritmo

Si può ridurre riorganizzando l'algoritmo

E' possibile dimostrare  
*(teoria della complessità degli algoritmi)*  
che per alcune classi di problemi  
esistono algoritmi con complessità di  
tempo minima  
**(algoritmi ottimali)**

# Soluzione B

## ▶ In generale

- una rappresentazione semplificata del tempo richiesto dalla risoluzione numerica di un problema è:

$$\tau = k \cdot T(n) \cdot \mu$$

$T(N)$ = complessità di tempo  
dell'algoritmo

Dipendenza dall'algoritmo

Si può ridurre riorganizzando l'algoritmo

Progettare un algoritmo 10x più  
veloce equivale a costruire un  
calcolatore 10x volte più potente

# Soluzione B

## In generale

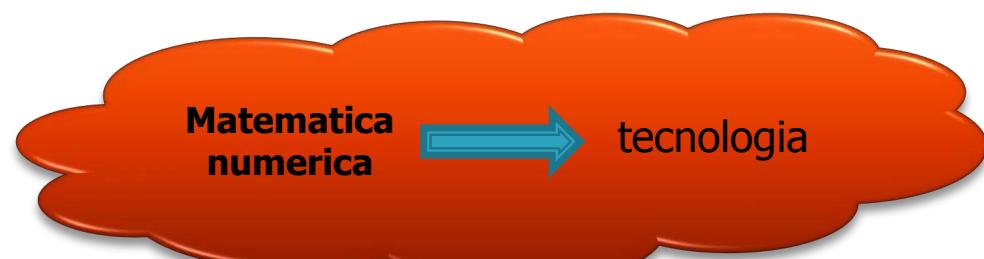
- una rappresentazione semplificata del tempo richiesto dalla risoluzione numerica di un problema è:

$$\tau = k \cdot T(n) \cdot \mu$$

$T(N)$ = complessità di tempo  
dell'algoritmo

Dipendenza dall'algoritmo

Si può ridurre riorganizzando l'algoritmo



# Soluzione C: CALCOLO PARALLELO

## ▶ In generale

- una rappresentazione semplificata del tempo richiesto dalla risoluzione numerica di un problema è:

$$\tau = k \cdot T(n) \cdot \mu$$

Decomporre un problema di dimensione N  
in P sottoproblemi di dimensione N/P  
e risolverli contemporaneamente su più calcolatori

# Soluzione C: CALCOLO PARALLELO

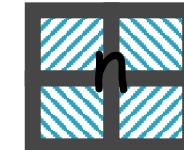
## ▶ In generale

- una rappresentazione semplificata del tempo richiesto dalla risoluzione numerica di un problema è:

$$\tau = k \cdot T(n) \cdot \mu$$

Tempo d'esecuzione per un problema di dimensione n:  
 $T(n)$

Decomporre un problema di dimensione N  
in P sottoproblemi di dimensione  $N/P$   
e risolverli contemporaneamente su più calcolatori



# Soluzione C: CALCOLO PARALLELO

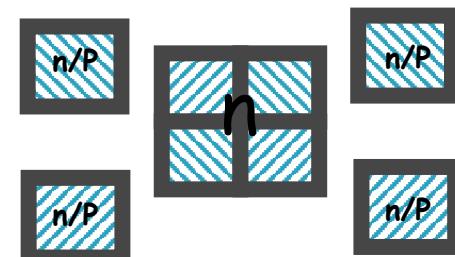
## ▶ In generale

- una rappresentazione semplificata del tempo richiesto dalla risoluzione numerica di un problema è:

$$\tau = k \cdot T(n) \cdot \mu$$

Decomporre un problema di dimensione N  
in P sottoproblemi di dimensione N/P  
e risolverli contemporaneamente su più calcolatori

Tempo d'esecuzione per un problema di dimensione n/P:  
 $T(n/P)$



# Soluzione C: CALCOLO PARALLELO

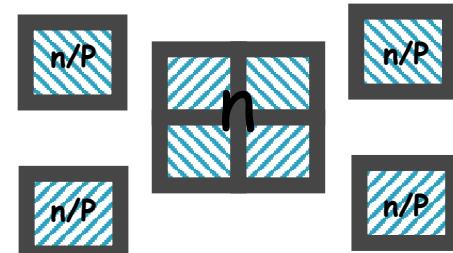
## ▶ In generale

- una rappresentazione semplificata del tempo richiesto dalla risoluzione numerica di un problema è:

$$\tau = k \cdot T(n) \cdot \mu$$

La quantità che meglio esprime l'importanza della decomposizione del problema è lo **SCALE UP**

$$\frac{T(n)}{P \cdot T\left(\frac{n}{P}\right)}$$



Decomporre un problema di dimensione N  
in P sottoproblemi di dimensione N/P  
e risolverli contemporaneamente su più calcolatori

# Soluzione C: CALCOLO PARALLELO

## ▶ In generale

- una rappresentazione semplificata del tempo richiesto dalla risoluzione numerica di un problema è:

$$\tau = k \cdot T(n) \cdot \mu$$

Decomporre un problema di dimensione N  
in P sottoproblemi di dimensione N/P  
e risolverli contemporaneamente su più calcolatori



CALCOLO PARALLELO

# Soluzione C: CALCOLO PARALLELO

## ▶ In generale

- una rappresentazione semplificata del tempo richiesto dalla risoluzione numerica di un problema è:

$$\tau = k \cdot T(n) \cdot \mu$$

Decomporre un problema di dimensione N  
in P sottoproblemi di dimensione N/P  
e risolverli contemporaneamente su più calcolatori

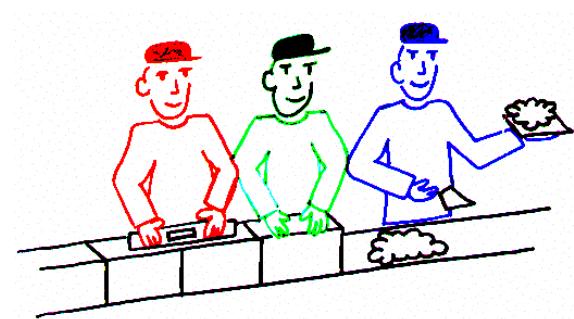
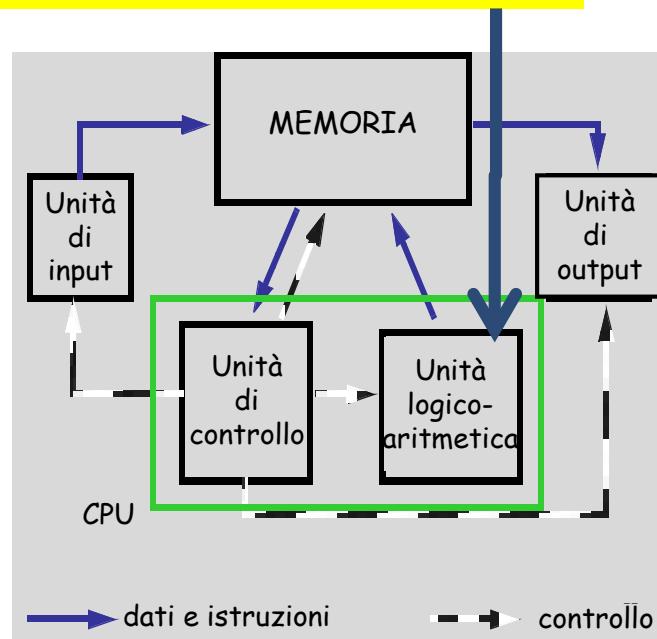
Sviluppo di nuovi strumenti  
computazionali  
Hardware/software/algoritmi

# Introduzione al corso

## »» Tipi di parallelismo

# Tipi di parallelismo

- ▶ **Parallelismo Temporale**
  - ▶ Tecnica della catena di montaggio (pipeline)
  - ▶ I tre operai eseguono contemporaneamente fasi successive dello stesso lavoro
- ▶ **In un CALCOLATORE:** a livello di ALU



Parallelismo  
*on-chip*

# Unità logico aritmetica

somma di 100 numeri floating point

$$c_i = a_i + b_i \quad i=1,100$$

L'unità funzionale per l'addizione floating point è divisa in segmenti

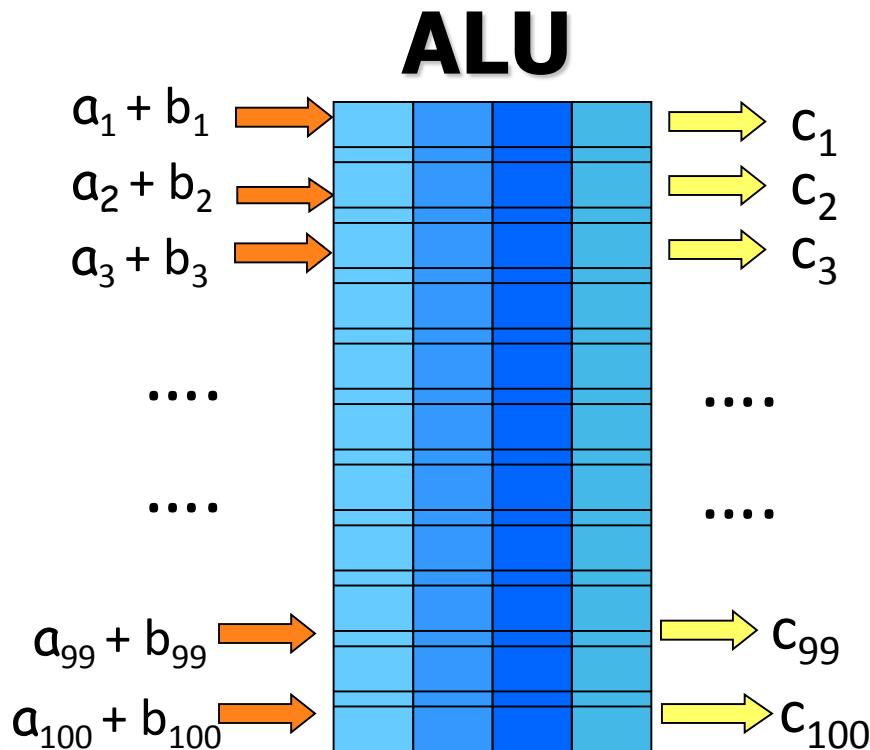
Ciascun segmento è preposto alla esecuzione di una fase dell'operazione, ad esempio per la somma:



# Unità logico aritmetica

somma di 100 numeri floating point

$$c_i = a_i + b_i \quad i=1, \dots, 100$$



## Unità tradizionale

$$N=100, 4 \text{ fasi}$$

$$t = \text{tempo di una fase}$$

$$T=N*4t$$

# Unità logico aritmetica pipelined

## somma di 100 numeri floating point

$$c_i = a_i + b_i \quad i=1,\dots,100$$

N=100, 4 fasi  
t= tempo di una fase

T=

# Unità logico aritmetica pipelined

## somma di 100 numeri floating point

$$c_i = a_i + b_i \quad i=1,\dots,100$$

N=100, 4 fasi  
t= tempo di una fase

T =

# Unità logico aritmetica pipelined

## somma di 100 numeri floating point

$$c_i = a_i + b_i \quad i=1,\dots,100$$

N=100, 4 fasi  
t= tempo di una fase

T =

# Unità logico aritmetica pipelined

## somma di 100 numeri floating point

$$c_i = a_i + b_i \quad i=1,\dots,100$$

N=100, 4 fasi  
t= tempo di una fase

T=

# Unità logico aritmetica pipelined

## somma di 100 numeri floating point

$$c_i = a_i + b_i \quad i=1,\dots,100$$

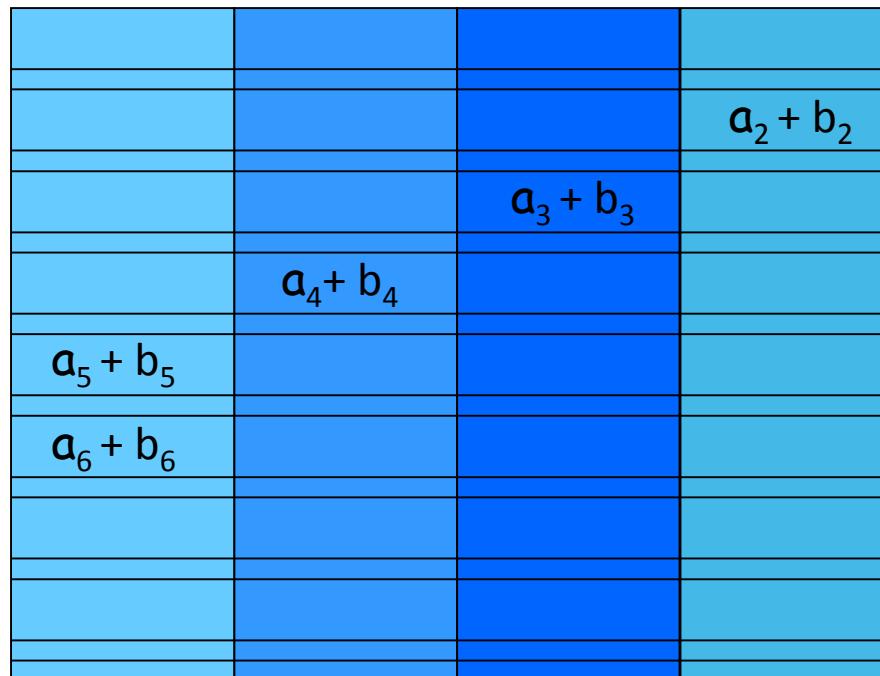
N=100, 4 fasi  
t= tempo di una fase

$$T=4t+$$

# Unità logico aritmetica pipelined

somma di 100 numeri floating point

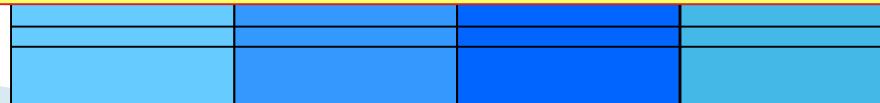
$$c_i = a_i + b_i \quad i=1, \dots, 100$$



N=100, 4 fasi  
t= tempo di una fase

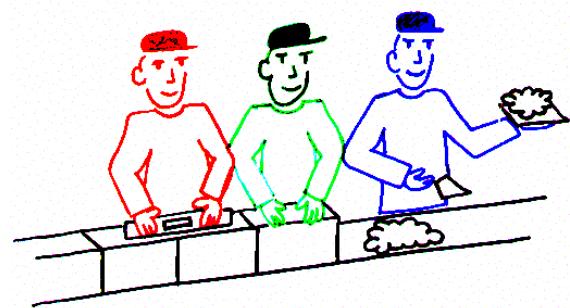
$$T=4t+(N-1)t$$

A regime sono attivi tutti i segmenti contemporaneamente



# Tipi di parallelismo

- ▶ **Parallelismo Temporale**
  - ▶ Tecnica della catena di montaggio (pipeline)
  - ▶ I tre operai eseguono contemporaneamente fasi successive dello stesso lavoro
- ▶ **In un CALCOLATORE:** a livello di ALU
- ▶ Architetture RISC, processori vettoriali



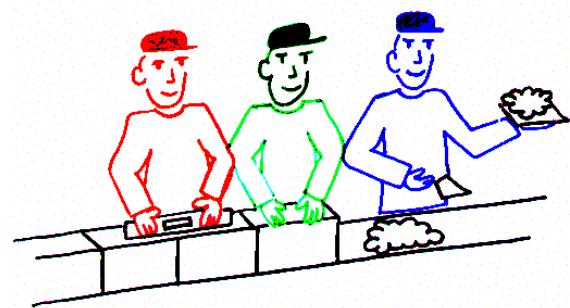
il primo “supercomputer” della storia  
CRAY 1(Los Alamos Laboratory, 1976)

133 Mflops  
8 registri scalari  
8 registri vettoriali



# Tipi di parallelismo

- ▶ **Parallelismo Temporale**
  - ▶ Tecnica della catena di montaggio (pipeline)
  - ▶ I tre operai eseguono contemporaneamente fasi successive dello stesso lavoro
- ▶ **In un CALCOLATORE:** a livello di ALU
- ▶ Architetture RISC, processori vettoriali

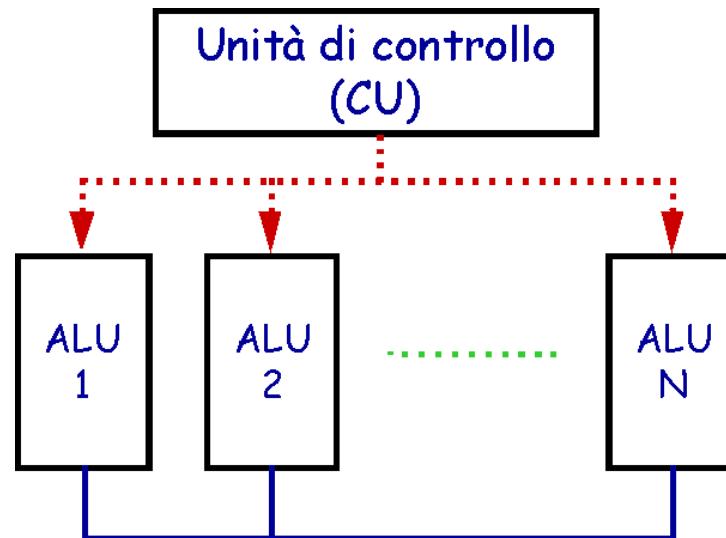


Attualmente  
**tutti i processori**  
utilizzano una struttura a pipeline per  
migliorare le loro prestazioni.



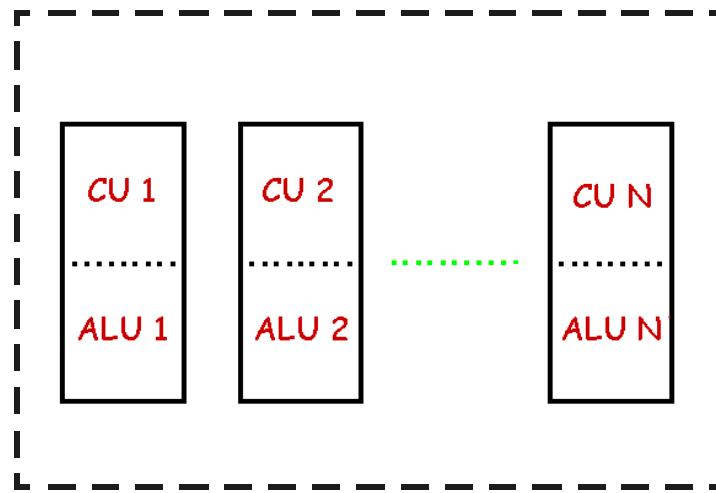
# Tipi di parallelismo

- ▶ **Parallelismo Spaziale**
  - ▶ I tre operai eseguono contemporaneamente la stessa azione su mattoni diversi
- ▶ **In un CALCOLATORE:**
  - più ALU per la stessa CU



# Tipi di parallelismo

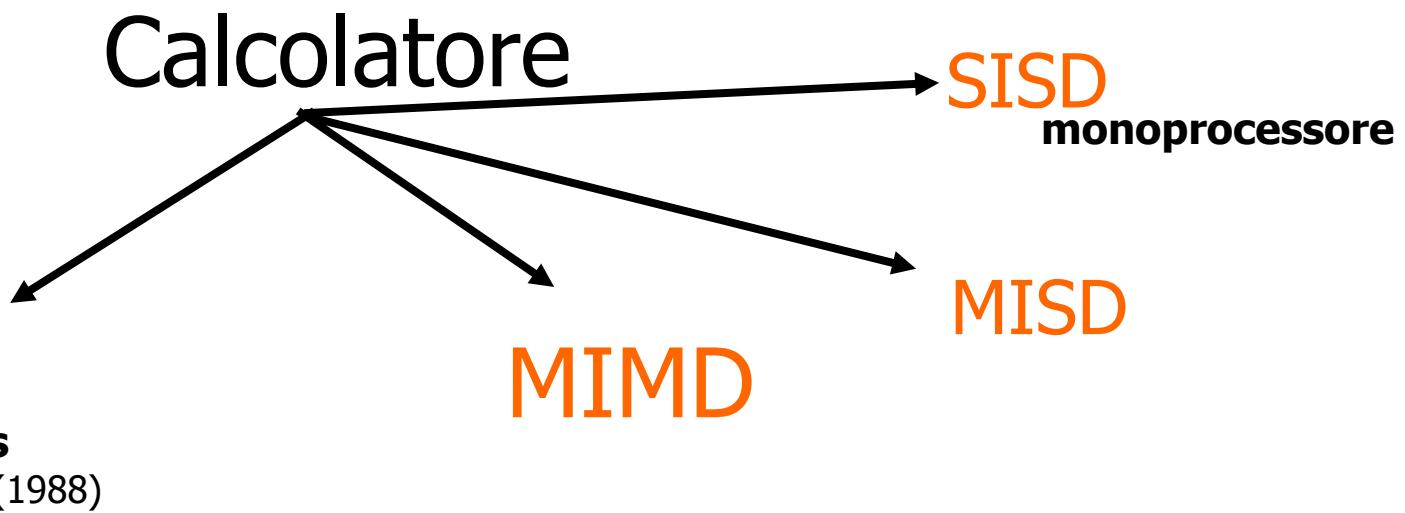
- ▶ **Parallelismo Asincrono**
  - ▶ I tre operai eseguono contemporaneamente azioni diverse su parti diverse
- ▶ **In un CALCOLATORE:**
  - Più CPU
  - Differenti processori cooperano eseguendo istruzioni diverse su dati diversi



# Introduzione al corso

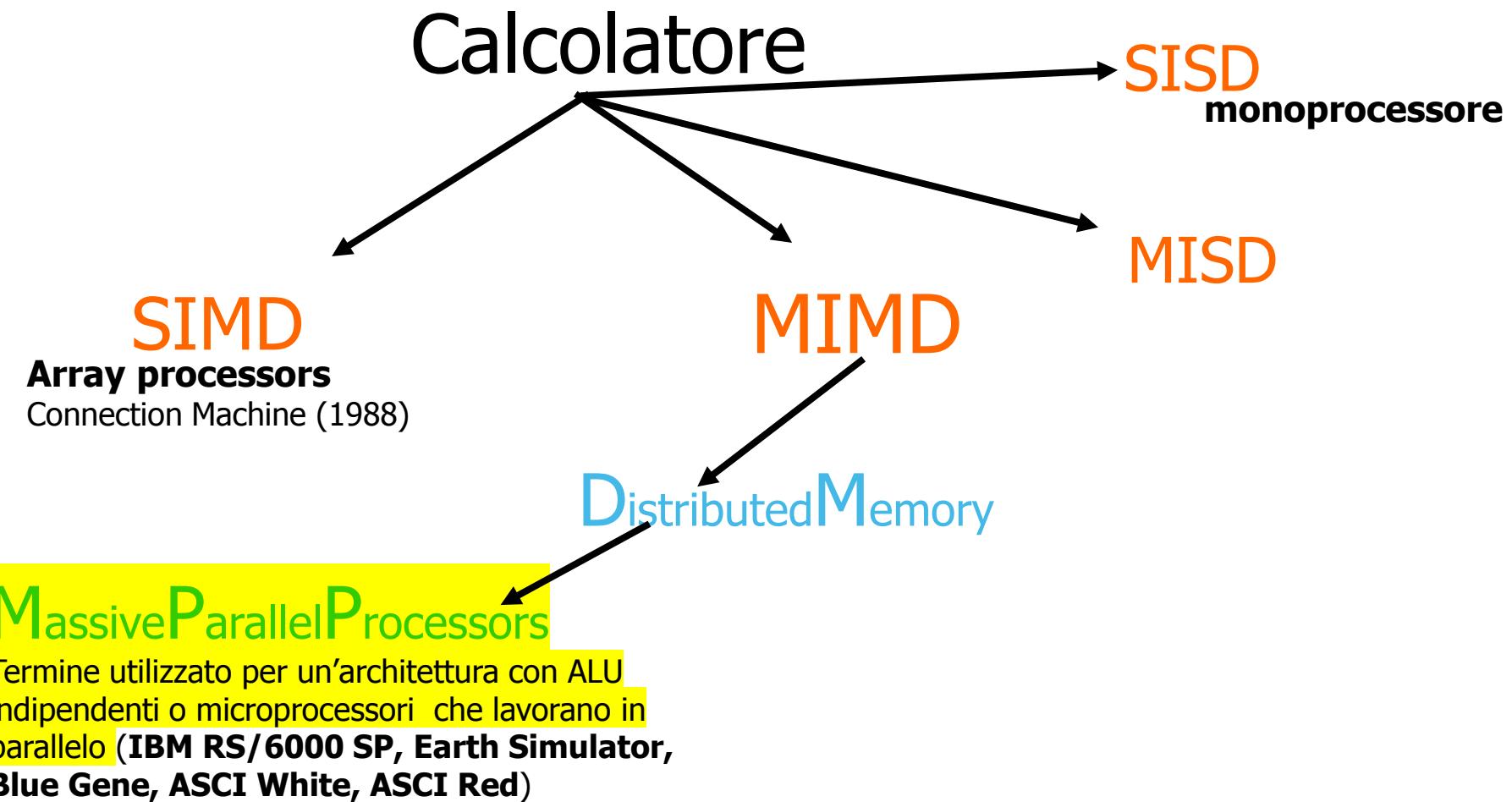
## » Tassonomia di Flynn

# Tassonomia di Flynn (dal 1965)



Negli anni '60 non erano ancora materialmente diffuse macchine diverse dai monoprocessore (SISD): all'inizio degli anni '70 erano nati alcuni array processors (che Flynn stesso definì come SIMD) e da alcune compagnie venivano le prime proposte di architetture che Flynn comprese nella sua definizione di macchina MIMD.

# Tassonomia di Flynn (dal 1965)



# Esempio MIMD–MPP

A memoria distribuita

MPP

Massive Parallel Processor:  
processori strettamente integrati,  
che danno un' immagine singola  
del sistema



ASCI Red

Architettura MPP *Intel TeraFLOPS*

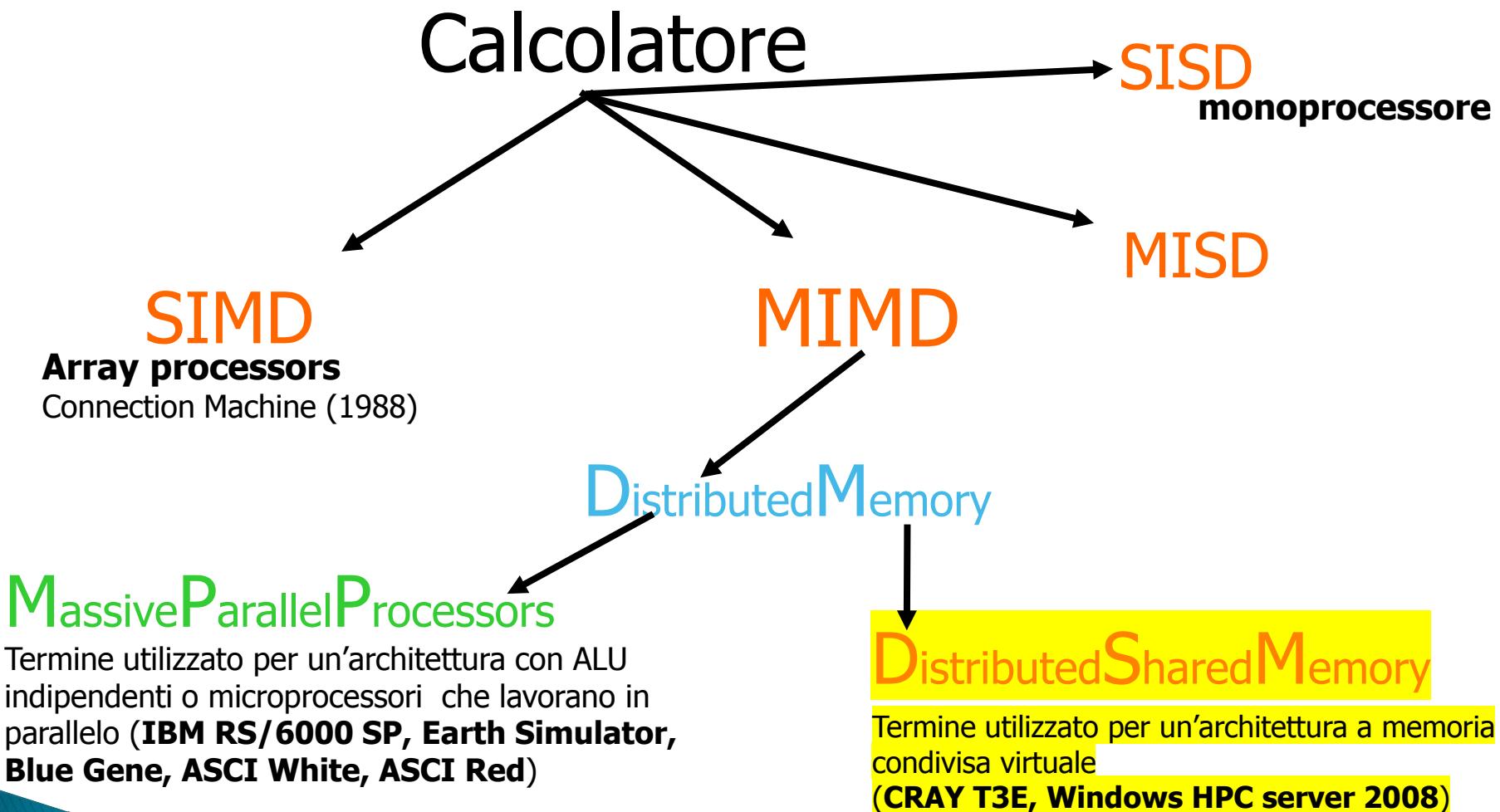
9632 processori

Topologia a griglia

Performance sostenuta: 2.3 TFlop

Picco di Performance: 3.2 TFlop

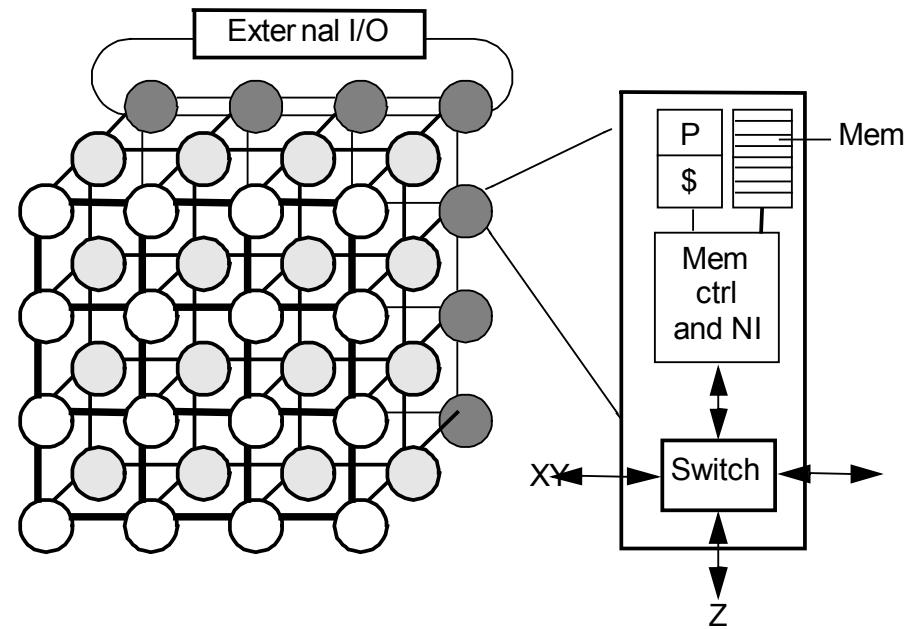
# Tassonomia di Flynn (dal 1965)



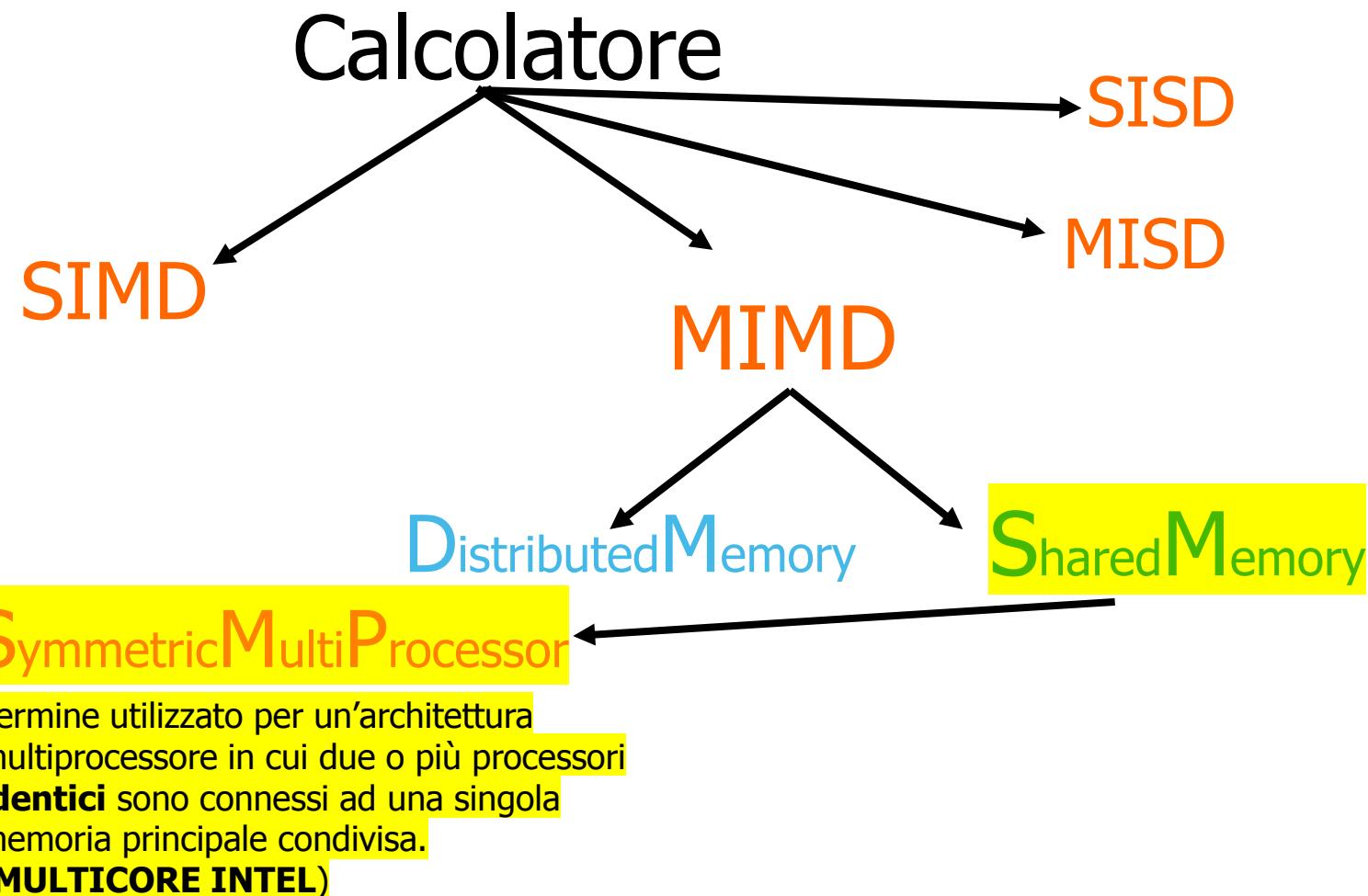
# Esempio MIMD-DSM



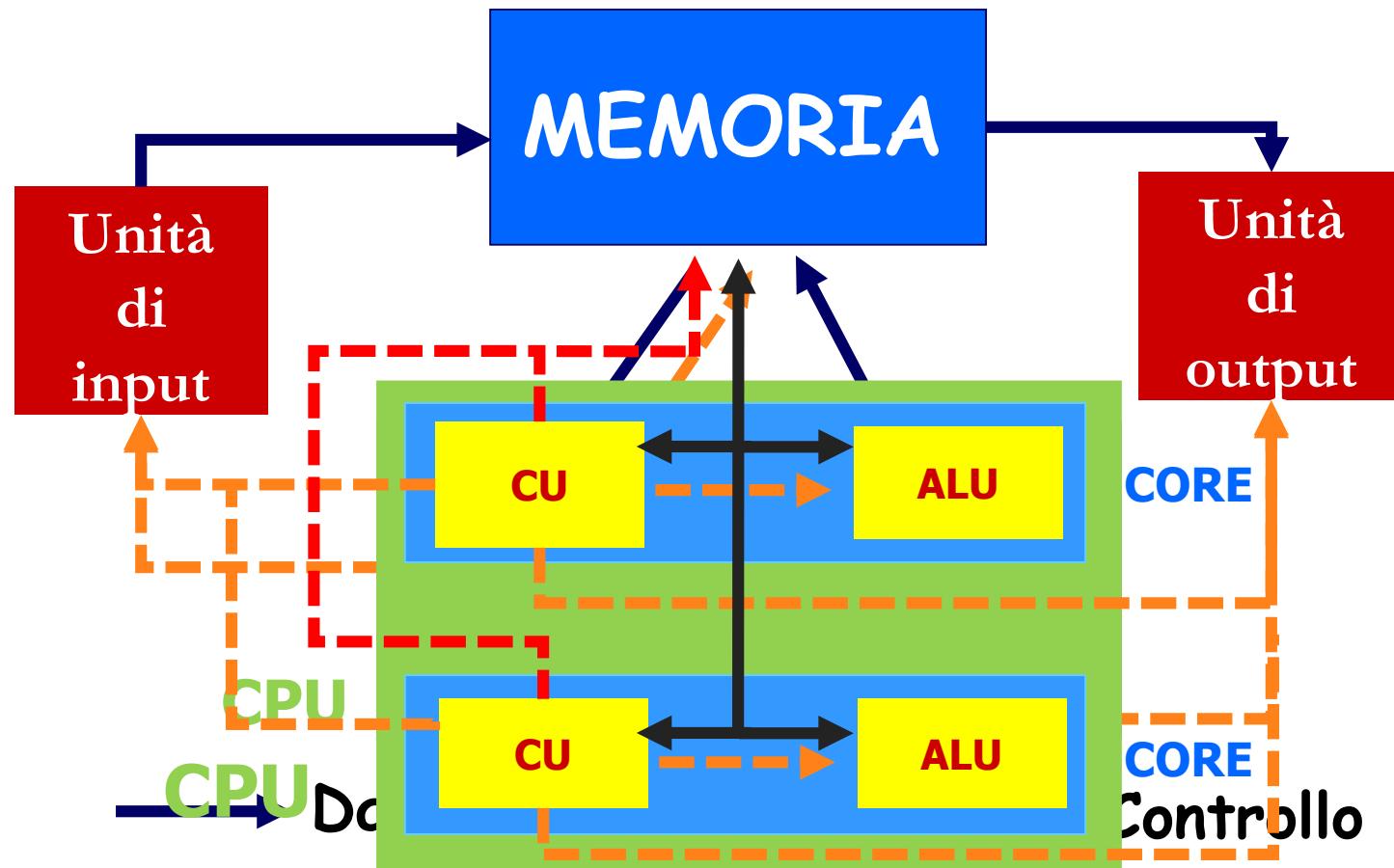
Cray T3E



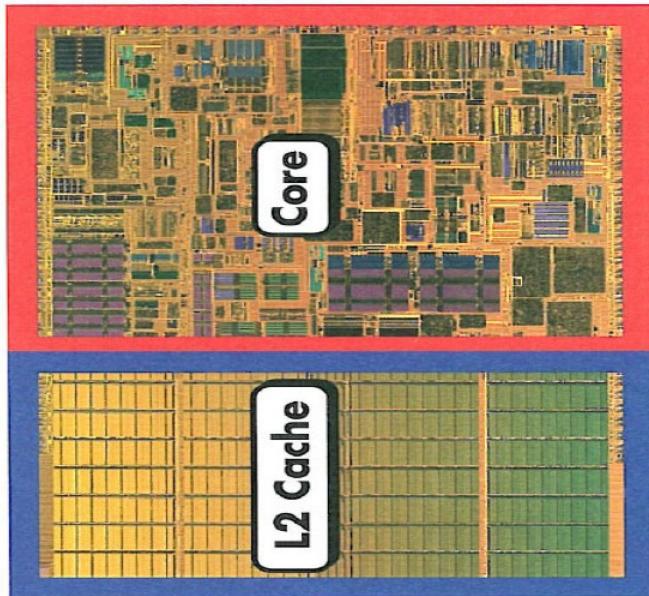
# Tassonomia di Flynn (dal 1965)



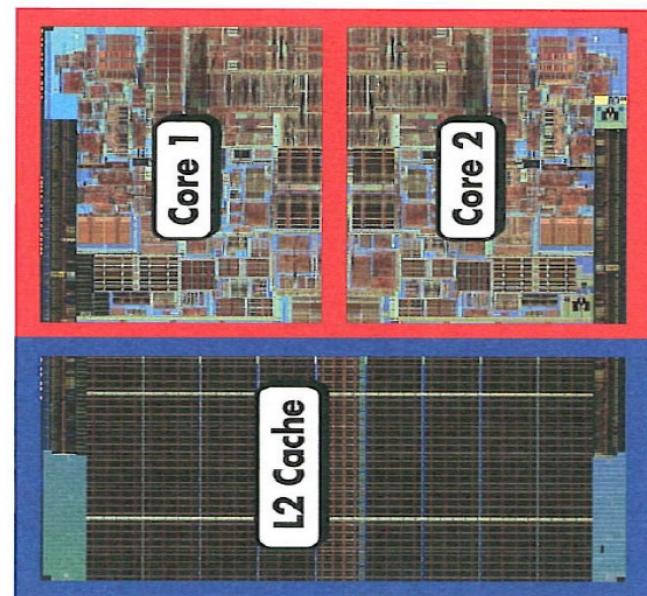
# Esempio MIMD-SMP (multicore)



# Esempio MIMD-SMP (multicore)



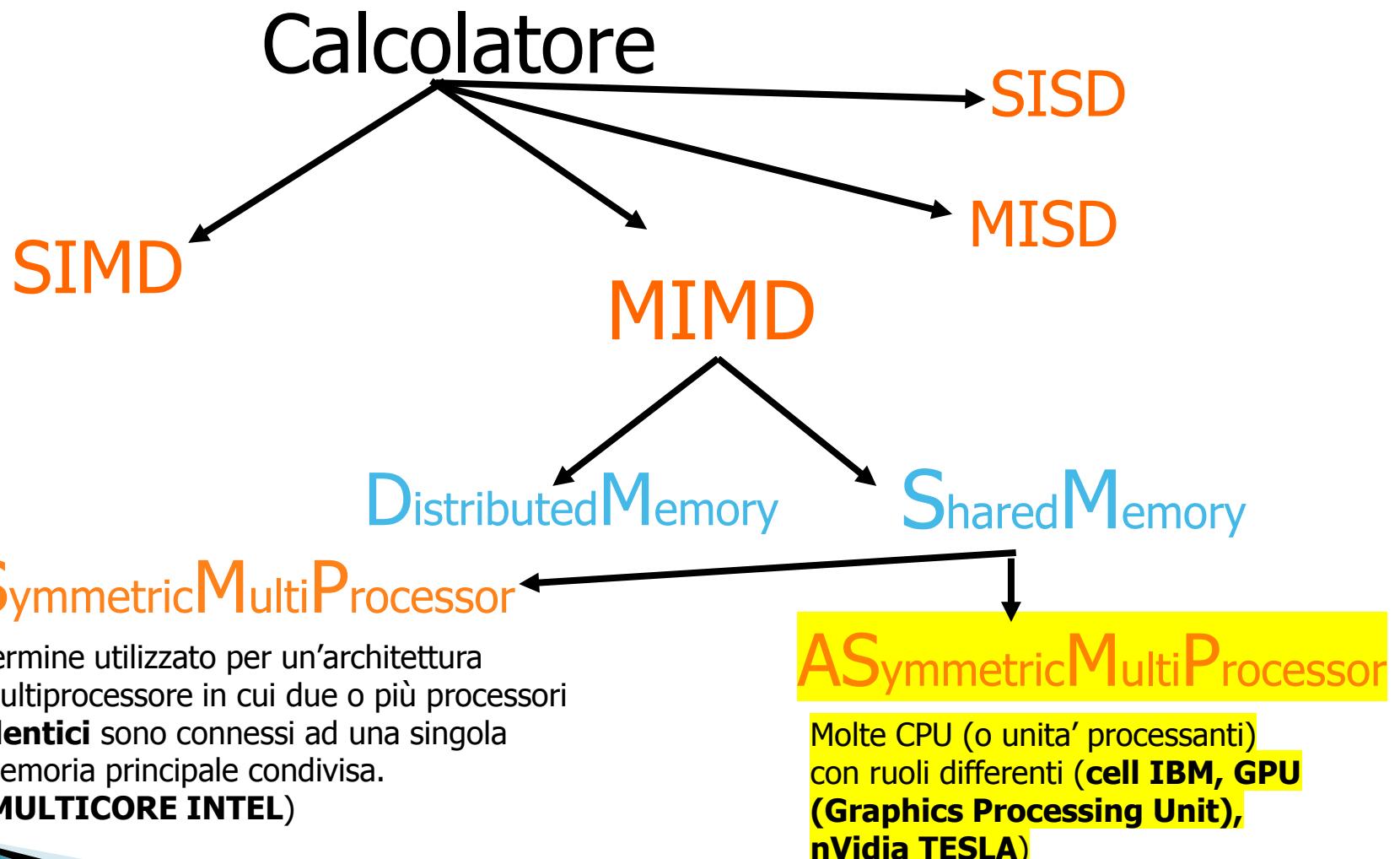
Processore Intel Pentium M (single core)



Processore Intel core duo

Parallelismo  
*on-chip*

# Tassonomia di Flynn (dal 1965)

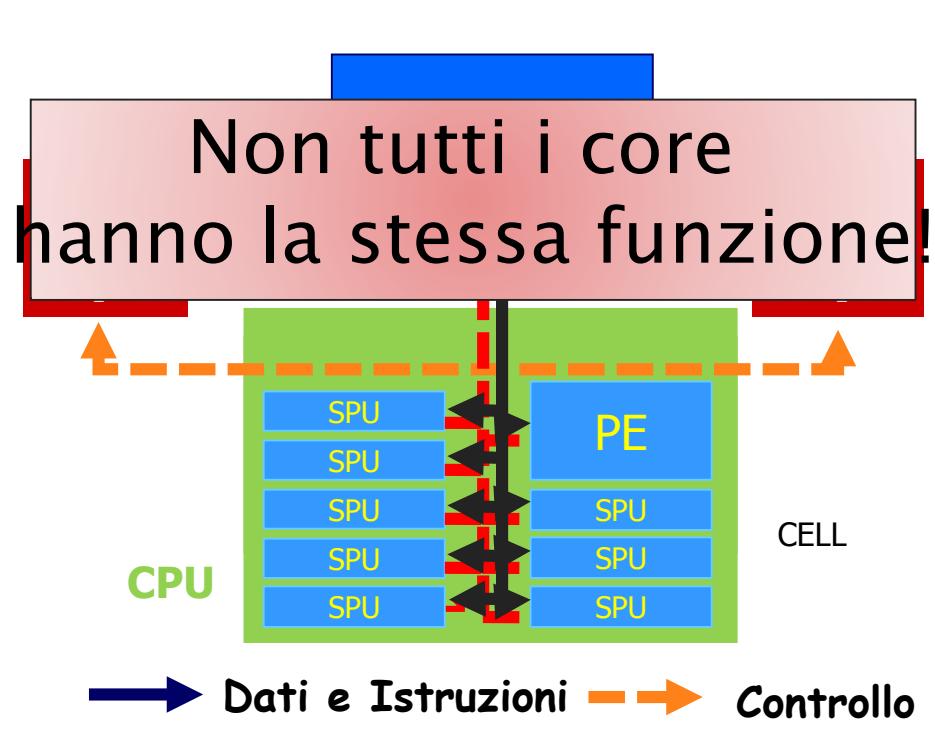
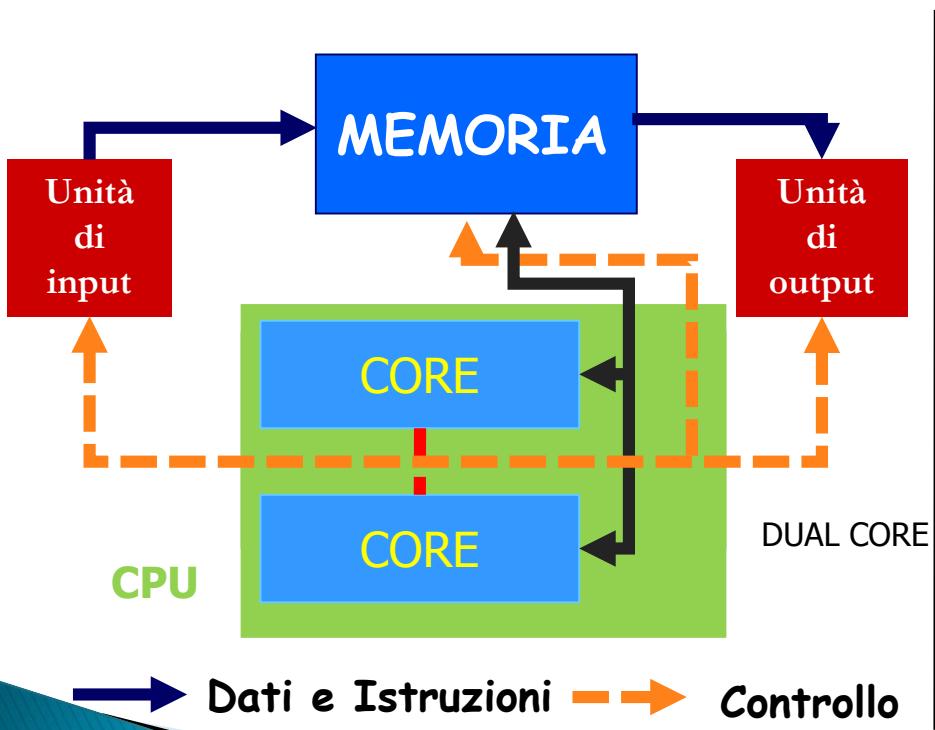


# Esempio MIMD-ASMP

## Cell IBM/Sony (PlayStation 3)

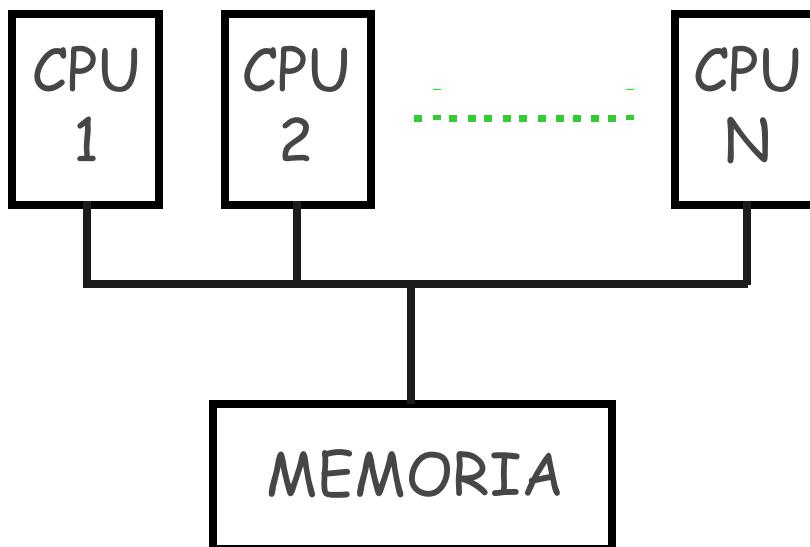
9 core

1 Processing Element (PE): architettura Risc con funzioni di controllo per gli altri 8  
8 Synergistic Processing Unit (SPU): processore vettoriale per il calcolo

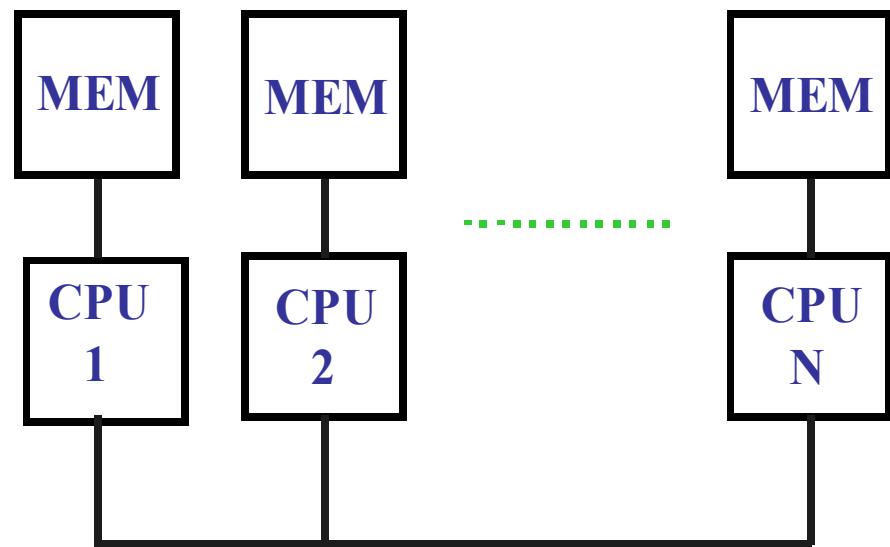


# MIMD

Calcolatori MIMD a memoria  
(shared-memory)

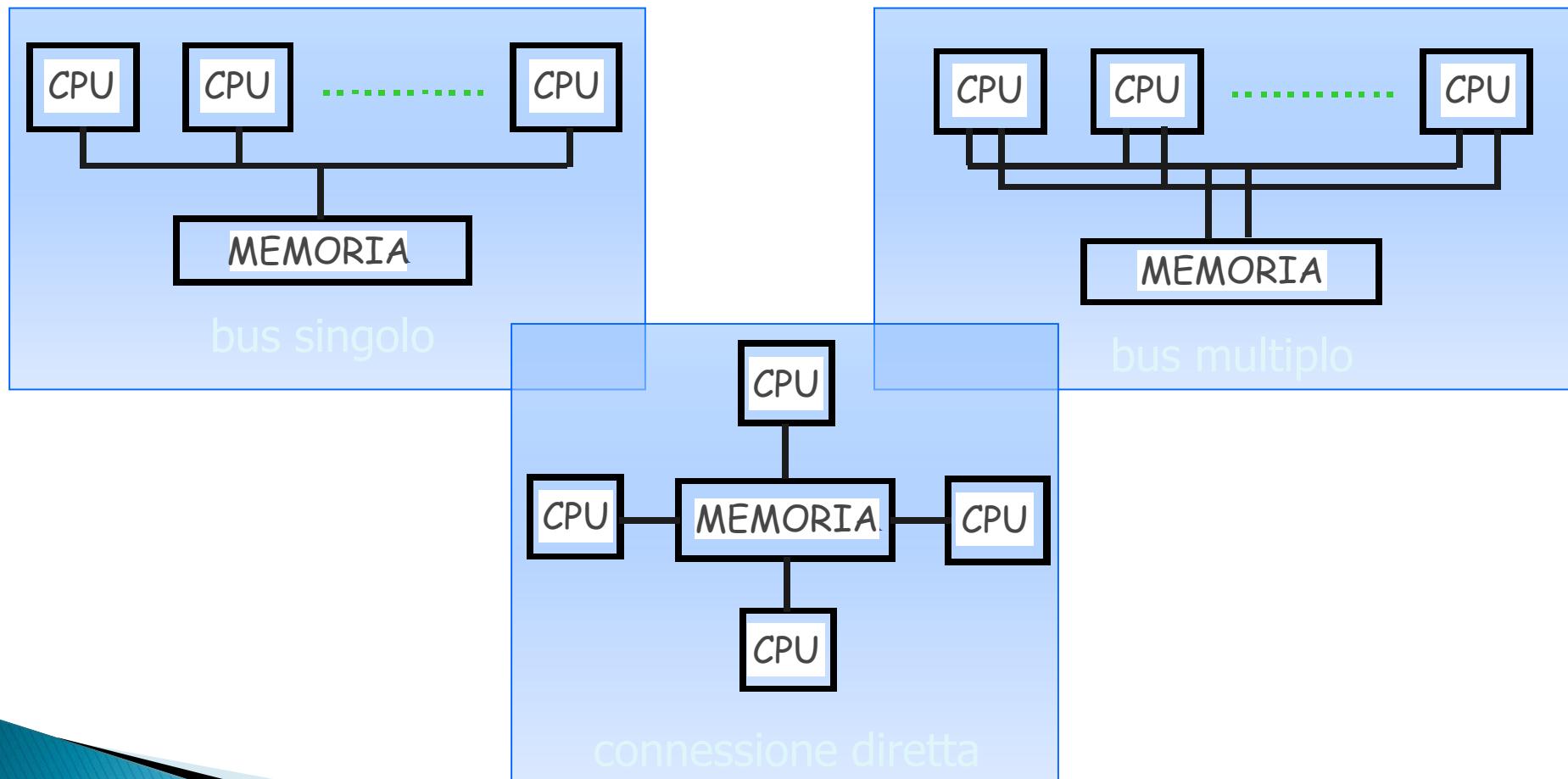


Calcolatori MIMD a memoria  
(distributed-memory)



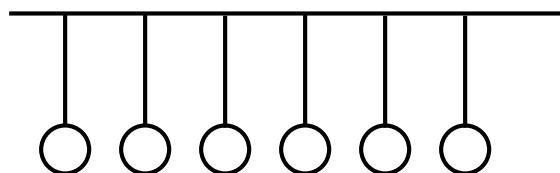
# MIMD: collegamento cpu–memoria

Calcolatori MIMD a memoria condivisa



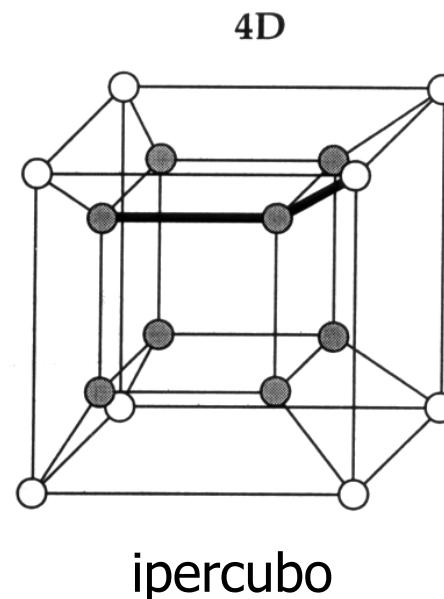
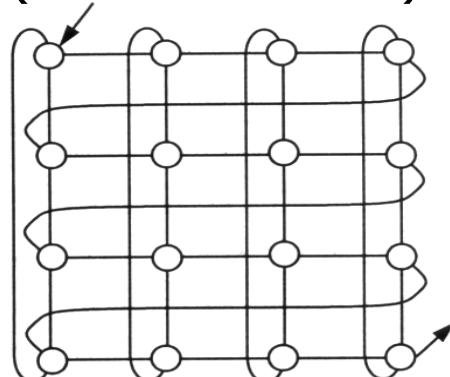
# MIMD: collegamento tra cpu

Calcolatori MIMD a memoria distribuita

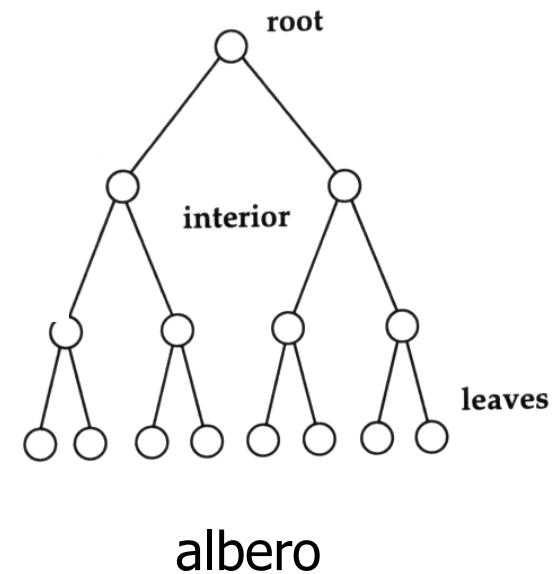


bus

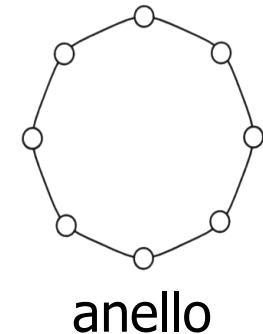
*reticolo 2d*  
(eventualm. toro)



ipercubo



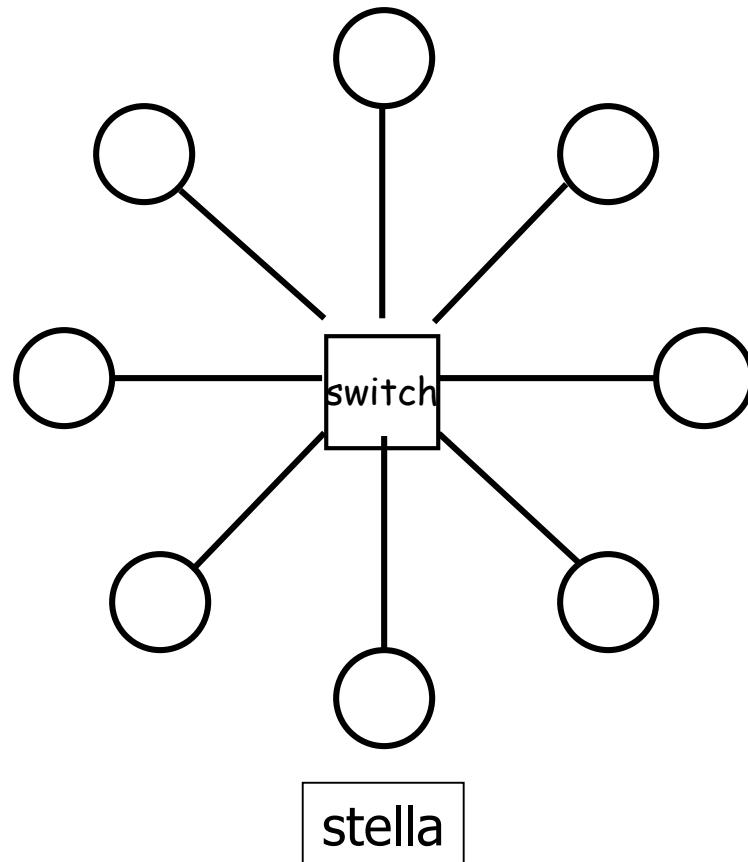
albero



anello

# MIMD: collegamento tra cpu

Calcolatori MIMD a memoria distribuita

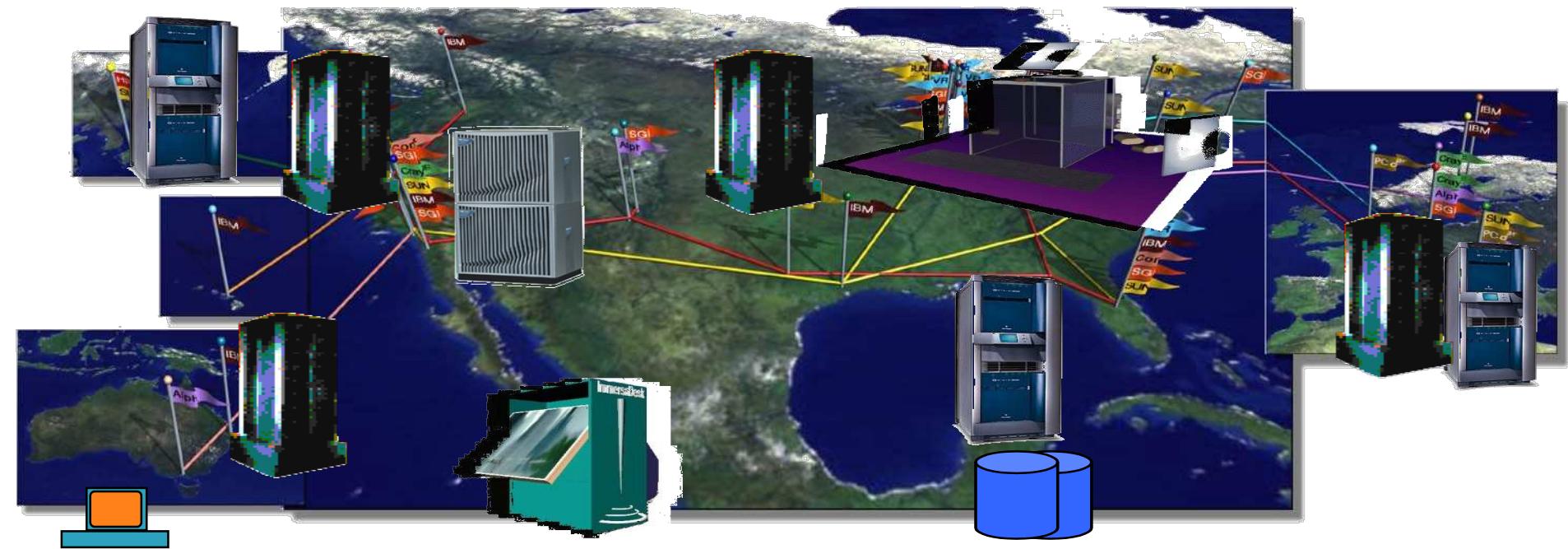


Attualmente  
**tutti i processori**  
utilizzano forme diverse di  
parallelismo, e nessun sistema in  
commercio si può definire  
puramente sequenziale.

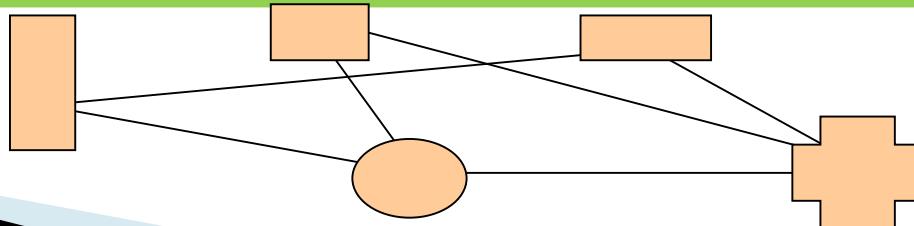
# Introduzione al corso

## » Calcolo Distribuito

# Calcolatore Distribuito

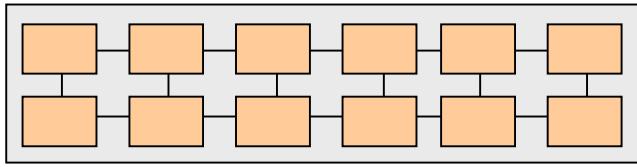


Calcolatore Distribuito (**Griglia**)

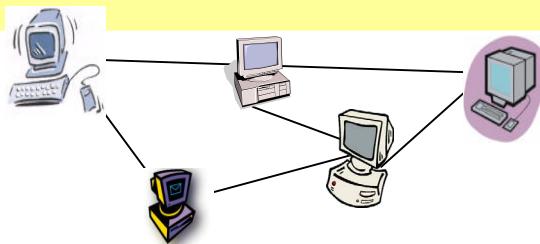


# Calcolatore Distribuito

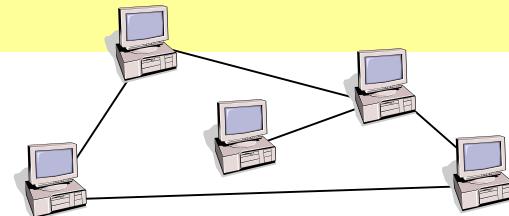
Calcolatore Parallello



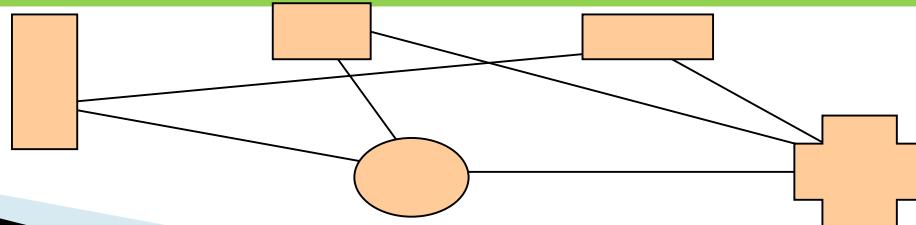
Cluster eterogeneo



Cluster omogeneo



Calcolatore Distribuito (**Griglia**)



# Ambiente di calcolo parallelo

Architetture

“ Un sistema di unità processanti

strettamente collegate

che comunicano

Reti

Risorse  
condivise

per risolvere problemi su larga scala

in maniera efficiente”

Algoritmi

Applicazioni

# Ambiente di calcolo distribuito

Architetture

“ Un sistema di unità processanti

autonome e indipendenti, fisicamente distribuite

che comunicano

Reti

Risorse  
distribuite

per risolvere problemi su larga scala

in maniera efficiente”

Algoritmi

Applicazioni

# Parallelo vs Distribuito

Calcolatore parallelo

sistema di nodi collegati  
da switch specializzati e  
dedicati

*(tightly coupled systems)*

Sistema ad arch.  
distribuita

sistema di nodi collegati da  
reti geografiche

*(loosely coupled systems)*

La differenza è nella rete di connessione

# Parallelo vs Distribuito

Calcolatore parallelo



Principale obiettivo:  
**Performance**

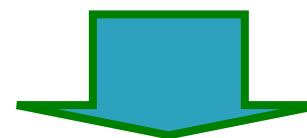


Risorse di calcolo  
omogenee

Sistema ad arch.  
distribuita



Principale obiettivo:  
**Ri-uso di risorse esistenti**



Risorse di calcolo  
eterogenee

# Parallelo vs Distribuito

“in parallel computing we decompose into parts,  
in distributed computing we assemble parts”

G.J. Fox, IEEE CiSE, 2002

“nel calcolo parallelo decomponiamo il problema,  
nel calcolo distribuito assembliamo le risorse”

# Calcolo parallelo per...

Ridurre il tempo necessario alla risoluzione computazionale  
di un problema reale



"wall - clock" time

# Calcolo distribuito per...

Riutilizzare “efficacemente” risorse hardware e software distribuite geograficamente sul territorio

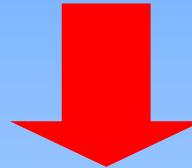


“(ri)uso efficiente delle risorse”

# Funzioni collettive

# Operazioni di riduzione

MPI possiede una classe  
di operazioni collettive chiamate  
*operazioni di riduzione.*



In ciascuna operazione di riduzione  
*tutti* i processori di un  
communicator *contribuiscono* al  
risultato di un'operazione.

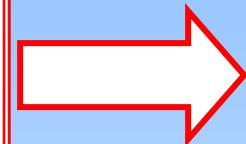
# Un semplice programma :

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{    int menum, nproc;
    int sum, sumtot;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&menum);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    sum=nproc;
    sumtot=0;
    sum+=menum;

MPI_Reduce (&sum ,&sumtot ,1 ,MPI_INT ,MPI_SUM ,0 ,
            MPI_COMM_WORLD);

    printf("Sono %d sum=%d sumtot=%d\n",menum,sum,sumtot);

    MPI_Finalize();
    return 0;
}
```



Nel programma ... :



```
MPI_Reduce(&sum,&sumtot,1,MPI_INT,MPI_SUM,  
0, MPI_COMM_WORLD);
```

- Il processore  $P_0$  otterrà la somma dei elementi **sum**, di tipo **MPI\_INT** e di dimensione **1**, distribuiti tra tutti i processori del communicator **MPI\_COMM\_WORLD**. Il risultato lo pone nella propria variabile **sumtot**.

Le operazioni globali di **Reduce** sono implementate in maniera efficiente in quanto:

- 1) Ottimizzano le comunicazioni tra i processori del communicator coinvolto. Le comunicazioni vengono eseguite seguendo uno schema ad albero.
- 2) Sfruttano la proprietà associativa e/o la proprietà commutativa.

## In Generale :

```
MPI_Reduce(void *operand, void *result,  
    int count, MPI_Datatype datatype,  
    MPI_Op op, int root,  
    MPI_Comm comm) ;
```

- Tutti i processori di **comm** combinano i propri dati memorizzati in **\*operand** utilizzando l'operazione **op**.
- Il risultato viene memorizzato in **\*result** di proprietà del processore con identificativo **root**
- **Count**, **datatype**, **comm** devono essere uguali per ogni processore di **comm**.

```
MPI_Reduce(void *operand, void *result,  
           int count, MPI_Datatype datatype,  
           MPI_Op op, int root,  
           MPI_Comm comm) ;
```

**\*operand** indirizzo dei dati su cui effettuare l'operazione.

**\*result** indirizzo del dato contenente il risultato.

**count** numero dei dati su cui effettuare l'operazione.

**datatype** tipo degli elementi da spedire.

**op** operazione effettuata.

**root** identificativo del processore che conterrà il risultato

**comm** identificativo del communicator

L'argomento **op**, che descrive l'operazione da eseguire sugli operandi **operand**, distribuiti fra i processori può essere scelto fra i seguenti valori predefiniti:

- MPI\_MAX** → Massimo di un vettore distribuito
  - MPI\_MIN** → Minimo di un vettore distribuito
  - MPI\_SUM** → Somma componente per componente fra vettori distribuiti
  - MPI\_PROD** → Prodotto componente per componente fra vettori distribuiti
- .....

# **Fine Esercitazione**

# Parallel and Distributed Computing

## Prodotto Matrice-Vettore

Materiale tratto da appunti e slides delle lezioni di *Calcolo Parallelo e Distribuito* tenute da A. Murli

e dal testo

A. Murli "Lezioni di Calcolo Parallelo", Liguori



# Parallel and Distributed Computing

## Prodotto Matrice-Vettore

Materiale tratto da appunti e slides delle lezioni di *Calcolo Parallelo e Distribuito* tenute da A. Murli

e dal testo

A. Murli "Lezioni di Calcolo Parallelo", Liguori

# PROBLEMA: Prodotto Matrice-Vettore

Progettazione  
di un algoritmo parallelo  
per architettura MIMD  
a memoria distribuita  
per il calcolo del prodotto  
di una matrice  $A$  per un vettore  $x$ :

$$Ax = y, \quad A \in \mathbb{R}^{n \times n}, \quad x, y \in \mathbb{R}^n$$

# Qual è l'algoritmo sequenziale?

Algoritmo sequenziale

**for**  $i:=0$  to  $n-1$  **do**

$y_i := 0$

**for**  $j:=0$  to  $n-1$  **do**

$y_i := y_i + a_{ij} x_j$

**endfor**

**endfor**

Prodotto Matrice-Vettore

$$Ax = y, \quad A \in \mathbb{R}^{n \times n}, \quad x, y \in \mathbb{R}^n$$

# In particolare ...

Algoritmo sequenziale

**for**  $i:=0$  to  $n-1$  **do**

$y_i := 0$

**for**  $j:=0$  to  $n-1$  **do**

$y_i := y_i + a_{ij} x_j$

**endfor**

**endfor**

Su un calcolatore tradizionale il vettore  $y$  viene "generalmente" calcolato componente per componente secondo un ordine prestabilito

L' $i$ -esimo elemento di  $y$  è il prodotto scalare della  $i$ -esima riga di  $A$  per il vettore  $x$

# Domanda

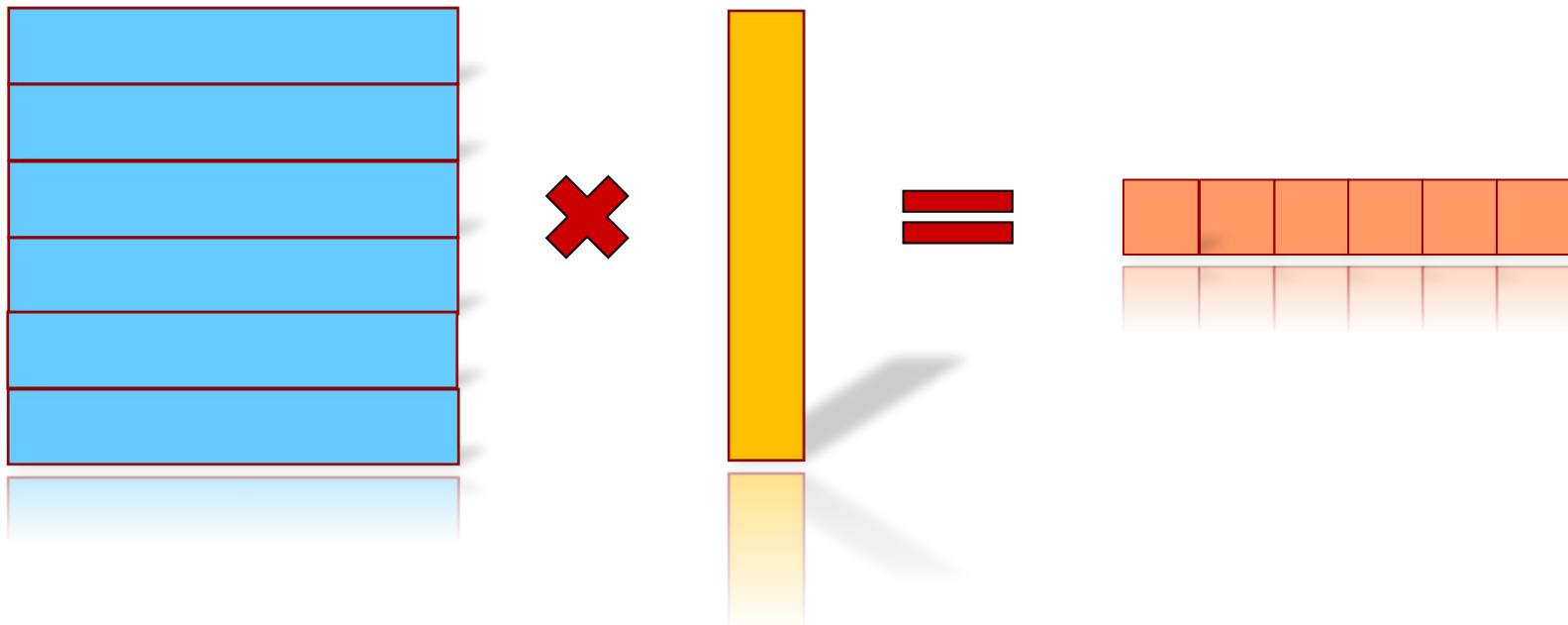
Qual è  
l'algoritmo parallelo  
?

ovvero

Come decomporre  
il problema  
Matrice-Vettore ?

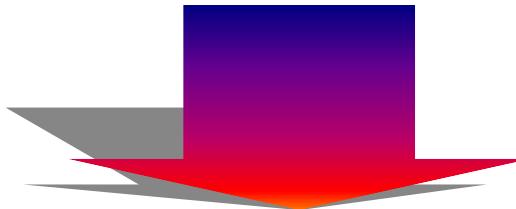
# Quali sono i sotto-problemi indipendenti?

Le componenti di  $y$  sono  
calcolate effettuando i  
prodotti scalari di  
ciascuna riga di  $A$  per il vettore  $x$



# Quali sono i sotto-problemi indipendenti?

Le componenti di  $y$  sono  
calcolate effettuando i  
prodotti scalari di  
ciascuna riga di  $A$  per il vettore  $x$



I prodotti scalari possono essere  
calcolati in maniera indipendente  
l'uno dall'altro

**IDEA!**

## **Decomposizione del problema Matrice-Vettore**



**Partizionamento della matrice A  
IN BLOCCHI**



**Riformulazione dell'algoritmo sequenziale  
"A BLOCCHI"**



**Parallelismo dell'algoritmo  
"A BLOCCHI"**

# Algoritmo a blocchi di righe

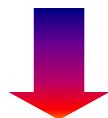
Distribuzione per blocchi di righe (I strategia)

$$\begin{bmatrix} A_0 \\ A_1 \\ \dots \\ A_{p-1} \end{bmatrix} \bullet x = y$$



$$A_i \in R^{r \times n}$$

$$i = 0, p-1$$



$$A_i x = y_i \quad || \quad \rightarrow$$

**begin**

$y := 0$

**for**  $i := 0$  to  $p-1$  **do**

$y_i := A_i x$

**endfor**

**end**

# Algoritmo a blocchi di colonne

Distribuzione per blocchi di colonne (II strategia)

$$\begin{bmatrix} A_0 & A_1 & \dots & A_{p-1} \end{bmatrix} \bullet \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{p-1} \end{bmatrix} = \sum_{i=0}^{p-1} r_i \quad A_i \in \mathbb{R}^{n \times r}, \quad x_i, r_i \in \mathbb{R}^r$$

$i = 0, p-1$

**begin**

$y := 0$

for  $i := 0$  to  $p-1$  do

$r_i := A_i x_i$

$y := y + r_i$

endfor

**end**

$\downarrow$

$A_i x_i = r_i \quad || \quad \rightarrow$

# Algoritmo a blocchi

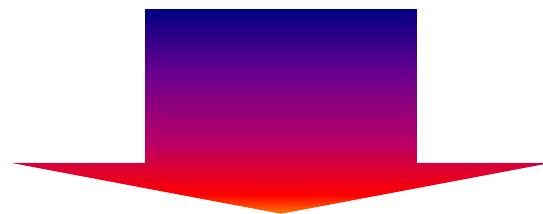
La decomposizione dei dati del problema  
in questo caso  
corrisponde ad un partizionamento in “blocchi”  
della matrice e del vettore



Il calcolo della soluzione viene ricondotto  
Al calcolo della soluzione relativa  
a ciascun “blocco”

# Qual è l'algoritmo parallelo ?

Partizionamento della matrice



Distribuzione dei blocchi  
ai processori



Algoritmo a blocchi

Algoritmo parallelo

# I STRATEGIA

Suddividiamo la  
matrice  $A$  in  
**BLOCCHI di RIGHE**

# I STRATEGIA: Esempio n= 6

Distribuzione della matrice A per blocchi di righe

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ \hline a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ \hline a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix} \bullet \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}$$

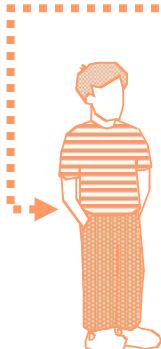


Suddividiamo la matrice in  
2 BLOCCHI di RIGHE  
ed assegniamo ogni blocco  
ad uno studente

# I STRATEGIA: Esempio n= 6

Distribuzione del vettore  $x$

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}$$



Studente 1



Studente 2

Assegniamo il vettore  $x$   
**INTERAMENTE**  
ai 2 studenti

# Domanda

Con i dati così distribuiti  
cosa può calcolare  
ciascuno studente

?

# I STRATEGIA: Esempio n= 6

Lo studente 1 può calcolare  
SOLO le prime tre componenti  
del vettore y



Studente 1

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix}$$

# I STRATEGIA: Esempio n= 6

Lo studente 2 può calcolare  
SOLO le altre tre componenti  
del vettore y

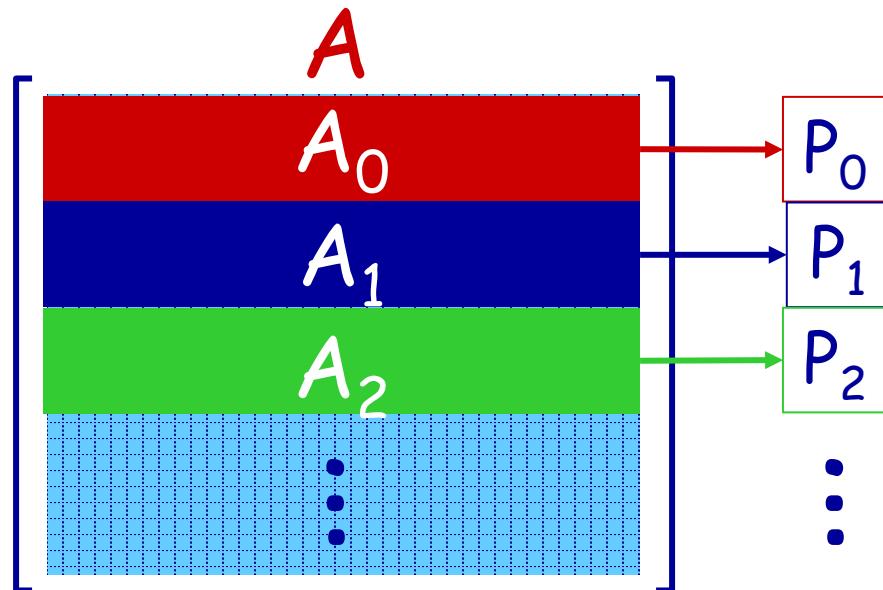
$$\begin{bmatrix} a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix} \bullet \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} y_3 \\ y_4 \\ y_5 \end{bmatrix}$$


Studente 2

# I STRATEGIA: In generale

## I passo: decomposizione del problema

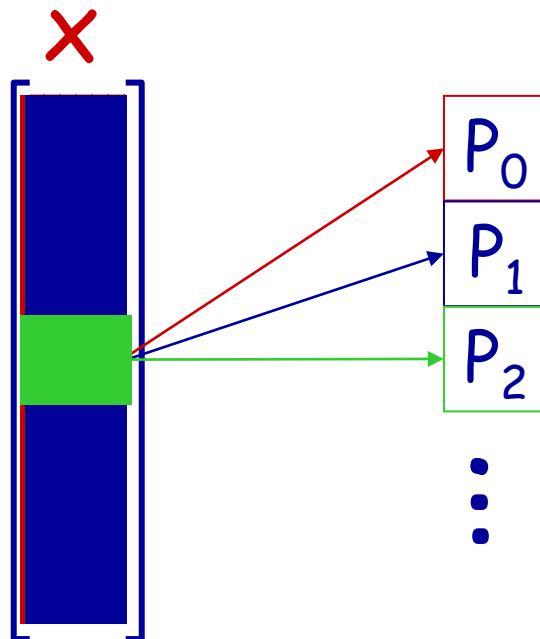
La matrice  $A$  viene distribuita  
in BLOCCHI di RIGHE  
fra p processori



# I STRATEGIA: In generale

## I passo: decomposizione del problema

Il vettore  $x$  viene assegnato INTERAMENTE  
ai  $p$  processori



# I STRATEGIA: In generale

## I passo: decomposizione del problema

Il vettore  $x$  viene assegnato INTERAMENTE  
ai  $p$  processori

Ciascun processore  $P_i$  calcola  
 $n/p$  componenti di  $y \rightarrow y_i = A_i \cdot x$

# I STRATEGIA: In generale

## II passo: risoluzione dei sottoproblemi

Il prodotto  $Ax=y$  viene decomposto  
in  $p$  prodotti del tipo

$$A_i \cdot x = y_i$$

Ciascun processore calcola  
un prodotto matrice vettore  
(di dimensione più piccola di quello assegnato).

# I strategia: caratteristiche

- ◆ non sono richieste "interazioni" tra processori

MA

- ◆ il vettore  $x$  è assegnato a tutti i processori

# Domanda

Qual è l'algoritmo parallelo  
con la I Strategia  
di decomposizione

?

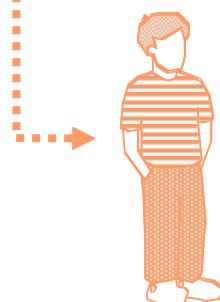
## **II STRATEGIA**

Suddividiamo  
la matrice A in  
**BLOCCHI di COLONNE**

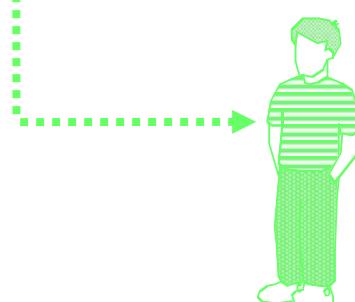
## II STRATEGIA: Esempio n= 6

Distribuzione della matrice A per blocchi di colonne

$$\begin{matrix} \boxed{\begin{matrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \\ a_{30} & a_{31} & a_{32} \\ a_{40} & a_{41} & a_{42} \\ a_{50} & a_{51} & a_{52} \end{matrix}} & \boxed{\begin{matrix} a_{03} & a_{04} & a_{05} \\ a_{13} & a_{14} & a_{15} \\ a_{23} & a_{24} & a_{25} \\ a_{33} & a_{34} & a_{35} \\ a_{43} & a_{44} & a_{45} \\ a_{53} & & a_{55} \end{matrix}} & \bullet & \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} \end{matrix}$$



Studente 1



Studente 2

Suddividiamo la matrice in  
**2 BLOCCHI** di COLONNE  
ed assegniamo ogni blocco  
ad uno studente

## II STRATEGIA: Esempio n= 6

Distribuzione del vettore  $x$

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}$$



Distribuiamo le  
Componenti del vettore  $x$   
Fra i 2 studenti

Studente 1

Studente 2

# Domanda

Con i dati così  
distribuiti  
cosa può calcolare  
ciascuno studente  
?

## II STRATEGIA: Esempio n= 6

Lo studente 1 calcola

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \\ a_{30} & a_{31} & a_{32} \\ a_{40} & a_{41} & a_{42} \\ a_{50} & a_{51} & a_{52} \end{bmatrix} \bullet \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \end{bmatrix}$$



Studente 1

Ovvero...

## II STRATEGIA: Esempio n= 6

Lo studente 1 calcola  
“un contributo” del prodotto finale



$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} a_{00}x_0 + a_{01}x_1 + a_{02}x_2 \\ a_{10}x_0 + a_{11}x_1 + a_{12}x_2 \\ a_{20}x_0 + a_{21}x_1 + a_{22}x_2 \\ a_{30}x_0 + a_{31}x_1 + a_{32}x_2 \\ a_{40}x_0 + a_{41}x_1 + a_{42}x_2 \\ a_{50}x_0 + a_{51}x_1 + a_{52}x_2 \end{bmatrix} + \begin{bmatrix} a_{03}x_3 + a_{04}x_4 + a_{05}x_5 \\ a_{13}x_3 + a_{14}x_4 + a_{15}x_5 \\ a_{23}x_3 + a_{24}x_4 + a_{25}x_5 \\ a_{33}x_3 + a_{34}x_4 + a_{35}x_5 \\ a_{43}x_3 + a_{44}x_4 + a_{45}x_5 \\ a_{53}x_3 + a_{54}x_4 + a_{55}x_5 \end{bmatrix} = \begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \end{bmatrix}$$

Studente 1

## II STRATEGIA: Esempio n= 6

Lo studente 2 calcola

$$\begin{bmatrix} a_{03} & a_{04} & a_{05} \\ a_{13} & a_{14} & a_{15} \\ a_{23} & a_{24} & a_{25} \\ a_{33} & a_{34} & a_{35} \\ a_{43} & a_{44} & a_{45} \\ a_{53} & a_{54} & a_{55} \end{bmatrix} \bullet \begin{bmatrix} x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{bmatrix}$$



Studente 2

Ovvero...

## II STRATEGIA: Esempio n= 6

Lo studente 2 calcola  
"un contributo" del prodotto finale

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + a_{03}x_3 + a_{04}x_4 + a_{05}x_5 \\ a_{10}x_0 + a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 + a_{15}x_5 \\ a_{20}x_0 + a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 + a_{25}x_5 \\ a_{30}x_0 + a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 + a_{35}x_5 \\ a_{40}x_0 + a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 + a_{45}x_5 \\ a_{50}x_0 + a_{51}x_1 + a_{52}x_2 + a_{53}x_3 + a_{54}x_4 + a_{55}x_5 \end{bmatrix}$$



Studente 2

$$= \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{bmatrix}$$

$$= \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{bmatrix}$$

# Domanda

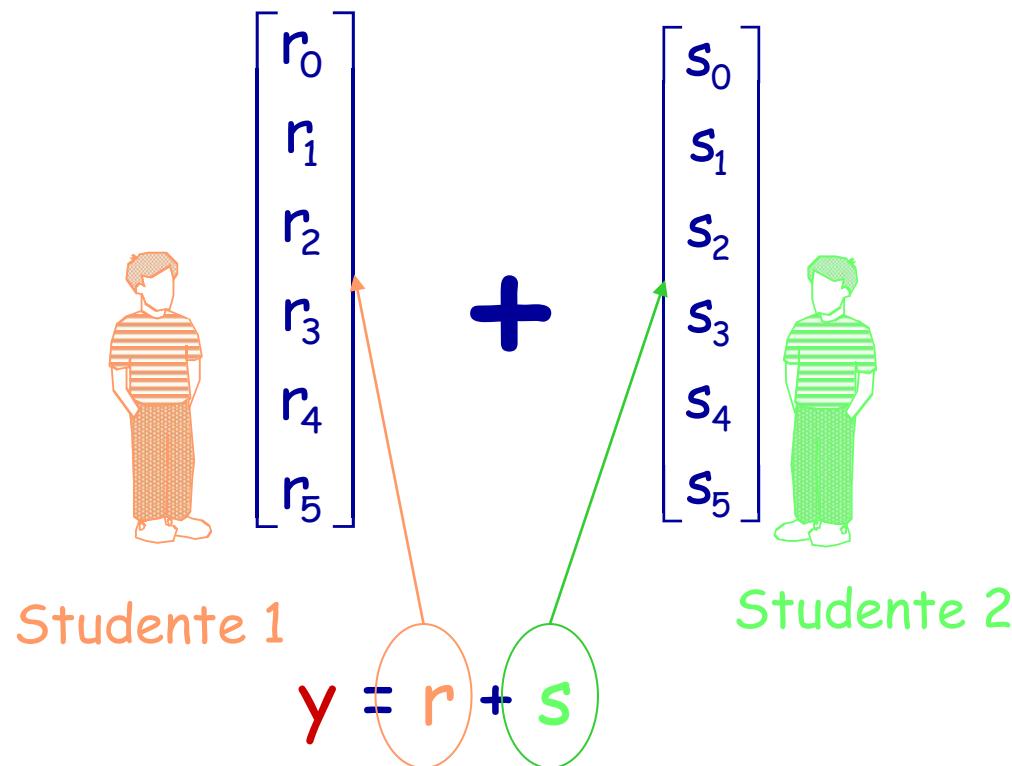
Come calcolare  
il vettore

$$y = r + s$$

?

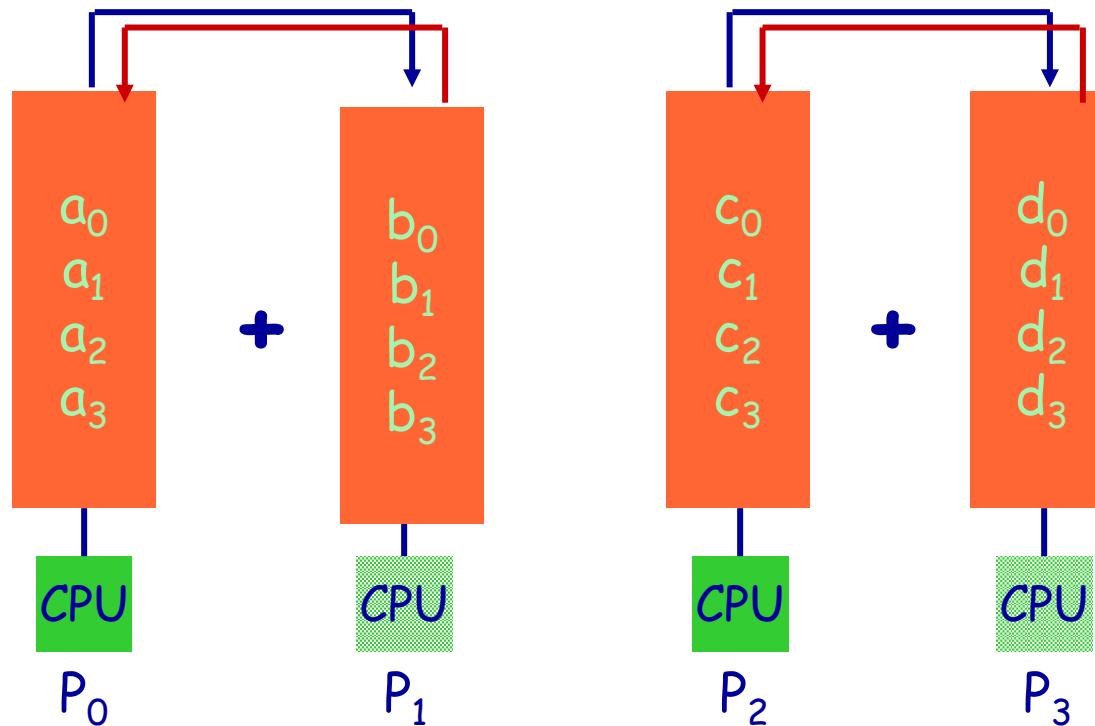
## II STRATEGIA: Esempio n= 6

Per ottenere il vettore  $y$   
gli studenti devono "interagire"  
sommando i loro risultati parziali



# Esempio N=4, p=4

I passo

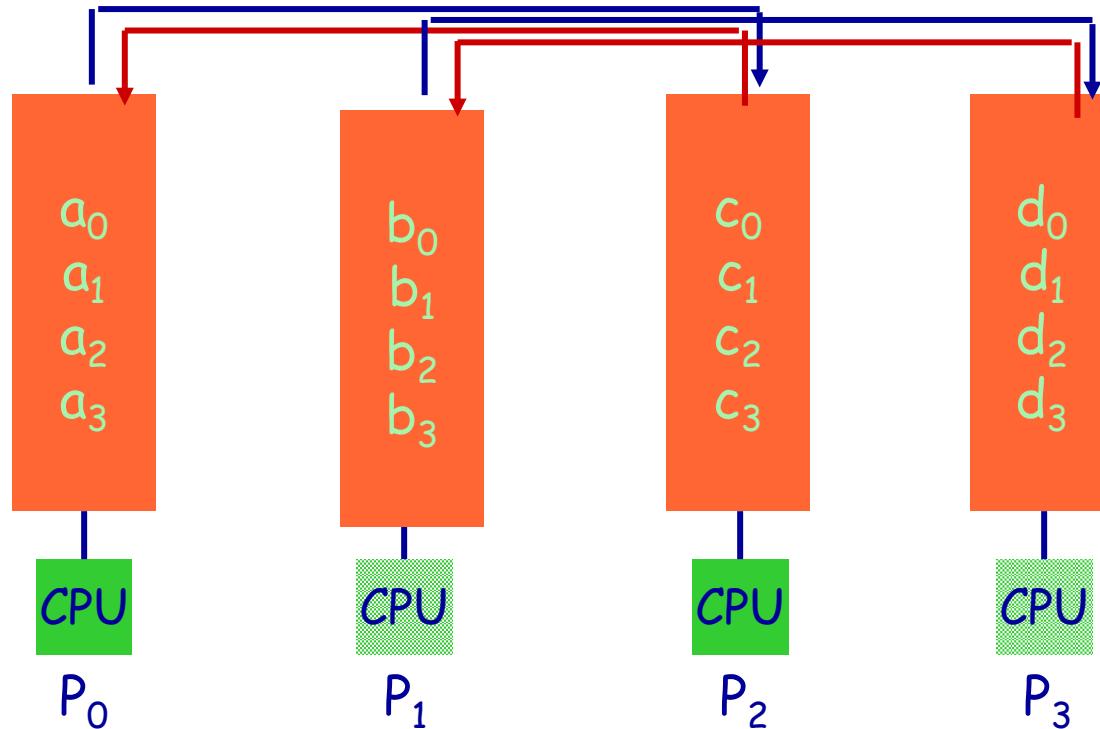


$P_0$  e  $P_1$  calcolano contemporaneamente entrambi la stessa somma di 2 vettori

$P_2$  e  $P_3$  calcolano contemporaneamente entrambi la stessa somma di 2 vettori

# Esempio N=4, p=4

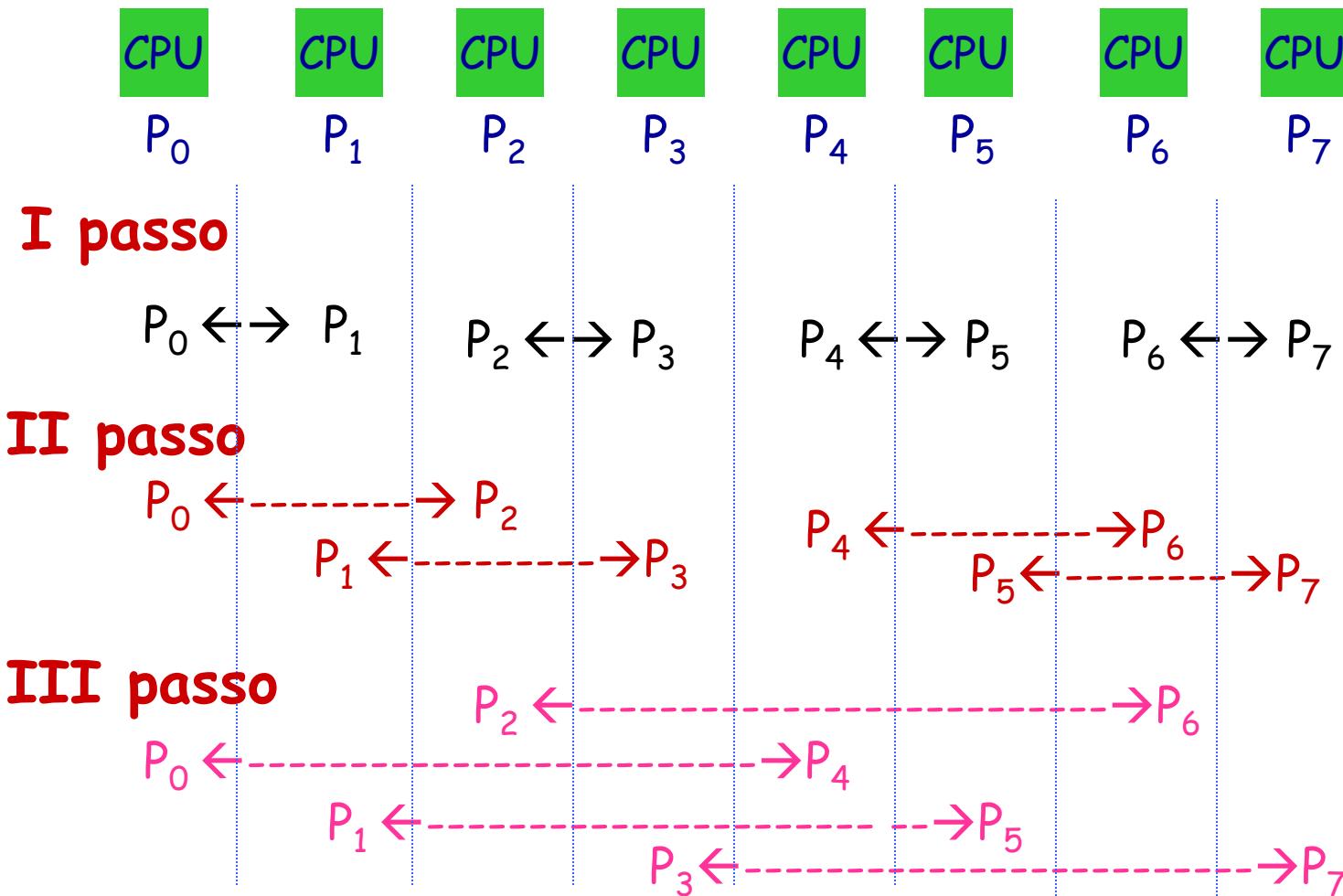
II passo



$P_0$  e  $P_2$  calcolano contemporaneamente entrambi la stessa somma di 2 vettori

$P_1$  e  $P_3$  calcolano contemporaneamente entrambi la stessa somma di 2 vettori

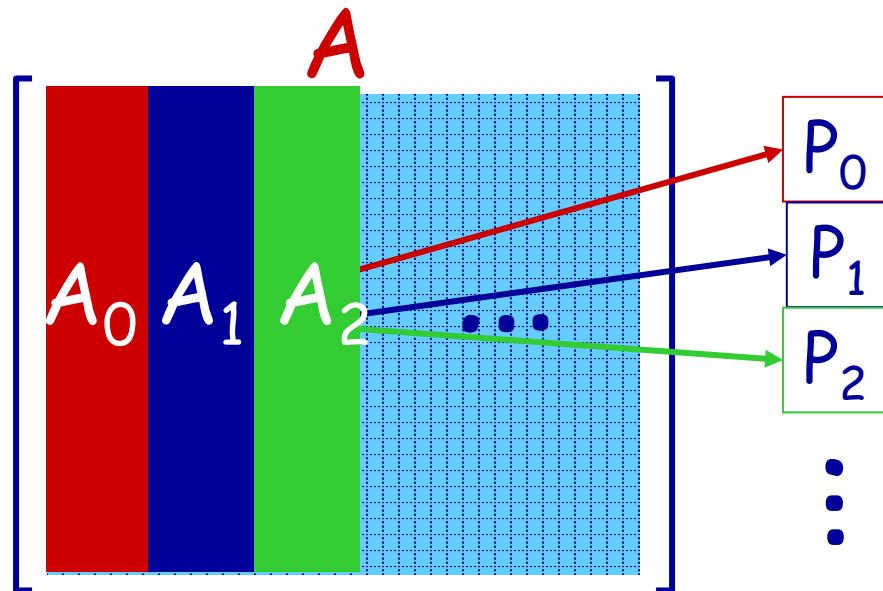
# Esempio N=8, p=8



## II STRATEGIA: In generale

### I passo: decomposizione del problema

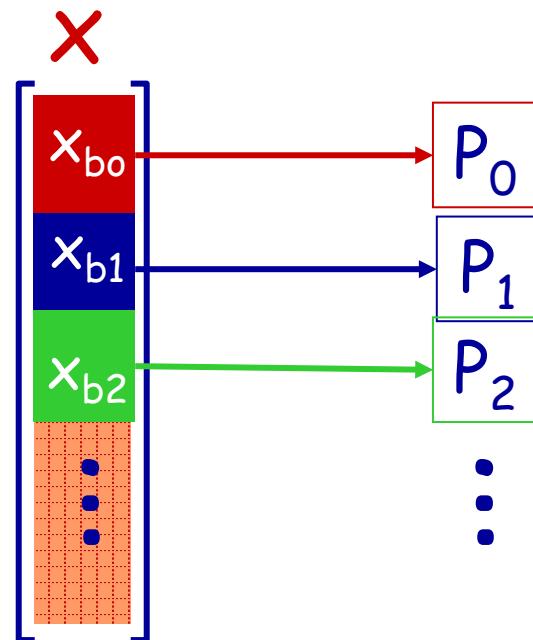
La matrice  $A$  viene distribuita  
in BLOCCHI di COLONNE  
fra p processori



## II STRATEGIA: In generale

### I passo: decomposizione del problema

Il vettore  $x$  viene distribuito  
fra i p processori



## II STRATEGIA: In generale

### II passo: risoluzione dei sottoproblemi

Il prodotto  $Ax=y$  viene decomposto  
in  $p$  prodotti del tipo

$$A_i \cdot x_{bi} = r_i \text{ dove } y = \sum_{i=0}^{p-1} r_i$$

Ciascun processore calcola  
un prodotto matrice vettore  
(di dimensione più piccola di quello assegnato).

## II strategia: caratteristiche

- ◆ Tutti i dati sono distribuiti tra processori
- ◆ In questo caso l'algoritmo parallelo è analogo a quello della somma

# Domanda

E' possibile realizzare  
un'altra decomposizione  
del problema:  
prodotto  
Matrice-Vettore

?

# Risposta: SI!

Decomposizione 1: BLOCCHI di RIGHE

+

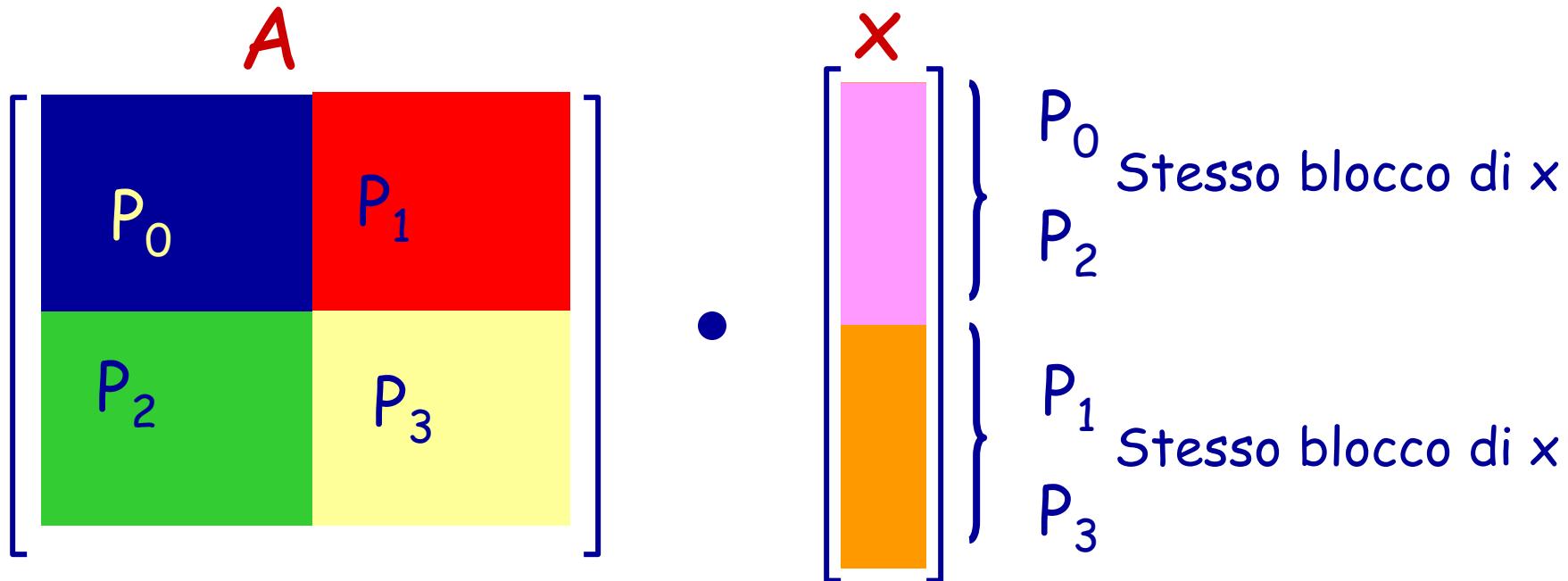
Decomposizione 2: BLOCCHI di COLONNE

=

Decomposizione 3: BLOCCHI QUADRATI

### III Strategia: Esempio (4 processori)

Distribuzione della matrice A per blocchi quadrati



Distribuzione del vettore  $x$  fra i processori

# Domanda

Ciascun processore  
quale “parte” di  $y$   
calcola

?

# Esempio N = 6 , Processori=4

P<sub>0</sub>

$$a_{00} \cdot x_0 + a_{01} \cdot x_1 + a_{02} \cdot x_2$$

$$a_{10} \cdot x_0 + a_{11} \cdot x_1 + a_{12} \cdot x_2$$

$$a_{20} \cdot x_0 + a_{21} \cdot x_1 + a_{22} \cdot x_2$$

P<sub>1</sub>

$$a_{03} \cdot x_3 + a_{04} \cdot x_4 + a_{05} \cdot x_5$$

$$a_{13} \cdot x_3 + a_{14} \cdot x_4 + a_{15} \cdot x_5$$

$$a_{23} \cdot x_3 + a_{24} \cdot x_4 + a_{25} \cdot x_5$$

P<sub>2</sub>

$$a_{30} \cdot x_0 + a_{31} \cdot x_1 + a_{32} \cdot x_2$$

$$a_{40} \cdot x_0 + a_{41} \cdot x_1 + a_{42} \cdot x_2$$

$$a_{50} \cdot x_0 + a_{51} \cdot x_1 + a_{52} \cdot x_2$$

P<sub>3</sub>

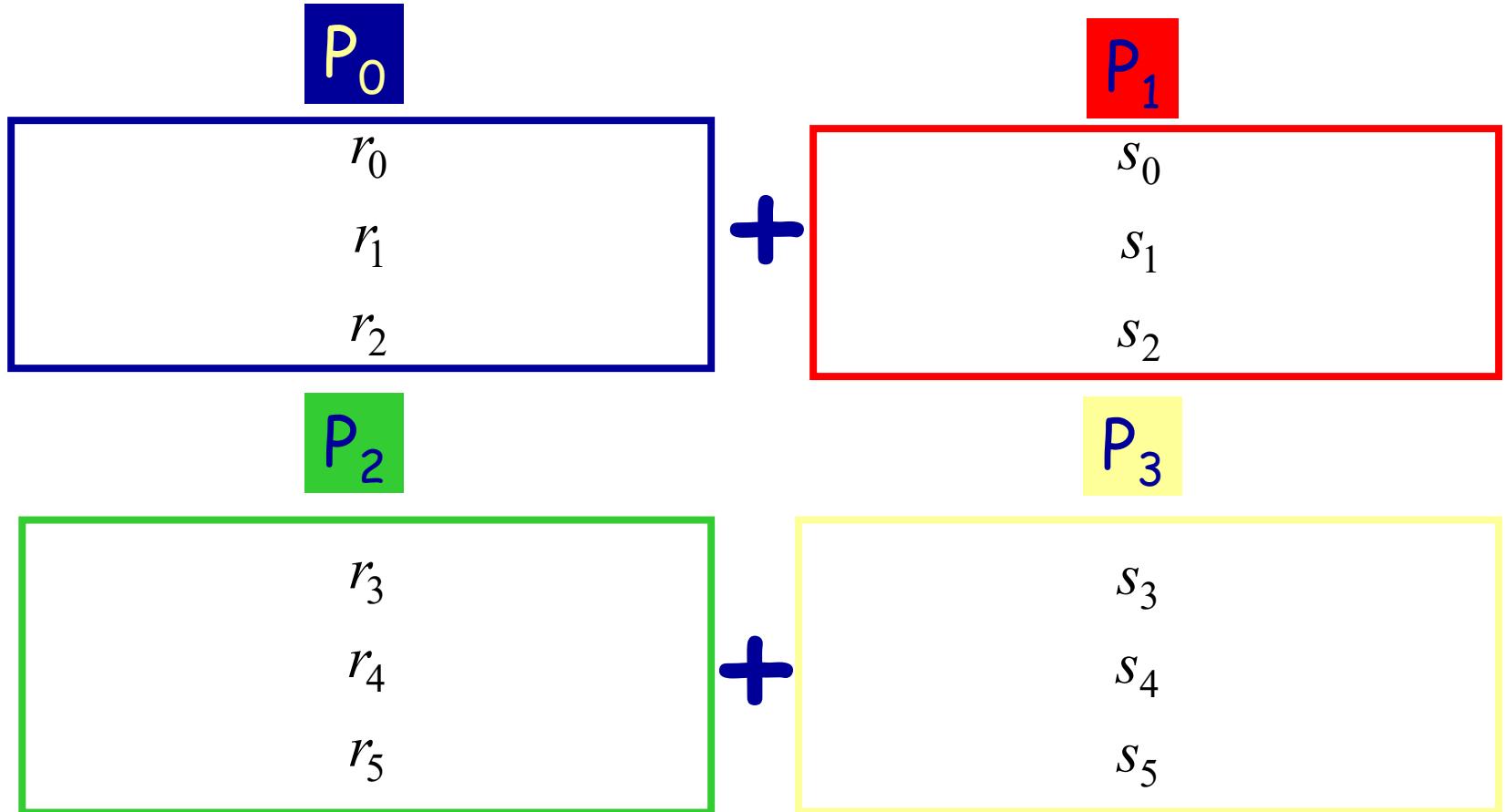
$$a_{33} \cdot x_3 + a_{34} \cdot x_4 + a_{35} \cdot x_5$$

$$a_{43} \cdot x_3 + a_{44} \cdot x_4 + a_{45} \cdot x_5$$

$$a_{53} \cdot x_3 + a_{54} \cdot x_4 + a_{55} \cdot x_5$$

Calcolo dei prodotti parziali

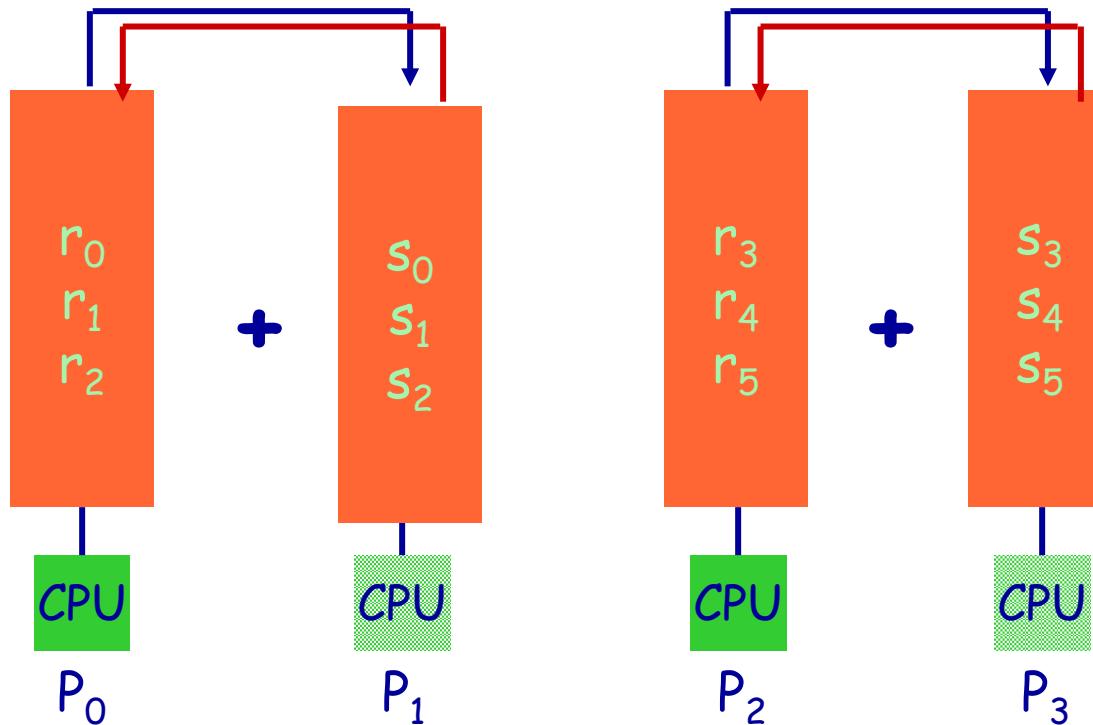
# Esempio $N = 6$ , Processori=4



Scambio: somma in parallelo

# Esempio III Strategia P=4

I passo

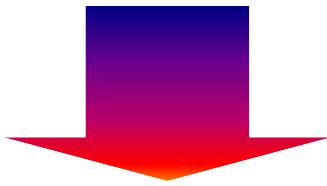


$P_0$  e  $P_1$  calcolano contemporaneamente entrambi la stessa somma di 2 vettori

$P_2$  e  $P_3$  calcolano contemporaneamente entrambi la stessa somma di 2 vettori

# III strategia: sintesi

Ciascun processore calcola  
somme parziali  
di alcune componenti del vettore  $y$

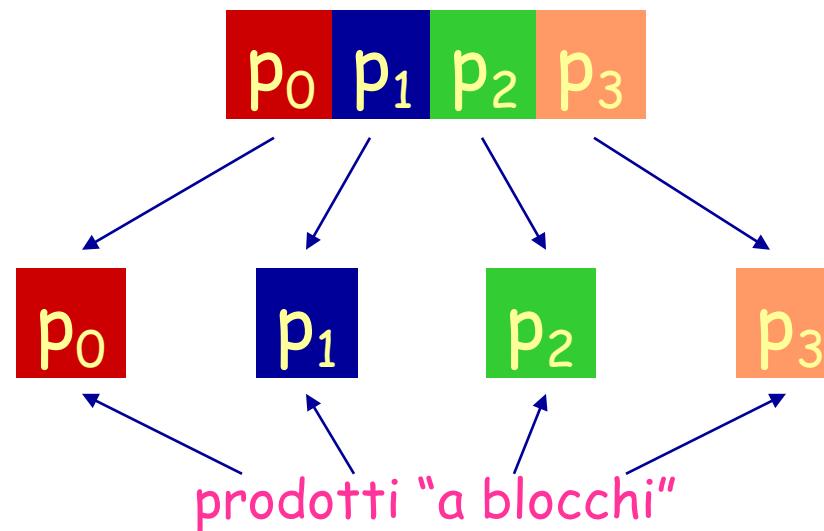


I processori devono sommare i risultati parziali  
e scambiarsi le componenti  
per avere il risultato finale,  $y$

# IDEA GENERALE: matrice vettore in parallelo

Riformulazione del prodotto matrice vettore in "prodotti a blocchi" ed assegnazione di ciascun di questi prodotti ad un processore

Prodotto matrice-vettore



# FINE LEZIONE

## Prodotto Matrice-Vettore

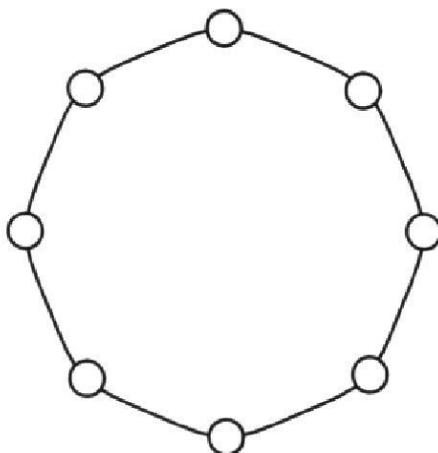
# **Message Passing Interface**

## **MPI**

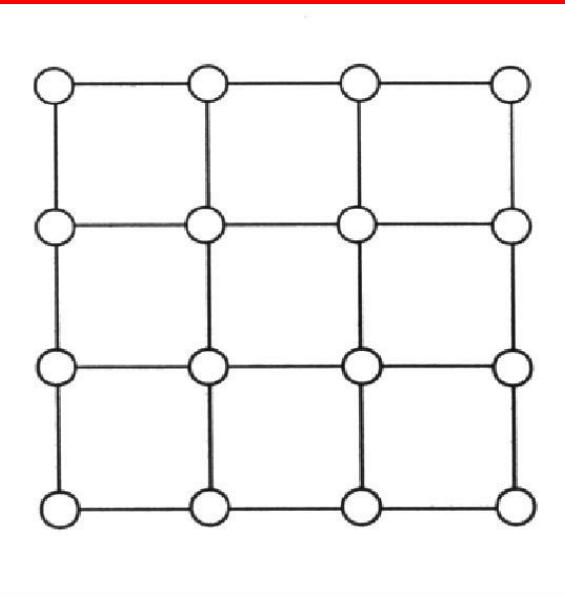
Dalle lezioni di Calcolo Parallelo  
del Prof. A. Murli

## Le topologie

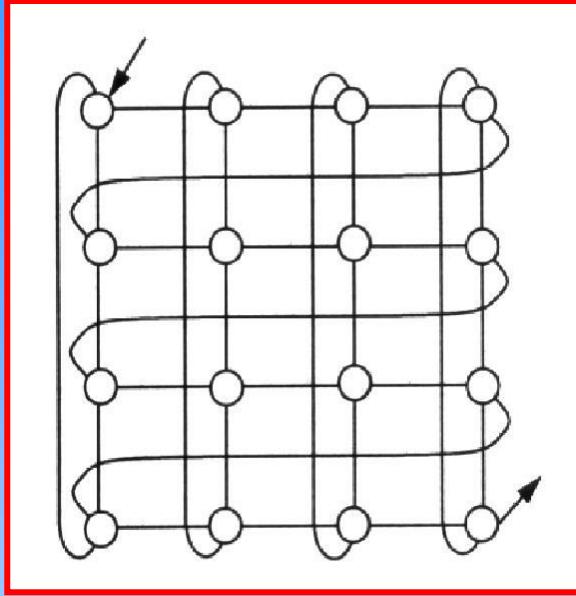
# Esempi di topologie



Anello



Griglia



Toro

L'utilizzo di una topologia per la progettazione di un algoritmo in ambiente MIMD è spesso legata alla geometria "intrinseca" del problema in esame.

# Definizione: *topologia*

Una topologia è  
**la geometria “virtuale”**  
in cui si immaginano disposti  
i processori.



La topologia “virtuale” in cui sono disposti  
i processori può non **avere alcun nesso**  
con la disposizione “reale” degli stessi!

**Griglia**

```
#include <stdio.h>
#include "mpi.h"
/* Scopo: definizione di una topologia
   a griglia bidimensionale nproc=row*col */
main(int argc, char **argv)
{ int menum,nproc,row,col;
  int dim,*ndim,reorder,*period,*coordinate;
  MPI_Comm comm_grid;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&menum);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
/* Numero di righe della griglia di processo */
  if (menum == 0)
  {   printf("Numero di righe della griglia");
      scanf("%d",&row); }
/* Spedizione di row da parte di 0 a tutti i processi */
  MPI_Bcast(&row,1,MPI_INT,0,MPI_COMM_WORLD);
/* Definizione del numero di colonne della griglia */
  col = nproc/row;
/* Numero di dimensioni della griglia */
  dim = 2;
  coordinate = (int*)calloc(dim,sizeof(int));
```

# Esempio: creazione di una griglia bidimensionale

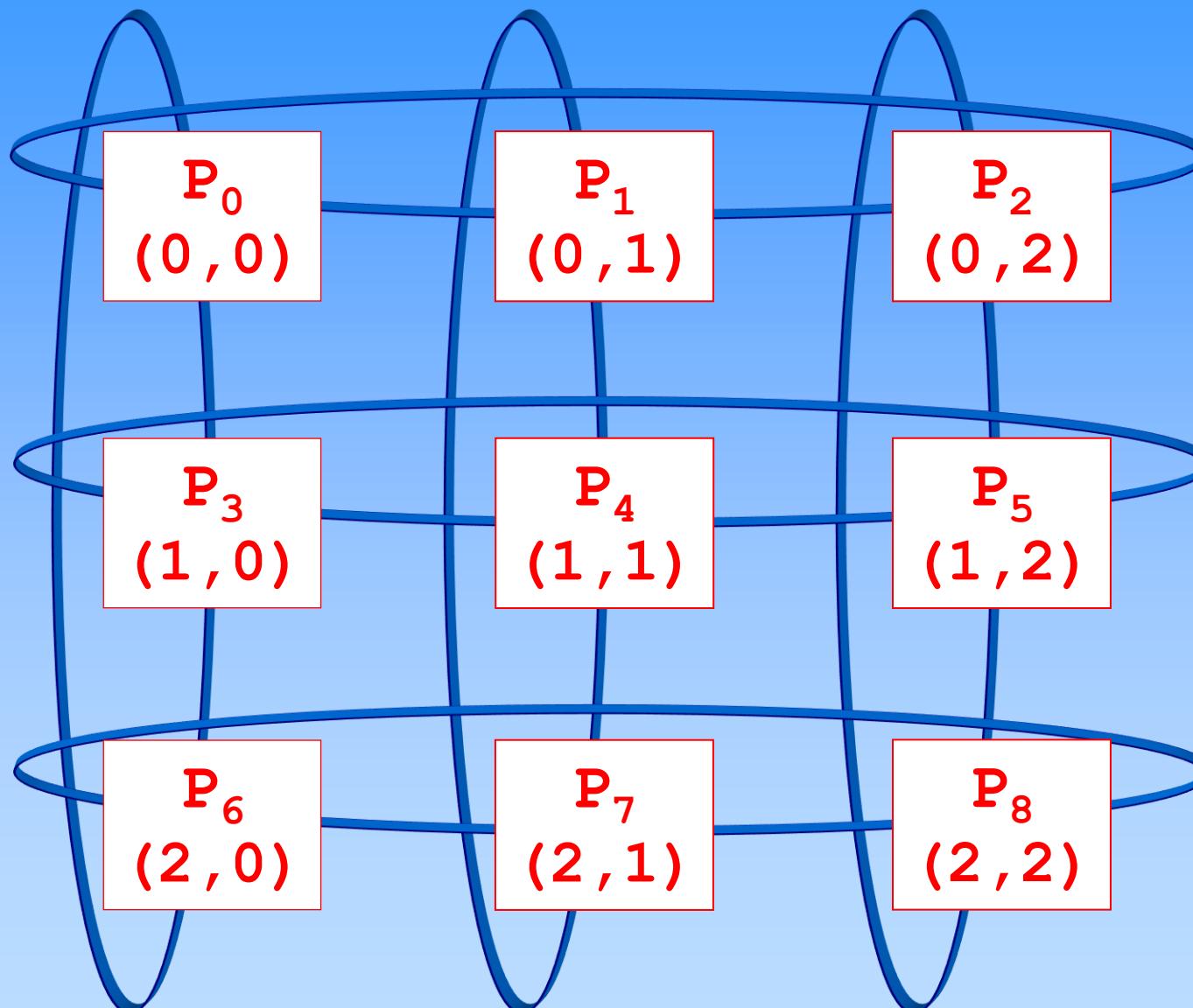
```
/* vettore contenente le lunghezze di ciascuna dimensione*/
ndim = (int*)calloc(dim,sizeof(int));
ndim[0] = row;
ndim[1] = col;
/* vettore contenente la periodicità delle dimensioni */
period = (int*)calloc(dim,sizeof(int));
period[0] = period[1] = 0;
reorder = 0;
/* Definizione della griglia bidimensionale */
MPI_Cart_create(MPI_COMM_WORLD,dim,ndim,period,reorder,
&comm_grid);
MPI_Comm_rank(comm_grid,&menu_num);
/* Definizione delle coordinate di ciascun processo
nella griglia bidimensionale */
MPI_Cart_coords(comm_grid,menu_num,dim,coordinate);
/* Stampa delle coordinate */
printf("Processore %d coordinate nella griglia
(%d,%d) \n",menu,*coordinate,* (coordinate+1));
MPI_Finalize();
return 0; }
```





```
MPI_Cart_create(MPI_COMM_WORLD, dim, ndim, period,  
reorder, &comm_grid);
```

- Ogni processo dell'ambiente `MPI_COMM_WORLD` definisce la griglia denominata `comm_grid`, di dimensione 2 (`dim`) e non periodica lungo le due componenti (`period[i]=0, i=0,1`). Il numero di righe e di colonne della griglia sono memorizzati rispettivamente nella prima e nella seconda componente del vettore `ndim`. I processi non sono riordinati secondo un particolare schema (`reorder=0`).



```
MPI_Cart_create(MPI_Comm comm_old,int dim,
                  int *ndim, int *period,
                  int reorder,
                  MPI_Comm *new_comm);
```

- Operazione collettiva che restituisce un nuovo communicator **new\_comm** in cui i processi sono organizzati in una griglia di dimensioni **dim**.
- L'i-esima dimensione ha lunghezza **ndim[i]**.
- Se **period[i]=1**, la i-esima dimensione della griglia è periodica; non lo è se **period[i]=0**.

```
MPI_Cart_create(MPI_Comm comm_old,int dim,
                  int *ndim, int *period,
                  int reorder,
                  MPI_Comm *new_comm);
```

**comm\_old** communicator di input

**dim** numero di dimensioni della griglia

**\*ndim** vettore di dimensione **dim** contenente le lunghezze di ciascuna dimensione

**\*period** vettore di dimensione **dim** contenente la periodicità di ciascuna dimensione

**reorder** permesso di riordinare i **menum** (1=si; 0=no)

**\*new\_comm** communicator di output associato alla griglia

# Esempio: creazione di una griglia bidimensionale

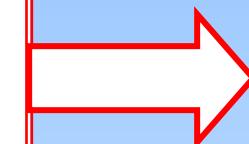
```
/* vettore contenente le lunghezze di ciascuna dimensione*/
ndim = (int*)calloc(dim,sizeof(int));
ndim[0] = row;
ndim[1] = col;
/* vettore contenente la periodicità delle dimensioni */
period = (int*)calloc(dim,sizeof(int));
period[0] = period[1] = 0;
reorder = 0;
/* Definizione della griglia bidimensionale */

MPI_Cart_Create(MPI_COMM_WORLD,dim,ndim,period,reorder,
                 &comm_grid);

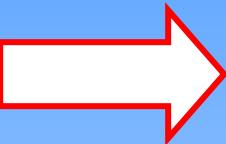
MPI_Comm_rank(comm_grid,&menu_num_grid);
/* Definizione delle coordinate di ciascun processo
nella griglia bidimensionale */

MPI_Cart_coords(comm_grid,menu_num,dim,coordinate);

/* Stampa delle coordinate */
printf("Processore %d coordinate nella griglia
      (%d,%d) \n",menu,*coordinate,* (coordinate+1));
MPI_Finalize();
return 0; }
```



Nel programma...:



```
MPI_Cart_coords(comm_grid, menum, dim, coordinate);
```

- Ogni processo **menum** calcola le proprie 2 (**dim**) coordinate (**coordinate<sub>i</sub>**, i=0,1) nell'ambiente **comm\_grid**.

# Funzione per definire le coordinate

```
MPI_Cart_coords (MPI_Comm comm_grid,
int menum_grid, int dim,
int *coordinate);
```

- Operazione collettiva che restituisce a ciascun processo di **comm\_grid** con identificativo **menum\_grid**, le sue coordinate all'interno della griglia predefinita.
- **coordinate** è un vettore di dimensione **dim**, i cui elementi rappresentano le coordinate del processo all'interno della griglia.

**Fine Esercitazione**

# Architettura delle GPU

## cenni alla programmazione CUDA

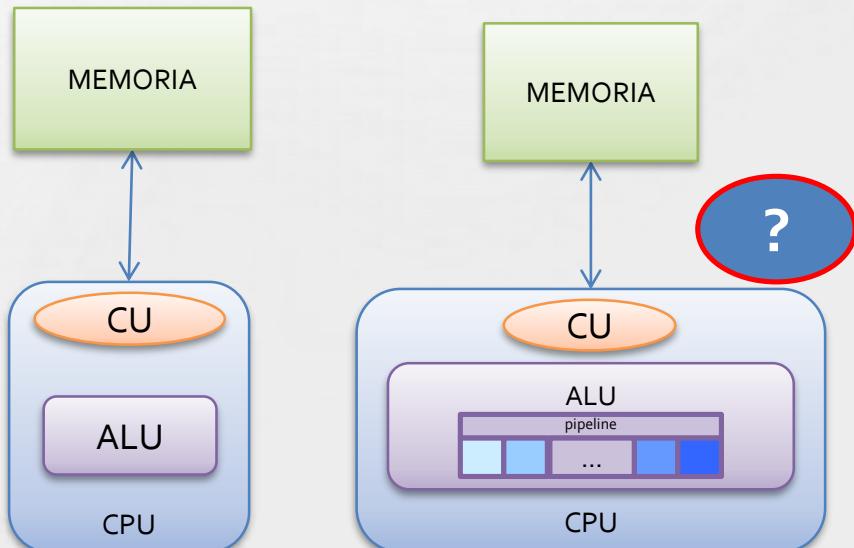
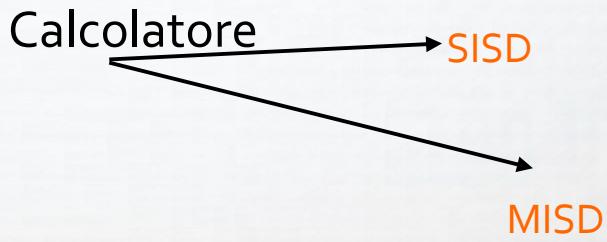
*Prof. G. Laccetti*

Corso di Parallel and Distributed Computing  
Università degli Studi di Napoli Federico II

a.a. 2021/22

# Premesse

## Tassonomia di Flynn



**Instruction Stream:** sequenza di istruzioni eseguite dalla macchina.

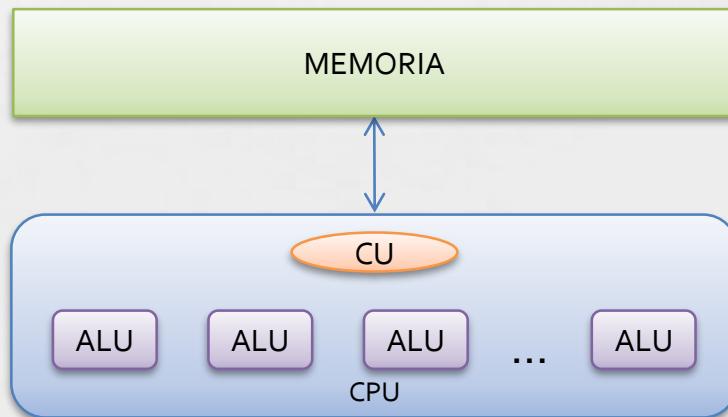
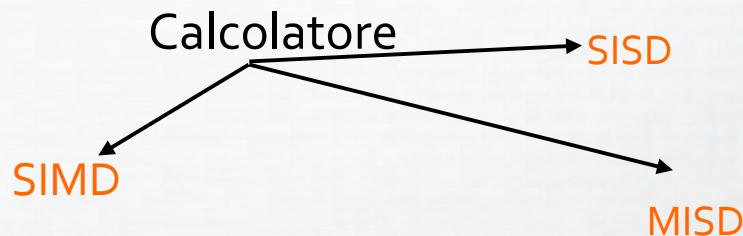
**Data Stream:** sequenza di dati richiesta dal flusso di istruzioni, compresi input e variabili d'appoggio.

[1]

- Single Instruction Stream / Single Data Stream (SISD)
- Single Instruction Stream / Multiple Data Stream (SIMD)
- Multiple Instruction Stream / Single Data Stream (MISD)
- Multiple Instruction Stream / Multiple Data Stream (MIMD)

# Premesse

## Tassonomia di Flynn



**Instruction Stream**: sequenza di istruzioni eseguite dalla macchina.

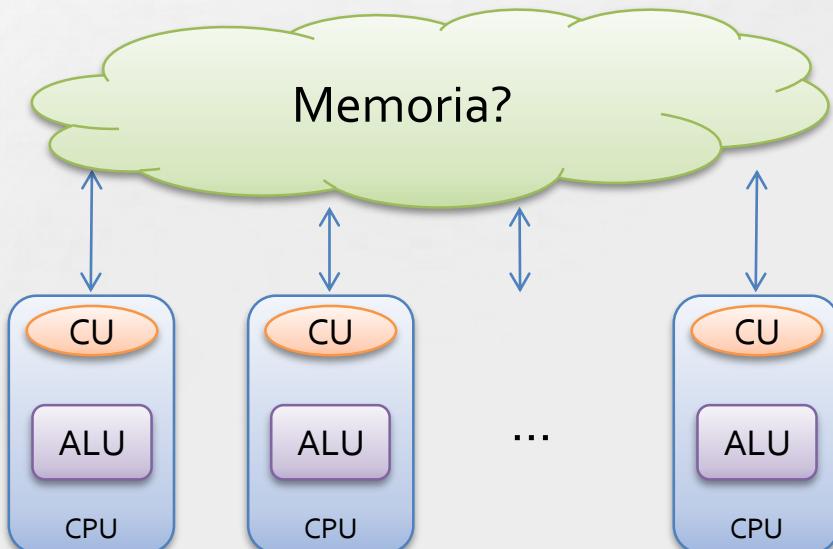
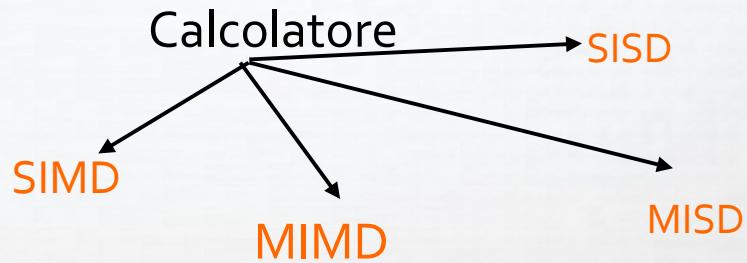
**Data Stream**: sequenza di dati richiesta dal flusso di istruzioni, compresi input e variabili d'appoggio.

[1]

- Single Instruction Stream / Single Data Stream (SISD)
- Single Instruction Stream / Multiple Data Stream (SIMD)
- Multiple Instruction Stream / Single Data Stream (MISD)
- Multiple Instruction Stream / Multiple Data Stream (MIMD)

# Premesse

## Tassonomia di Flynn



**Instruction Stream:** sequenza di istruzioni eseguite dalla macchina.

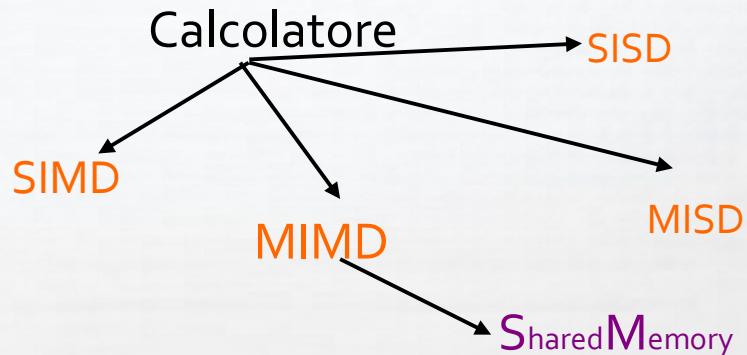
**Data Stream:** sequenza di dati richiesta dal flusso di istruzioni, compresi input e variabili d'appoggio.

[1]

- Single Instruction Stream / Single Data Stream (SISD)
- Single Instruction Stream / Multiple Data Stream (SIMD)
- Multiple Instruction Stream / Single Data Stream (MISD)
- Multiple Instruction Stream / Multiple Data Stream (MIMD)

# Premesse

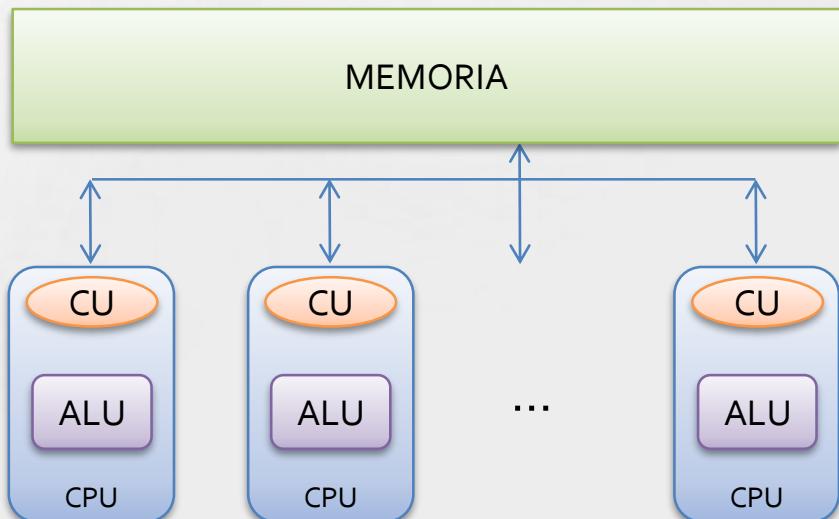
## Tassonomia di Flynn



**Instruction Stream:** sequenza di istruzioni eseguite dalla macchina.

**Data Stream:** sequenza di dati richiesta dal flusso di istruzioni, compresi input e variabili d'appoggio.

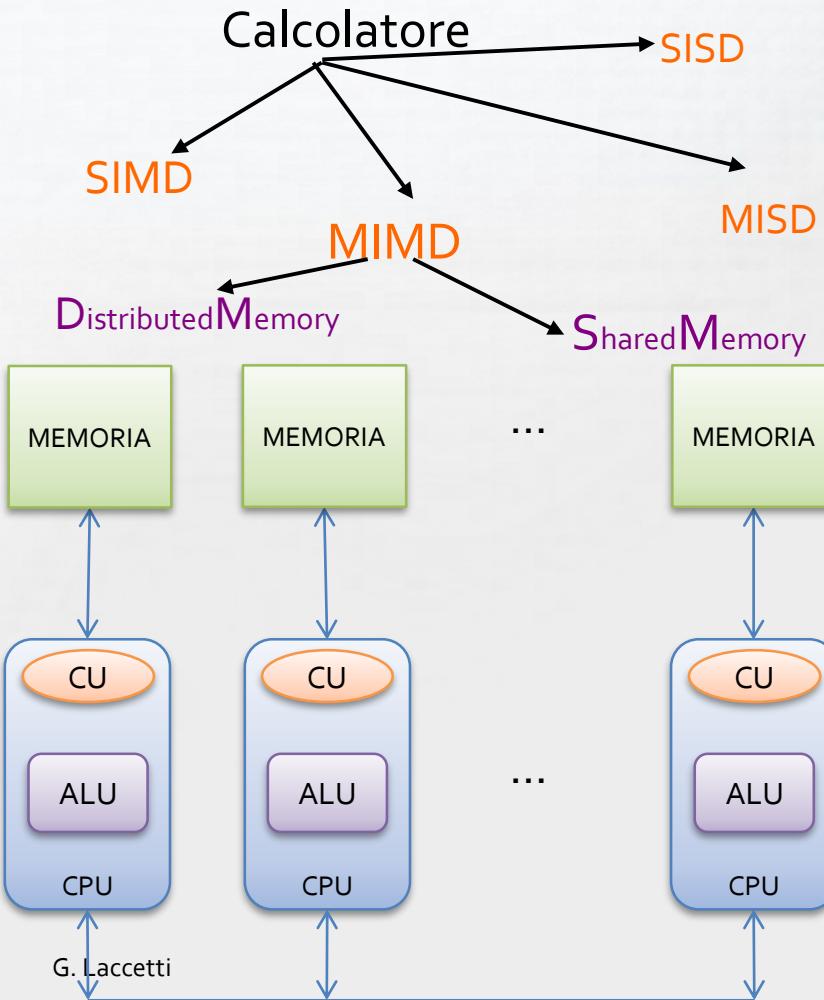
[1]



- Single Instruction Stream / Single Data Stream (SISD)
- Single Instruction Stream / Multiple Data Stream (SIMD)
- Multiple Instruction Stream / Single Data Stream (MISD)
- Multiple Instruction Stream / Multiple Data Stream (MIMD)

# Premesse

## Tassonomia di Flynn



**Instruction Stream:** sequenza di istruzioni eseguite dalla macchina.

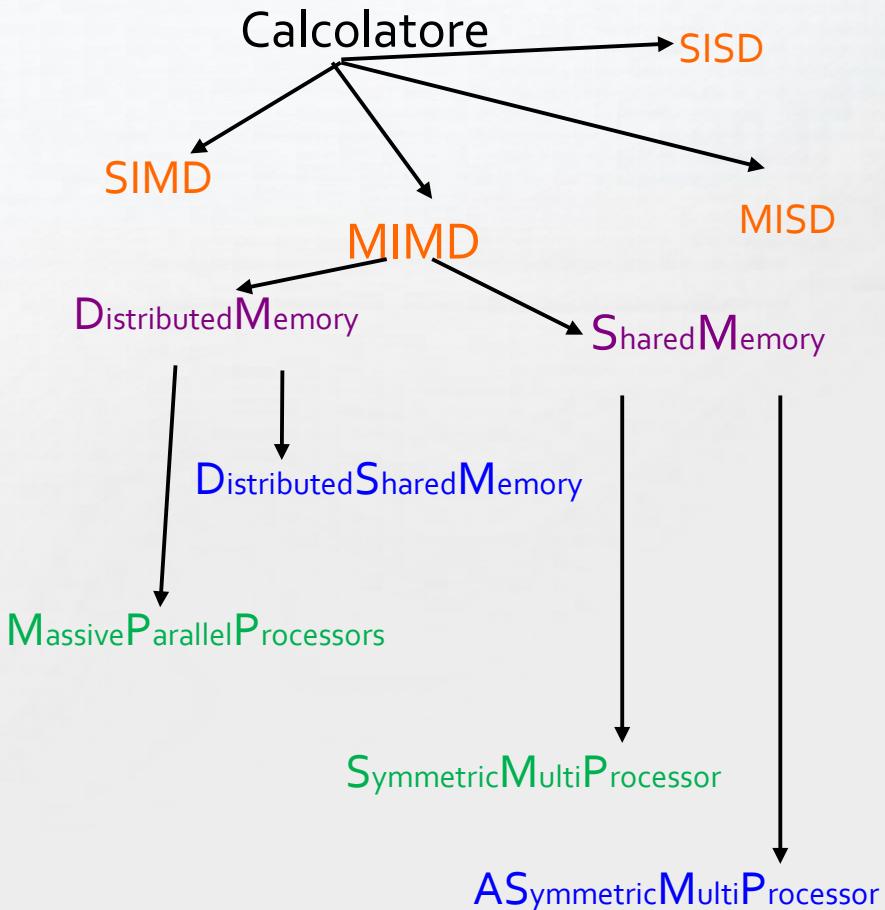
**Data Stream:** sequenza di dati richiesta dal flusso di istruzioni, compresi input e variabili d'appoggio.

[1]

- Single Instruction Stream / Single Data Stream (SISD)
- Single Instruction Stream / Multiple Data Stream (SIMD)
- Multiple Instruction Stream / Single Data Stream (MISD)
- Multiple Instruction Stream / Multiple Data Stream (MIMD)

# Premesse

## Tassonomia di Flynn



**Instruction Stream:** sequenza di istruzioni eseguite dalla macchina.

**Data Stream:** sequenza di dati richiesta dal flusso di istruzioni, compresi input e variabili d'appoggio.

[1]

- Single Instruction Stream / Single Data Stream (SISD)
- Single Instruction Stream / Multiple Data Stream (SIMD)
- Multiple Instruction Stream / Single Data Stream (MISD)
- Multiple Instruction Stream / Multiple Data Stream (MIMD)

# Premesse

## Sistemi di calcolo moderni

- I sistemi attuali sono generalmente **ibridi** rispetto alla tassonomia di Flynn e molto spesso **eterogenei**

### IBRIDISMO

Nei progetti di architetture moderne vengono stratificati stili di parallelismo diversi

### ETEROGENEITA'

Sottosistemi con strutture completamente diverse vengono accostati per cooperare

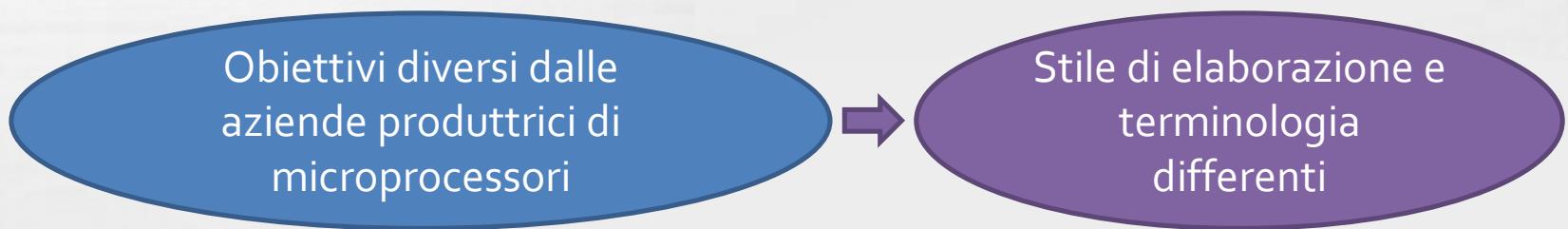
SVARIATE combinazioni delle classi di Flynn

Esigenza di estendere e dettagliare i modelli esistenti a supporto della valutazione e della progettazione

# GPU

## Evoluzione

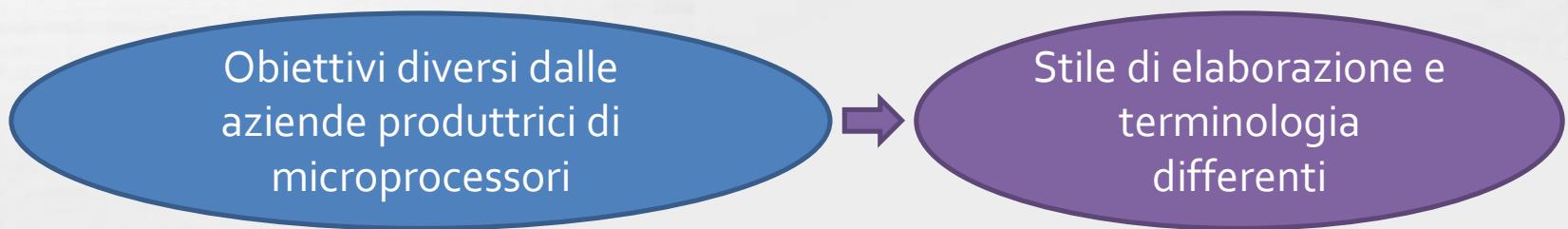
- L'industria dei videogiochi ha investito moltissimo nel miglioramento dell'elaborazione grafica.
  - Nei primi anni '80 furono introdotte componenti specializzate nel rendering grafico per alleggerire il compito computazionale delle CPU



# GPU

## Evoluzione

- L'industria dei videogiochi ha investito moltissimo nel miglioramento dell'elaborazione grafica.
  - Nei primi anni '80 furono introdotte componenti specializzate nel rendering grafico per alleggerire il compito computazionale delle CPU

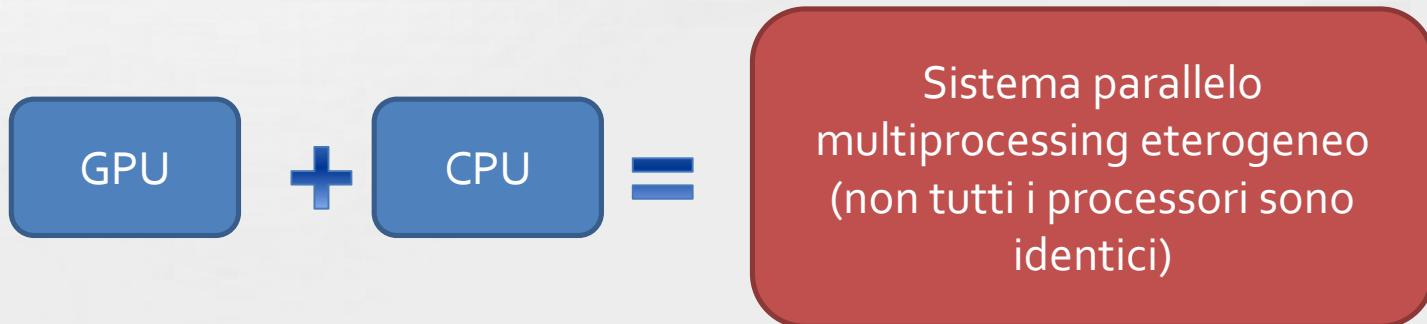


- PIPELINE GRAFICA: processo di **rendering** (generazione di un'immagine a partire da una descrizione di oggetti 3D) realizzato attraverso un hardware più o meno specializzato.

# GPU

## Evoluzione

- Le GPU, in quanto coprocessori ad integrazione della CPU, non hanno bisogno di eseguire tutti i compiti che sono eseguiti normalmente da una CPU.
  - Possono dedicare tutte le risorse alla grafica.
  - in un sistema che contiene sia una GPU che una CPU sarà la CPU ad eseguire i compiti per cui la GPU non è efficiente.



- Col tempo sono diventate sempre più programmabili e precise nei calcoli.

# GPU

## Evoluzione

- I dispositivi più moderni sono multiprocessori altamente paralleli e multithread, che nascondono l'elevata latenza di memoria sfruttando il parallelismo tra molti thread, per cui la memoria principale viene ottimizzata per massimizzare la larghezza di banda del trasferimento invece che per minimizzare la latenza.
- Vengono ormai progettate utilizzando processori di uso generico tra loro identici, diventando così più simili alle architetture multicore dei microprocessori convenzionali.



Stesso tipo di core:

- Maggiore scalabilità
- Migliore bilanciamento del carico

**MIGLIORI PRESTAZIONI**

# GPU

## Evoluzione

- I dispositivi più moderni sono multiprocessori altamente paralleli e multithread, che nascondono l'elevata latenza di memoria sfruttando il parallelismo tra molti thread, per cui la memoria principale viene ottimizzata per massimizzare la larghezza di banda del trasferimento invece che per minimizzare la latenza.
- Vengono ormai progettate utilizzando processori di uso generico tra loro identici, diventando così più simili alle architetture multicore dei microprocessori convenzionali.



**GPGPU**

**General Purpose GPUs**  
utilizzate per applicazioni  
generiche

# GPU

## Evoluzione

- I dispositivi più moderni sono multiprocessori altamente paralleli e multithread, che nascondono l'elevata latenza di memoria sfruttando il parallelismo tra molti thread, per cui la memoria principale viene ottimizzata per massimizzare la larghezza di banda del trasferimento invece che per minimizzare la latenza.
- Vengono ormai progettate utilizzando processori di uso generico tra loro identici, diventando così più simili alle architetture multicore dei microprocessori convenzionali.



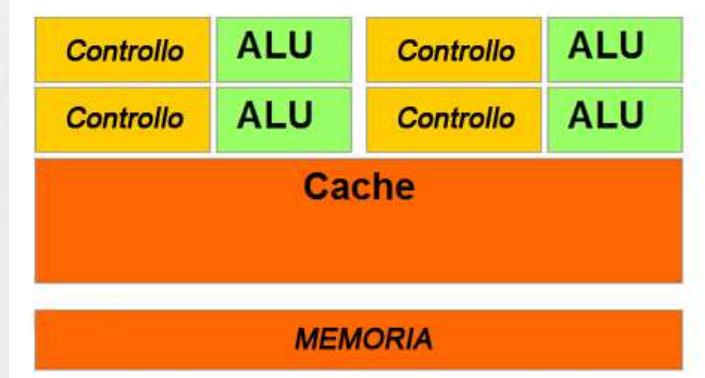
**GPGPU**

Affiancate alla CPU  
che mantiene il controllo  
(da cui il termine  
“acceleratore”)

# GPU vs CPU

## CPU

- Molto spazio on-chip dedicato al controllo e alla cache
- Poche unità grafiche processanti, potenti e sofisticate
- Adatta a database, algoritmi ricorsivi, flussi di controllo non regolari



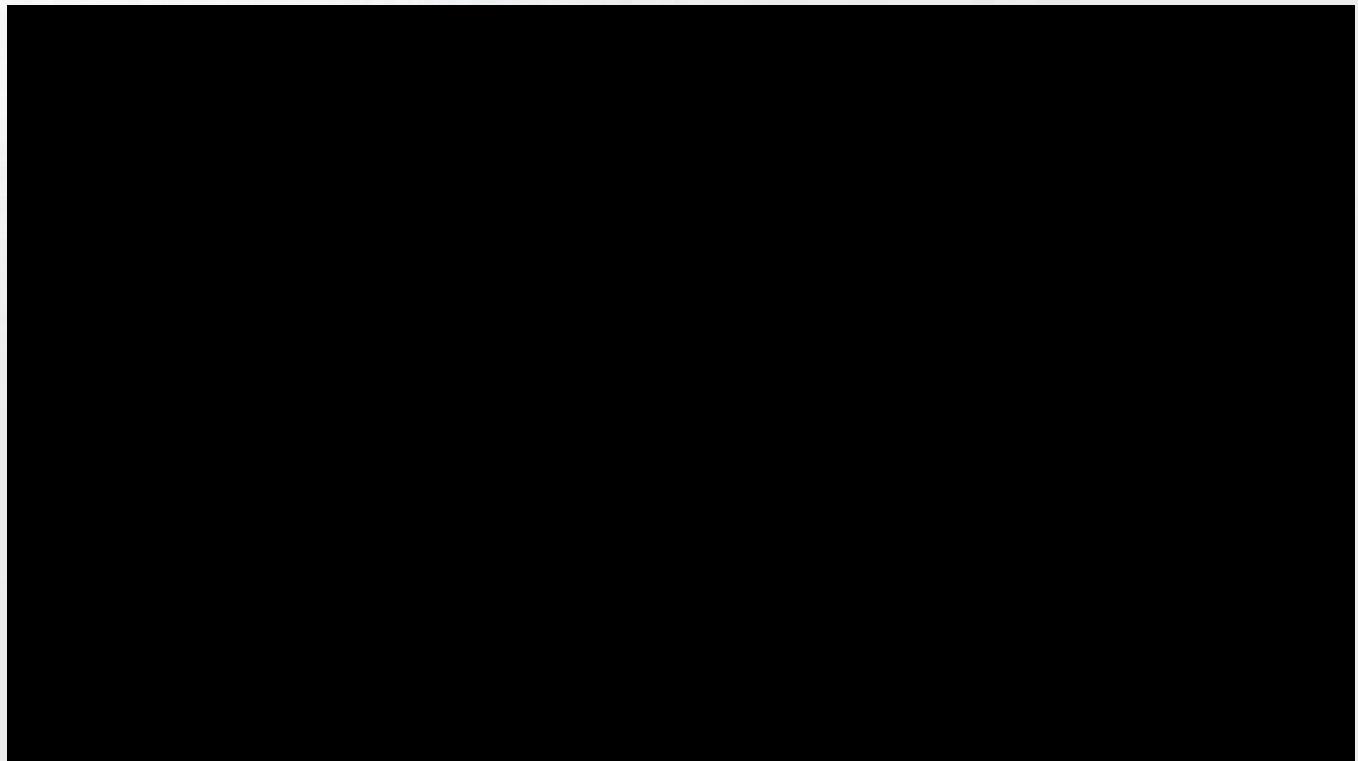
## GPU

- Molte unità dedicate al calcolo
- Adatto a calcoli ripetitivi e su grandi quantità di dati
- La memoria on-chip evita il collo di bottiglia del bus di sistema



# GPU vs CPU

Differenza secondo NVIDIA...



# GPU vs CPU

## Nella realtà...

- Non per tutti i problemi vale la superiorità della GPU
- Non per tutti i MODI di affrontare i problemi vale la superiorità della GPU
  - Anche nelle architetture più moderne la flessibilità non raggiunge quella delle CPU
- La cosa veramente importante è saper riconoscere il modo migliore di far collaborare le GPU e le CPU del sistema che abbiamo a disposizione.
  - Le GPU si utilizzano per ACCELERARE porzioni di codice con le caratteristiche giuste
  - Sono ormai disponibili in quasi tutti i sistemi, dai PC ai supercomputer, a costi relativamente contenuti
  - Il numero di core che forniscono continua a crescere

# GPU

## CUDA

- Con il GPGPU nascono anche linguaggi di alto livello adatti a programmare questi dispositivi.
- Alla fine degli anni 2000 NVIDIA introduce una particolare architettura di elaborazione in parallelo: **CUDA** (Compute Unified Device Architecture),
  - Con questa architettura nascono anche estensioni per i principali linguaggi di programmazione, soprattutto per C, che individuano un modello di programmazione parallela scalabile a memoria condivisa.
  - Attraverso CUDA/C lo sviluppatore può servirsi di alcune virtualizzazioni dell'architettura che semplificano molto la programmabilità della GPU.
- Nel 2008, AMD ed NVIDIA, insieme ad altre industrie del settore, hanno investito sulla realizzazione di OpenCL, linguaggio che segue la filosofia e i principi di CUDA/C ma concepito come *open-source*
  - dal 2009 OpenCL è supportato dai prodotti di entrambe le case

# GPU

## CUDA

- Approfondire CUDA e CUDA/C è un buon modo per approcciare lo studio delle GPU.
  - L'architettura NVIDIA G80 ha segnato una tendenza nelle scelte progettuali, non radicalmente diverse da quelle fatte da AMD (ATI), che caratterizzano ormai questo tipo di dispositivi.
  - Il modello CUDA è applicabile anche ad altre architetture di elaborazione parallela a memoria condivisa, come le CPU multicore.
  - Le astrazioni introdotte da CUDA rispecchiano esattamente le caratteristiche dell'hardware delle GPU NVIDIA, che riuniscono più livelli di parallelismo: tali astrazioni guidano il programmatore nella suddivisione del problema e nella sincronizzazione del lavoro.

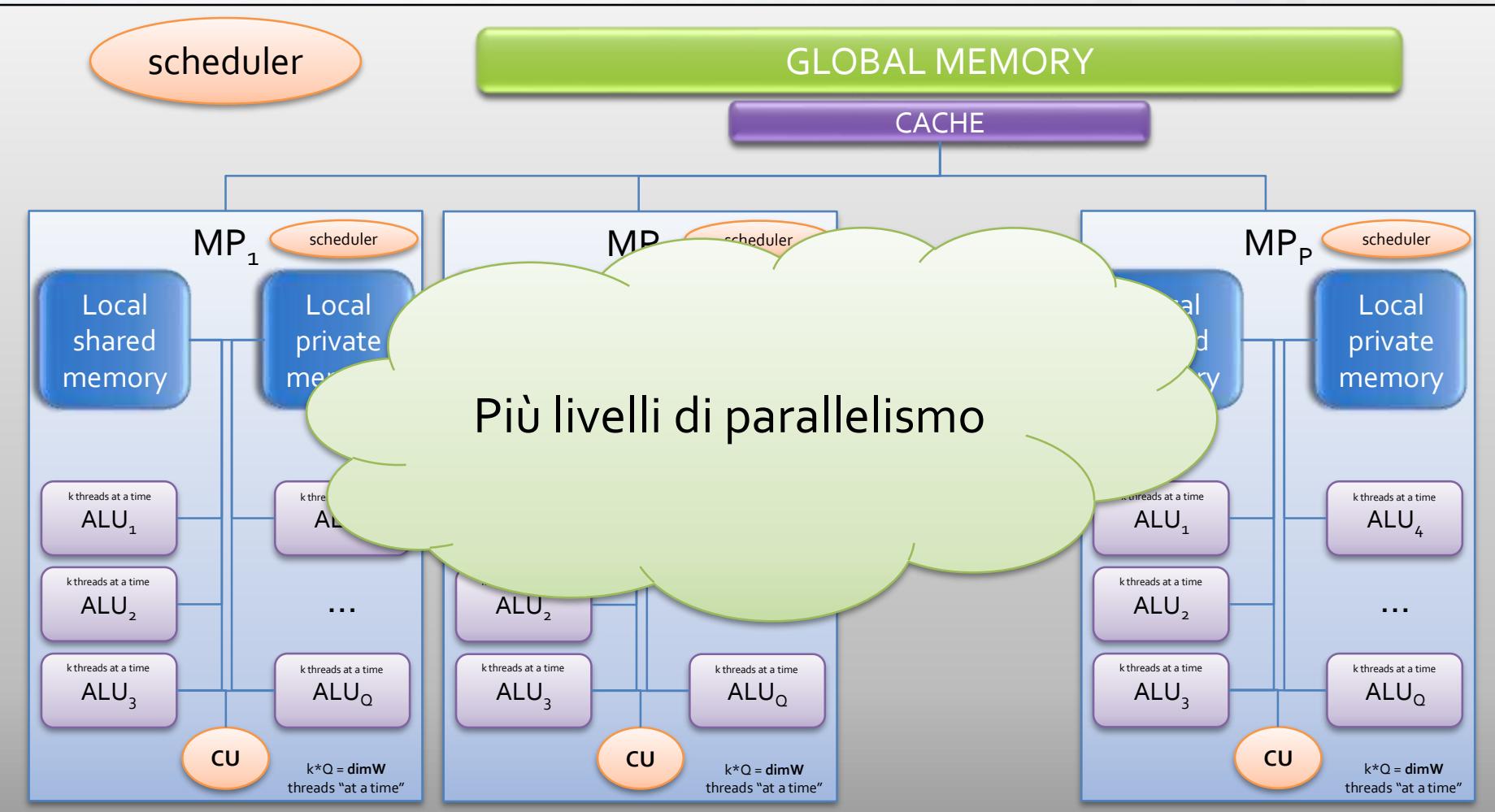
# GPU

## Architettura di una GPU

Cominciamo ora a modellare  
un'architettura che rispecchi le caratteristiche delle più  
comuni GPU

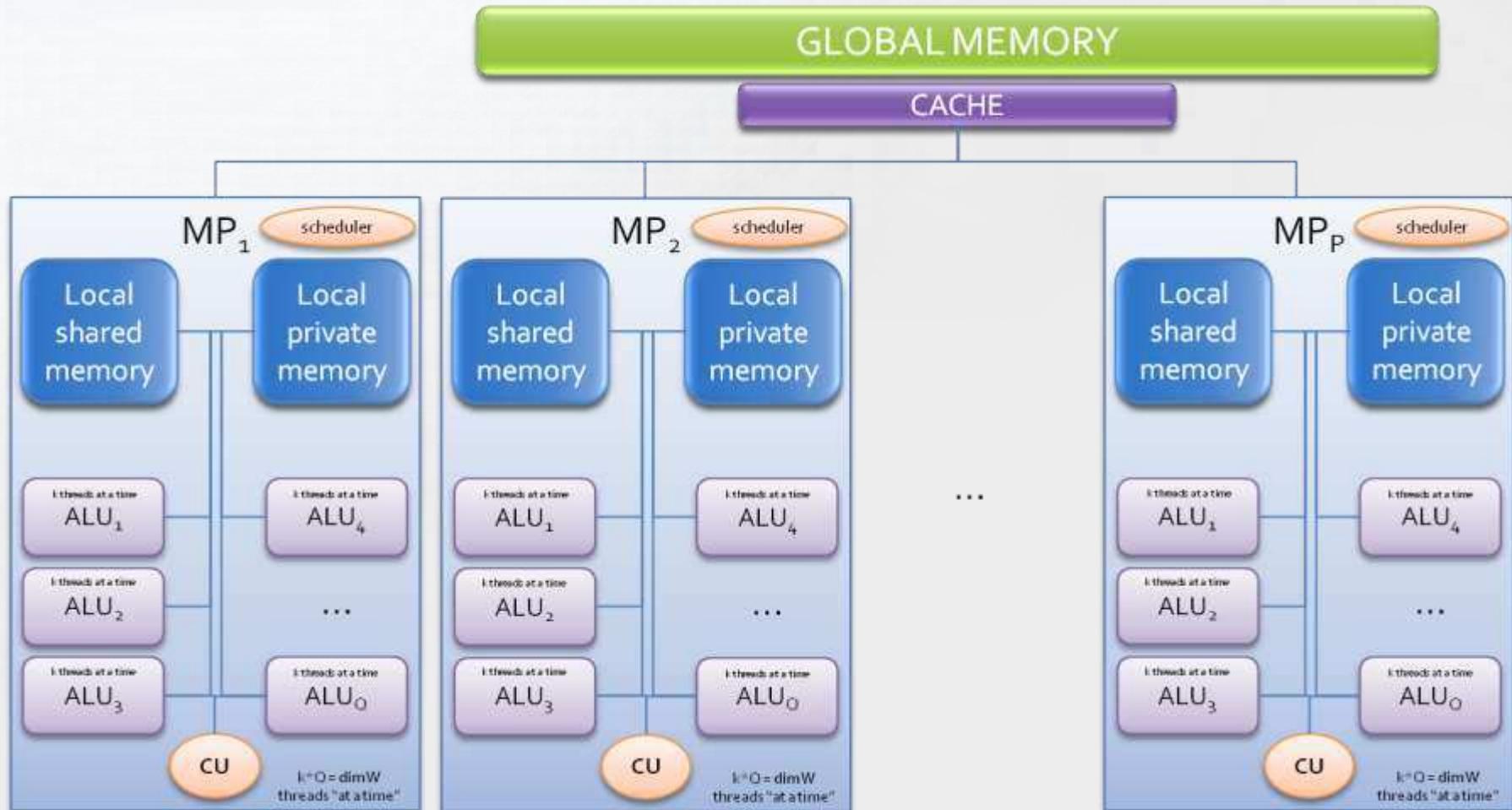
# GPU

## Architettura di una GPU: Più Livelli di Parallelismo



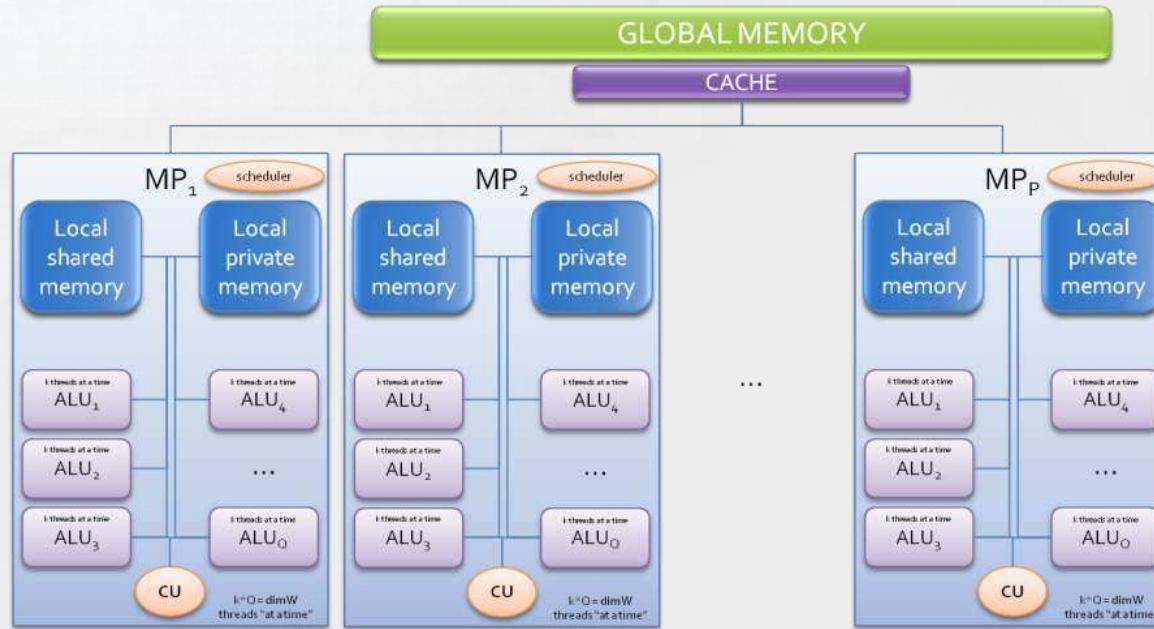
# GPU

## Architettura di una GPU: Più Livelli di Parallelismo



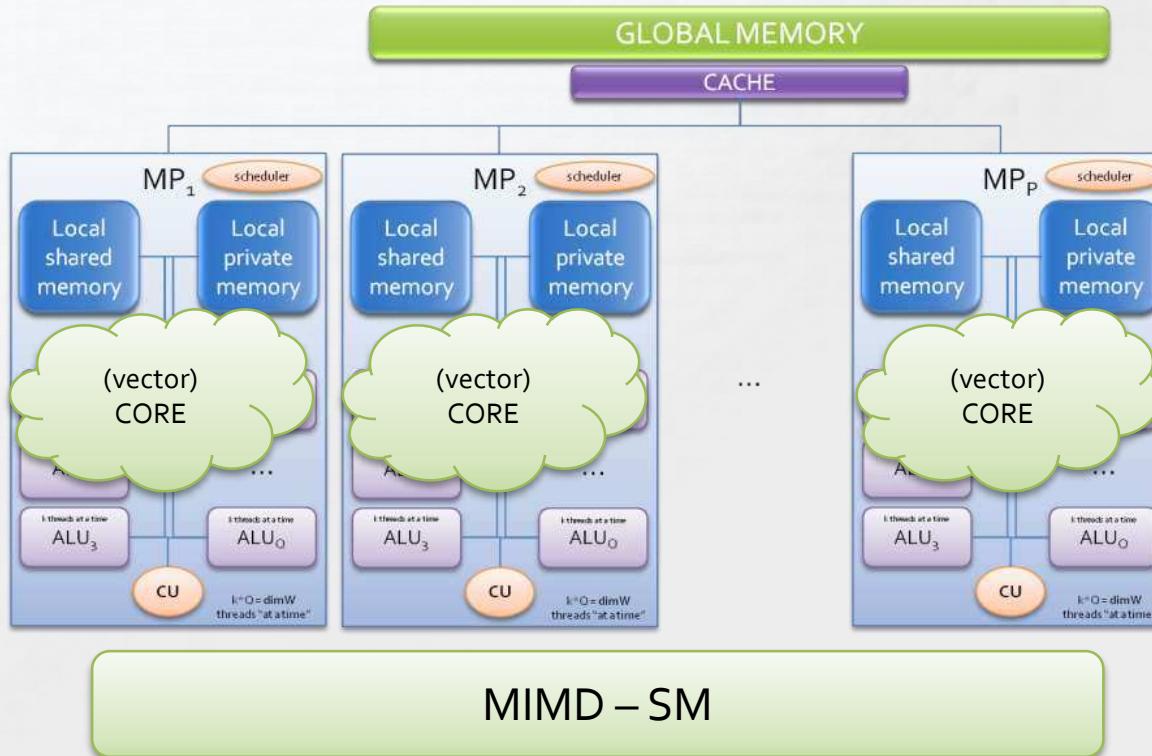
# GPU

## Architettura di una GPU: Più Livelli di Parallelismo



# GPU

## Architettura di una GPU: Più Livelli di Parallelismo



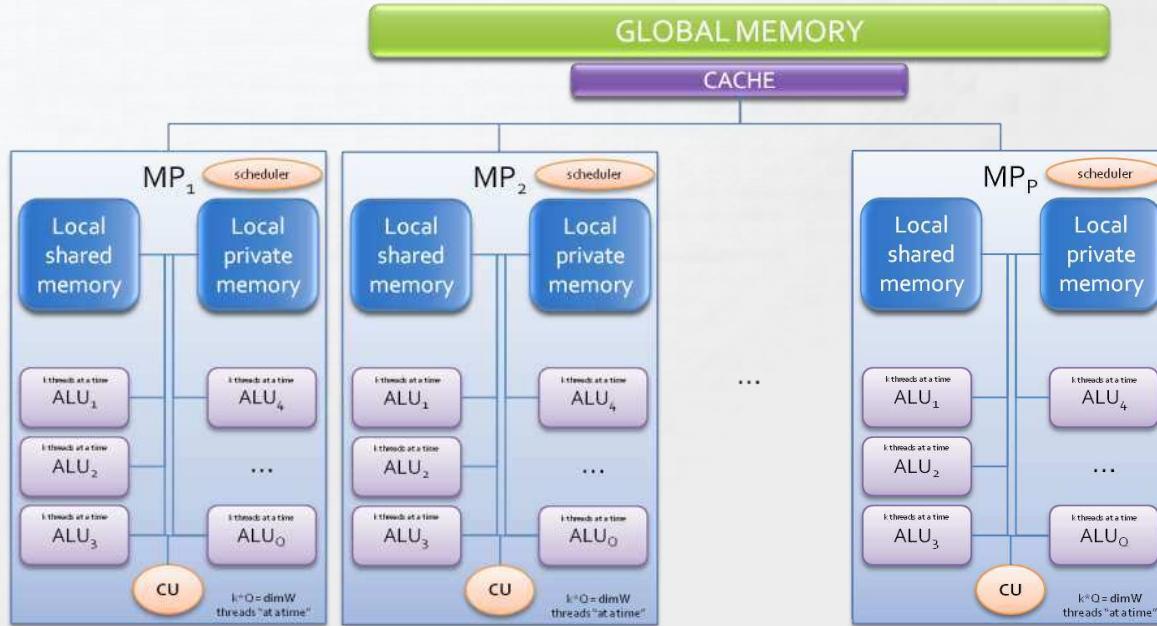
### Livello più alto

P core che condividono una memoria (con eventuale cache):

macchina MIMD  
shared memory.

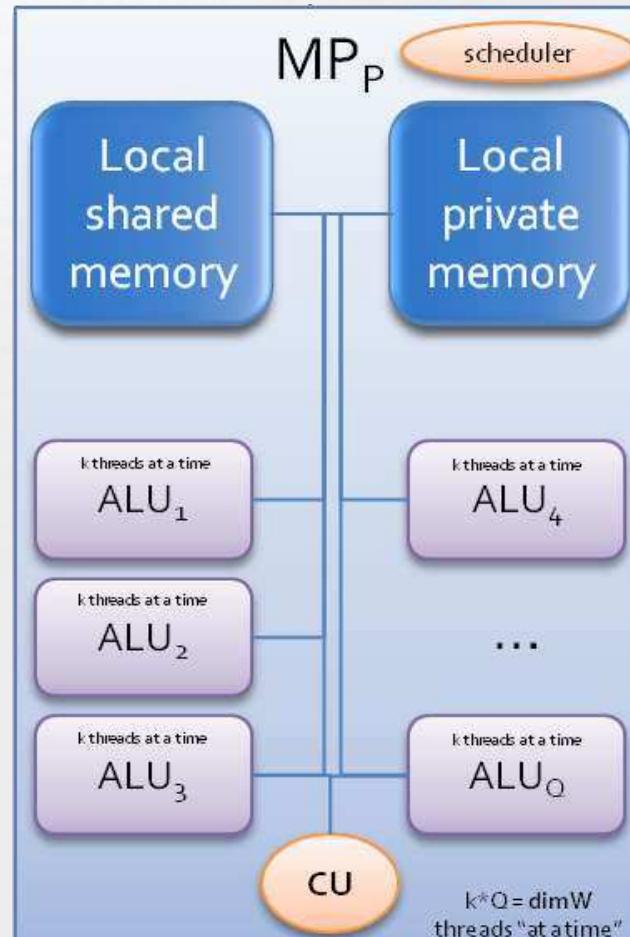
# GPU

## Architettura di una GPU: Più Livelli di Parallelismo



# GPU

## Architettura di una GPU: Più Livelli di Parallelismo

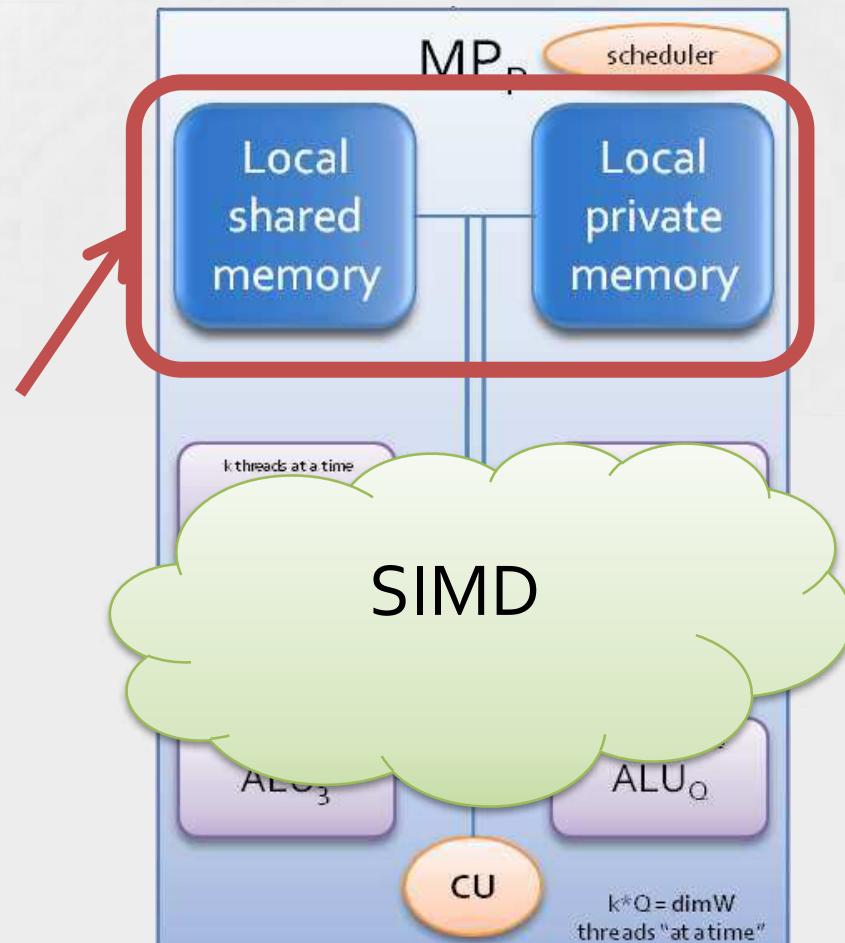


# GPU

## Architettura di una GPU: Più Livelli di Parallelismo

### Livello intermedio

Ogni MP è un calcolatore SIMD con  $Q$  ALU,



Ogni MP ha una sua area di memoria locale

# GPU

## Architettura di una GPU: Più Livelli di Parallelismo

### Livello intermedio

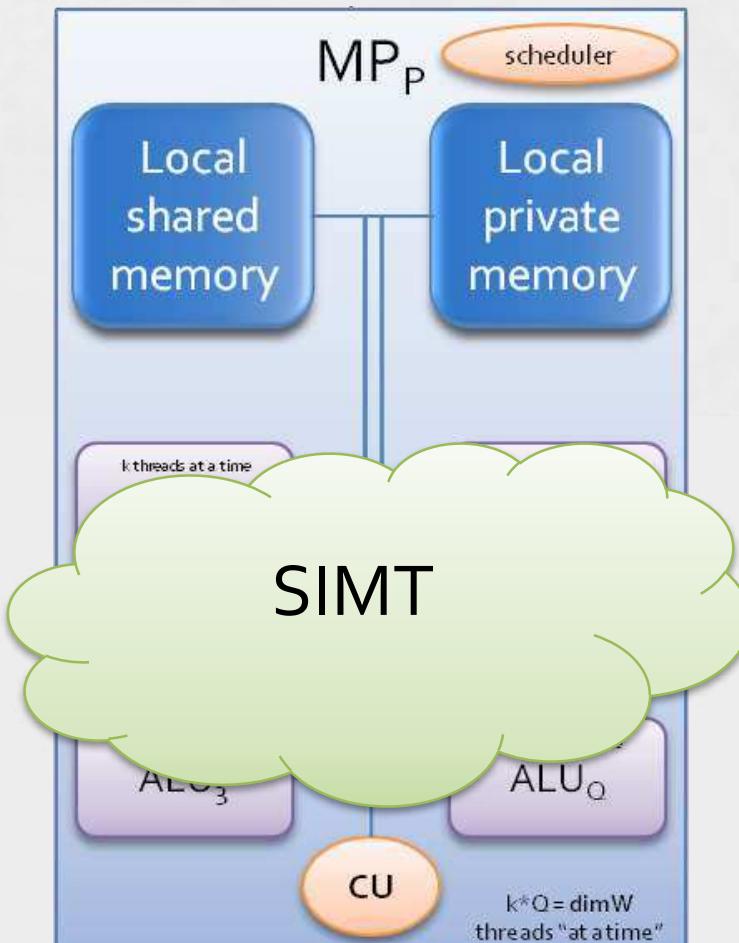
Ogni MP è un calcolatore SIMD con **Q ALU**,

Commercialmente viene chiamata **SIMT**

Ovvero

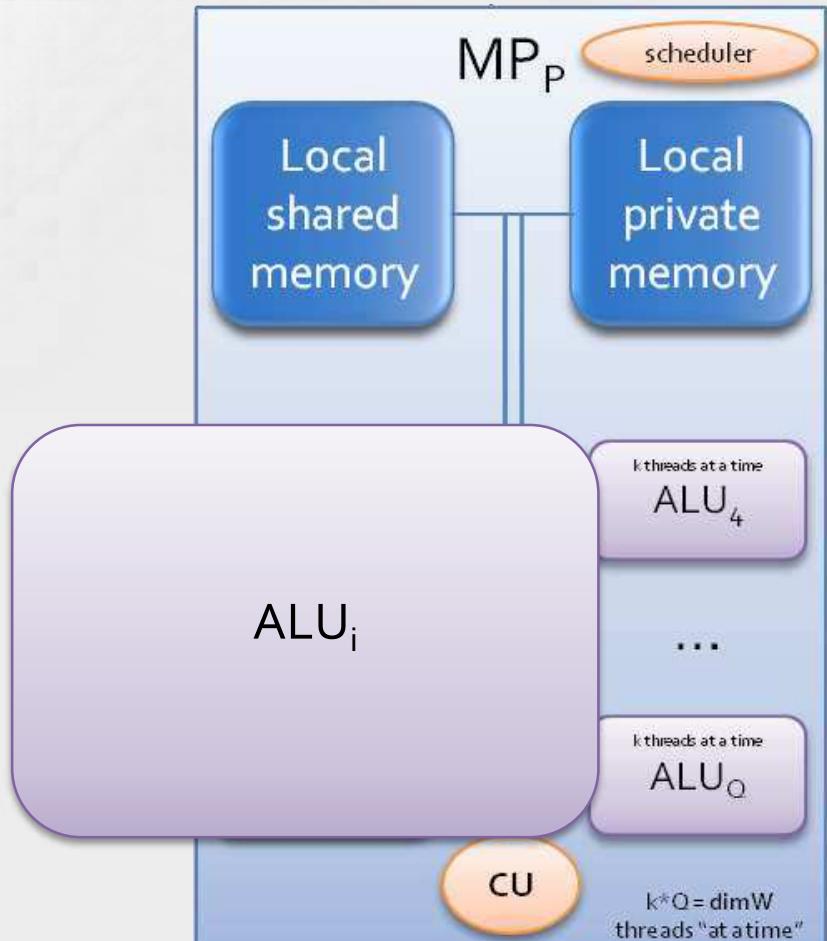
**Single Instruction Multiple Threads**

Questo perché nella realizzazione fisica non si tratta di ALU ma di piccole e semplici CPU complete, e i thread possono avere comportamenti leggermente diversi tra loro (sono ammessi i branch) : eseguono la stessa porzione di codice, ma non sempre ESATTAMENTE le stesse istruzioni.



# GPU

## Architettura di una GPU: Più Livelli di Parallelismo



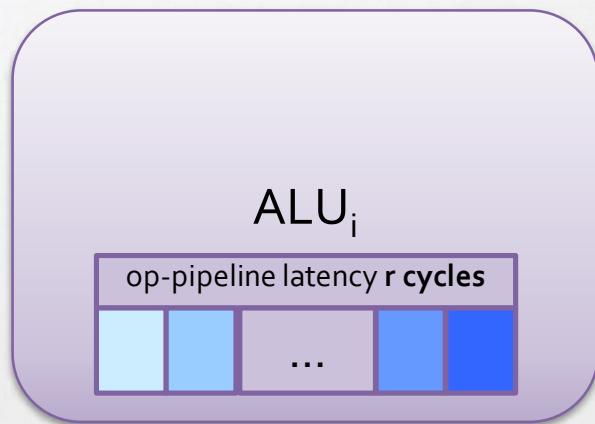
# GPU

## Architettura di una GPU: Più Livelli di Parallelismo



# GPU

## Architettura di una GPU: Più Livelli di Parallelismo



**Livello più basso**  
Ogni ALU è pipelined

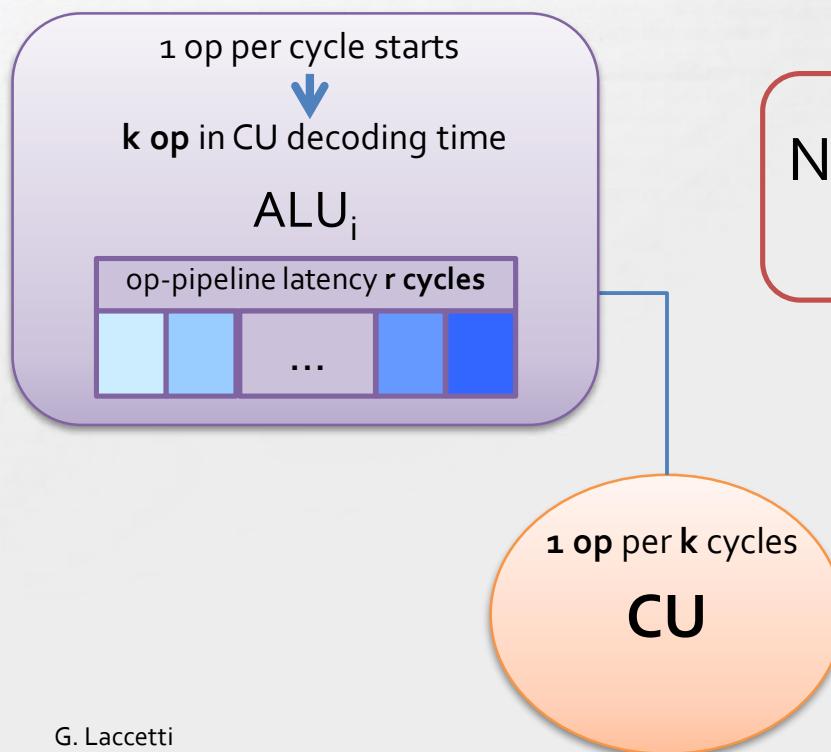


latenza di pipeline di **r** cicli.

# GPU

## Quanti thread simultanei?

La CU dell'MP impiega  **$k < r$  cicli** per la decodifica di un'istruzione.  
(Consideriamo che a un'istruzione corrisponda un'operazione).



Nella pipeline di ogni ALU può entrare **1 operazione** per ciclo.

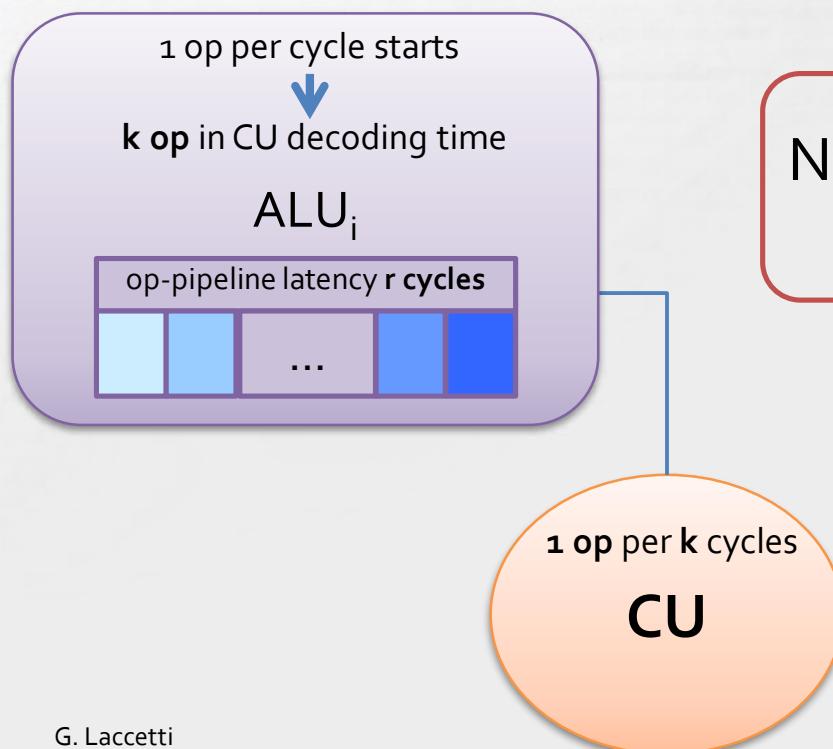
per ogni istruzione decodificata possono partire **k operazioni**

# GPU

## Quanti thread simultanei?

Se la CU impiega **k cicli a decodificare**, ogni k cicli entra in pipeline un'istruzione diversa.  
Se ad ogni ciclo può partire un'istruzione in pipeline, allora tra una decodifica e l'altra escono dalla pipeline (**terminano**) k istruzioni già inserite.  
E' come aver eseguito contemporaneamente k thread.

La CU dell'MP impiega **k< r cicli** per la decodifica di un'istruzione.  
(Consideriamo che a un'istruzione corrisponda un'operazione).



Nella pipeline di ogni ALU può entrare **1 operazione** per ciclo.



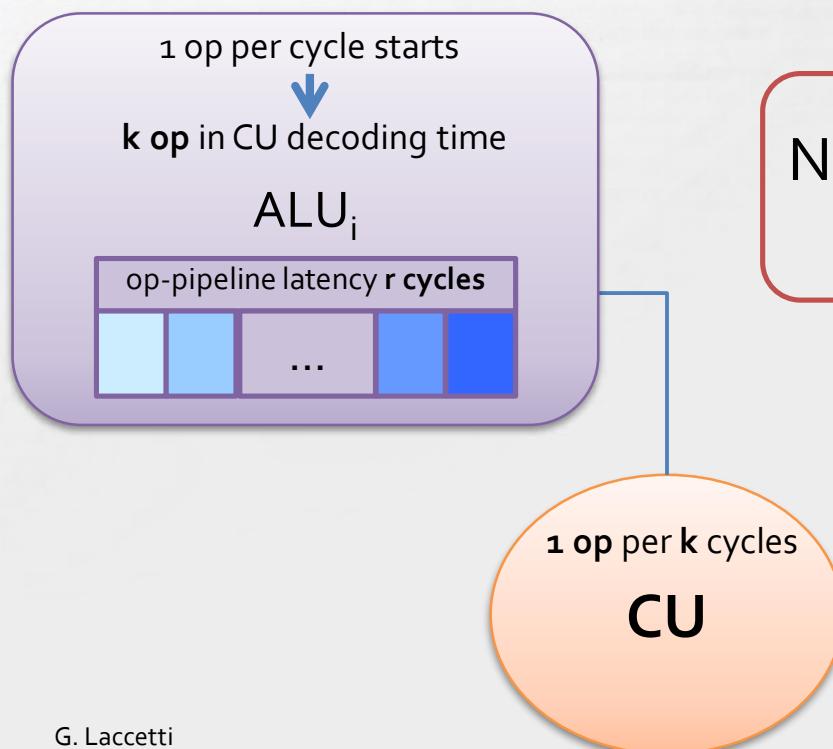
per ogni istruzione decodificata possono partire **k operazioni**

# GPU

## Quanti thread simultanei?

k si suppone minore di r (numero di cicli perché la pipeline vada a regime) perché se la CU ci mette più della latenza di pipeline a decodificare, la pipeline non è mai a regime.

La CU dell'MP impiega **k< r cicli** per la decodifica di un'istruzione.  
(Consideriamo che a un'istruzione corrisponda un'operazione).



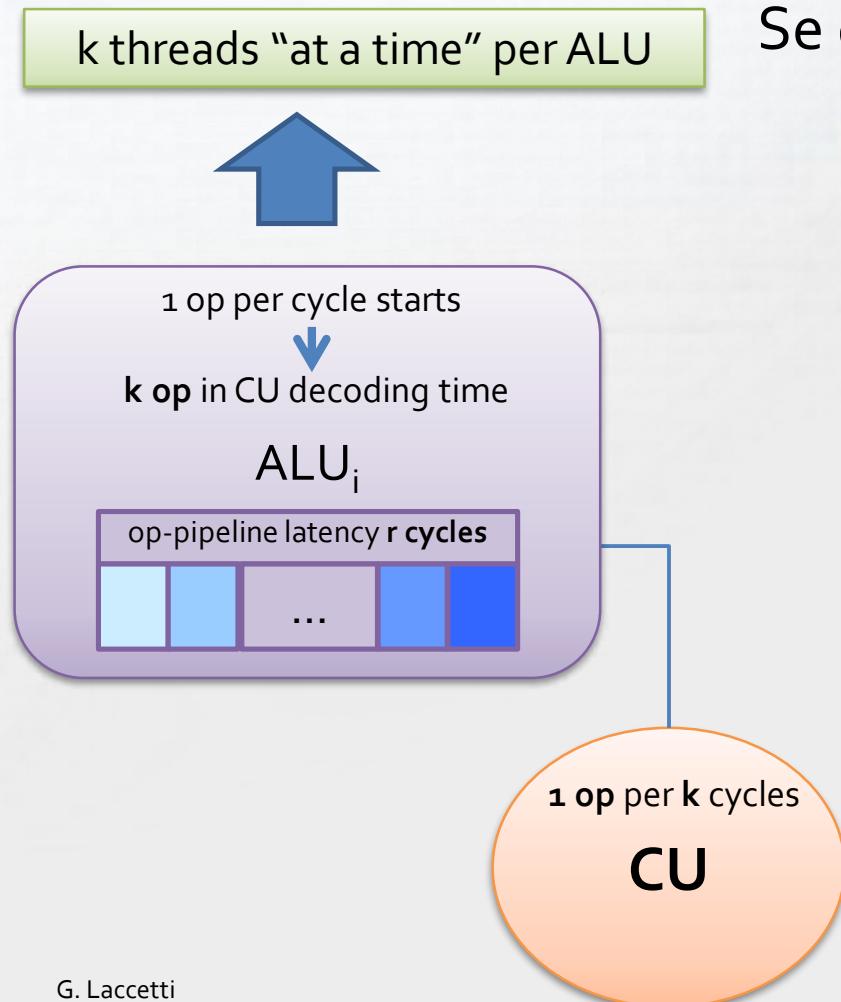
Nella pipeline di ogni ALU può entrare **1 operazione** per ciclo.



per ogni istruzione decodificata possono partire **k operazioni**

# GPU

## Quanti thread simultanei?



Se consideriamo che ogni operazione sia svolta da un thread diverso

1 operazione → 1 thread



per ogni istruzione decodificata  
**k thread eseguono l’operazione corrispondente**  
su ogni ALU

# GPU

## Quanti thread simultanei?

ALU<sub>i</sub>

# GPU

## Quanti thread simultanei?

ALU<sub>i</sub>

# GPU

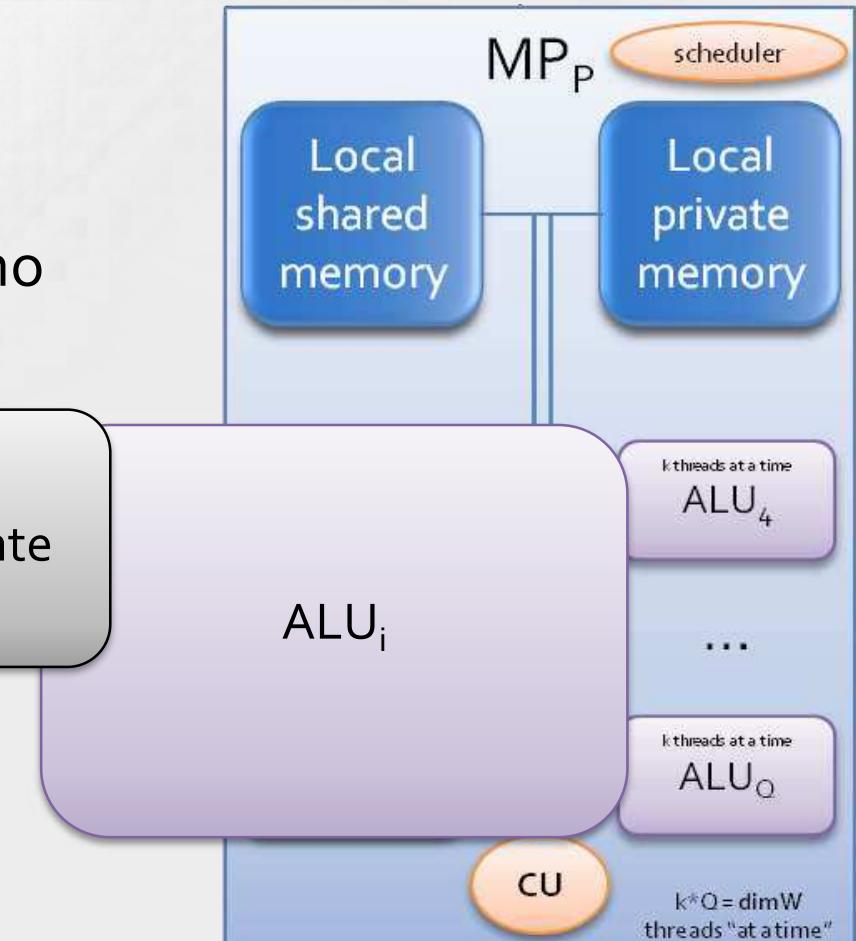
## Quanti thread simultanei?

Ogni MP ha Q ALU



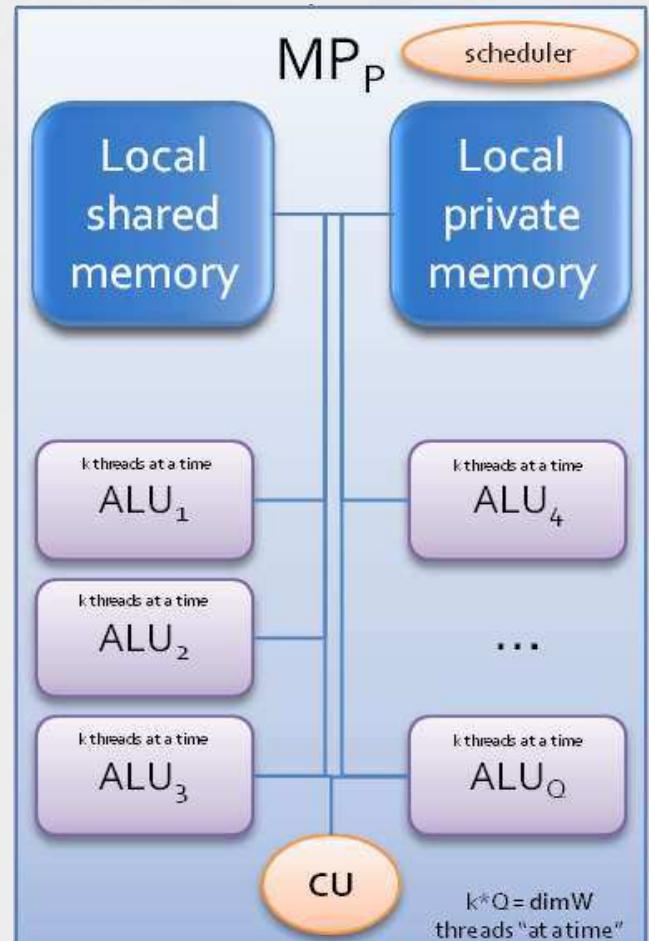
Per ogni istruzione sull'MP partono  
**k\*Q thread**

**Warp**  
gruppo di **k\*Q** thread simultaneamente  
eseguiti da un MP. **k\*Q=dimW**



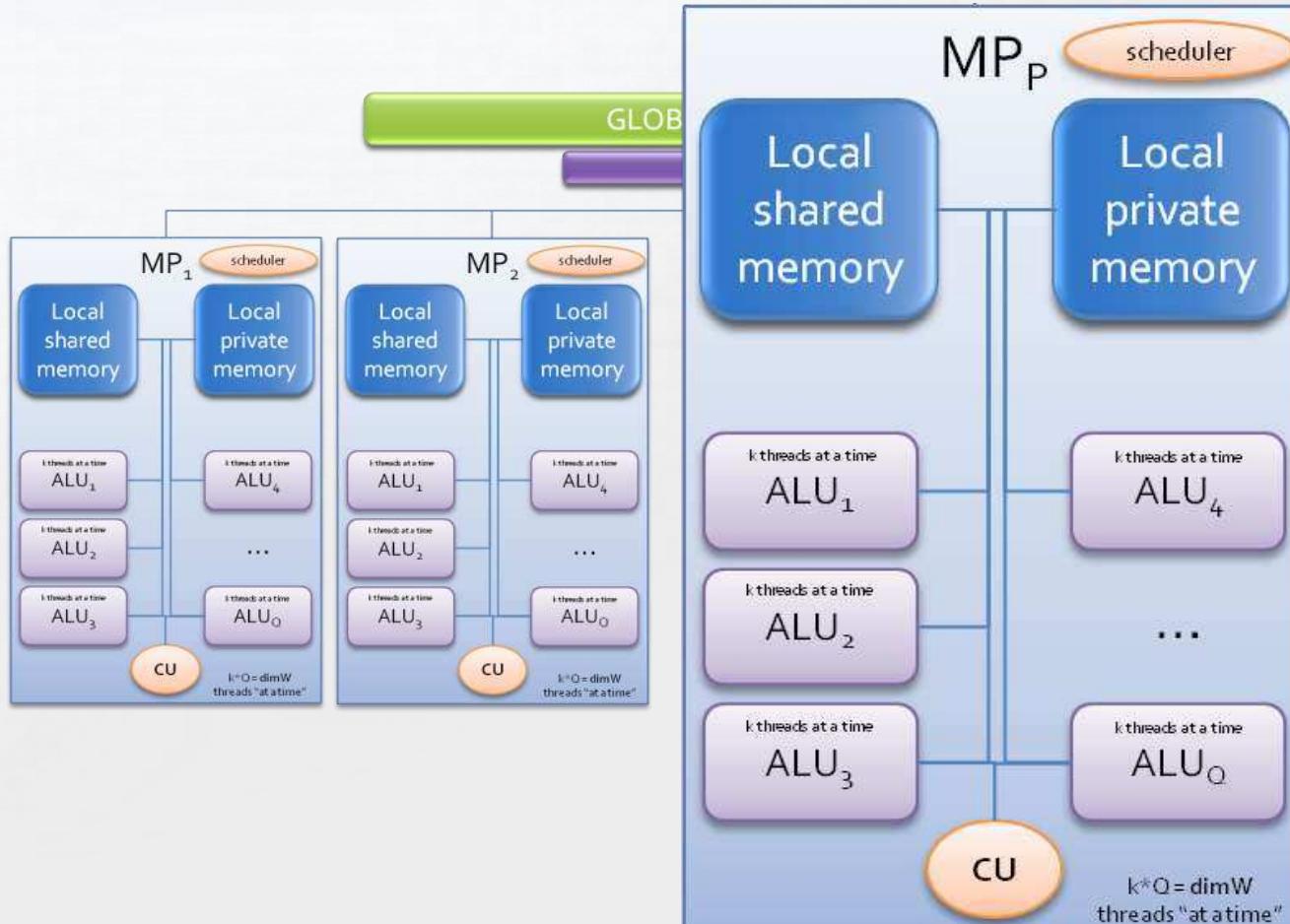
# GPU

## Quanti thread simultanei?



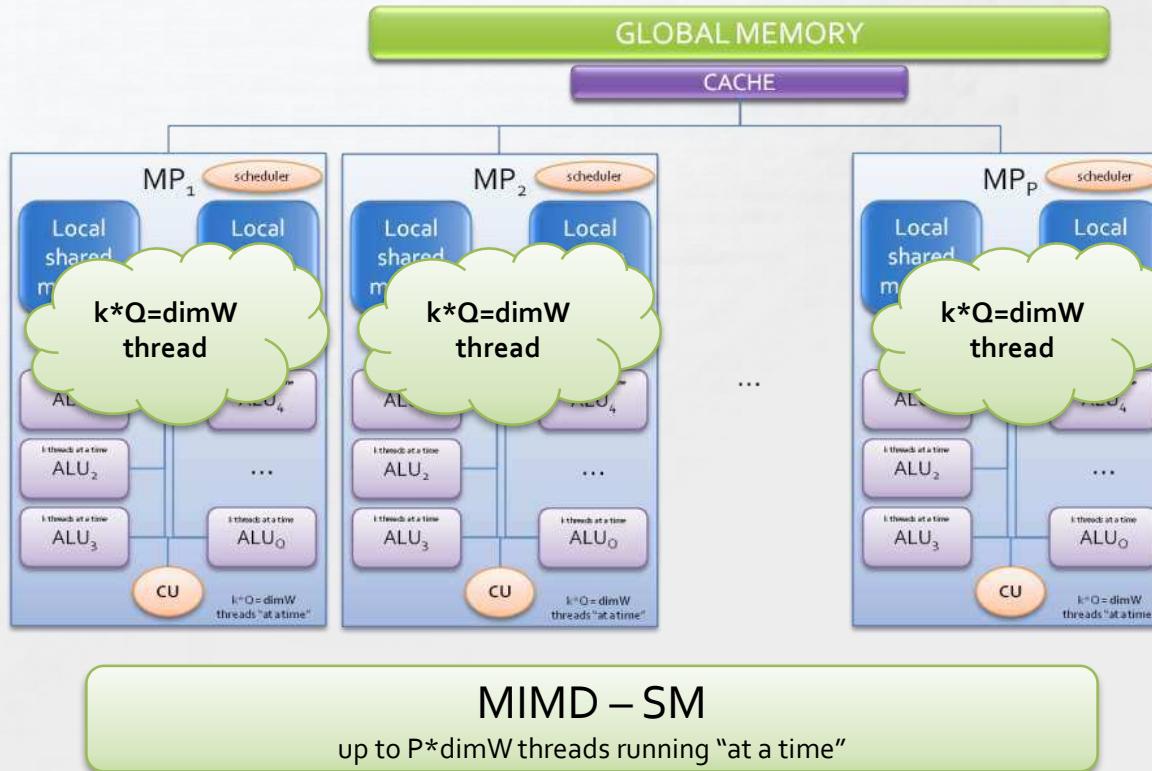
# GPU

## Quanti thread simultanei?



# GPU

## Quanti thread simultanei?



Ogni GPU ha  $P$  MP  
↓  
Per ogni istruzione  
sulla GPU partono  
fino a  
 **$P \cdot \text{dimW} = P \cdot k \cdot Q$**   
thread  
contemporaneamente

# GPU

## Tempo di calcolo

P core con Q unità  
Pipeline con k segmenti  
k thread per ogni unità

Ogni thread esegue istruzioni tutte dipendenti tra loro.

Algoritmo puramente parallelo  
T tempo di esecuzione sequenziale

IDEALE!!

**Tempo d'esecuzione algoritmo per GPU**

$$T_{GPU}(P \cdot Q \cdot k, N) = \frac{T(N)}{P \cdot Q \cdot k} \cdot t_{calc}$$

N dimensione del problema

A questo tempo va aggiunto quello di accesso alla memoria

# GPU

## Tempo di calcolo

P core con Q unità  
Pipeline con k segmenti  
k thread per ogni unità

Ogni thread esegue istruzioni tutte dipendenti tra loro.

Algoritmo puramente parallelo  
 $T_c$  tempo di esecuzione concorrente

IDEALE!!

Tempo d'esecuzione algoritmo per GPU

$$T_{GPU}(P \cdot Q \cdot k, N) = \frac{T_c(N)}{P \cdot Q \cdot k} \cdot t_{calc}$$

N dimensione del problema

Lanciando l'esecuzione di un numero di thread molto maggiore di  $P \cdot Q \cdot k$  (possibilmente multiplo) per la **stessa dimensione N**, il tempo di calcolo teoricamente aumenta, ma quello d'esecuzione totale **NO**: si copre la latenza di memoria

# GPU

## Tempo di calcolo

P core con Q unità  
Pipeline con k segmenti  
k thread per ogni unità

Ogni thread esegue istruzioni tutte dipendenti tra loro.

Algoritmo puramente parallelo  
 $T_c$  tempo di esecuzione concorrente

IDEALE!!

Tempo d'esecuzione algoritmo per GPU

$$T_{GPU}(P \cdot Q \cdot k, N) = \frac{T_c(N)}{P \cdot Q \cdot k} \cdot t_{calc}$$

N dimensione del problema

Lanciando l'esecuzione di un numero di thread molto maggiore di  $P \cdot Q \cdot k$  (possibilmente multiplo) **aumentando** proporzionalmente la dimensione **N**, il tempo di calcolo teoricamente aumenta, ma quello d'esecuzione totale **NO**: si copre la latenza di memoria

# GPU

## Tempo di calcolo

P core con Q unità  
Pipeline con k segmenti  
k thread per ogni unità

Ogni thread esegue istruzioni tutte dipendenti tra loro.

Algoritmo puramente parallelo  
 $T_c$  tempo di esecuzione concorrente

IDEALE!!

Tempo d'esecuzione algoritmo per GPU

$$T_{GPU}(P \cdot Q \cdot k, N) = \frac{T_c(N)}{P \cdot Q \cdot k} \cdot t_{calc}$$

N dimensione del problema

Risolviamo un problema di dimensione maggiore nello stesso tempo d'esecuzione  
Cioè le possibilità di **scalabilità** sono maggiori di quello che ci si aspetterebbe dal numero di unità

# GPU

## Tempo di calcolo

P core con Q unità  
Pipeline con k segmenti  
k thread per ogni unità

Ogni thread esegue istruzioni tutte dipendenti tra loro.

Algoritmo puramente parallelo  
 $T_c$  tempo di esecuzione concorrente

IDEALE!!

Tempo d'esecuzione algoritmo per GPU

$$T_{GPU}(P \cdot Q \cdot k, N) = \frac{T_c(N)}{P \cdot Q \cdot k} \cdot t_{calc}$$

N dimensione del problema

**ASSUNZIONE FONDAMENTALE:** Algoritmo perfettamente parallelo

Il carico deve essere molto ben bilanciato tra le unità: ottime performance in applicazioni che prevedono le stesse operazioni su dati diversi (come appunto il rendering)

# Compilare ed eseguire sorgente CUDA

- Per compilare basta digitare nella shell:

```
nvcc -o nome-eseguibile nome-codice.cu
```

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

```
./nome-eseguibile
```

# Installare CUDA in locale

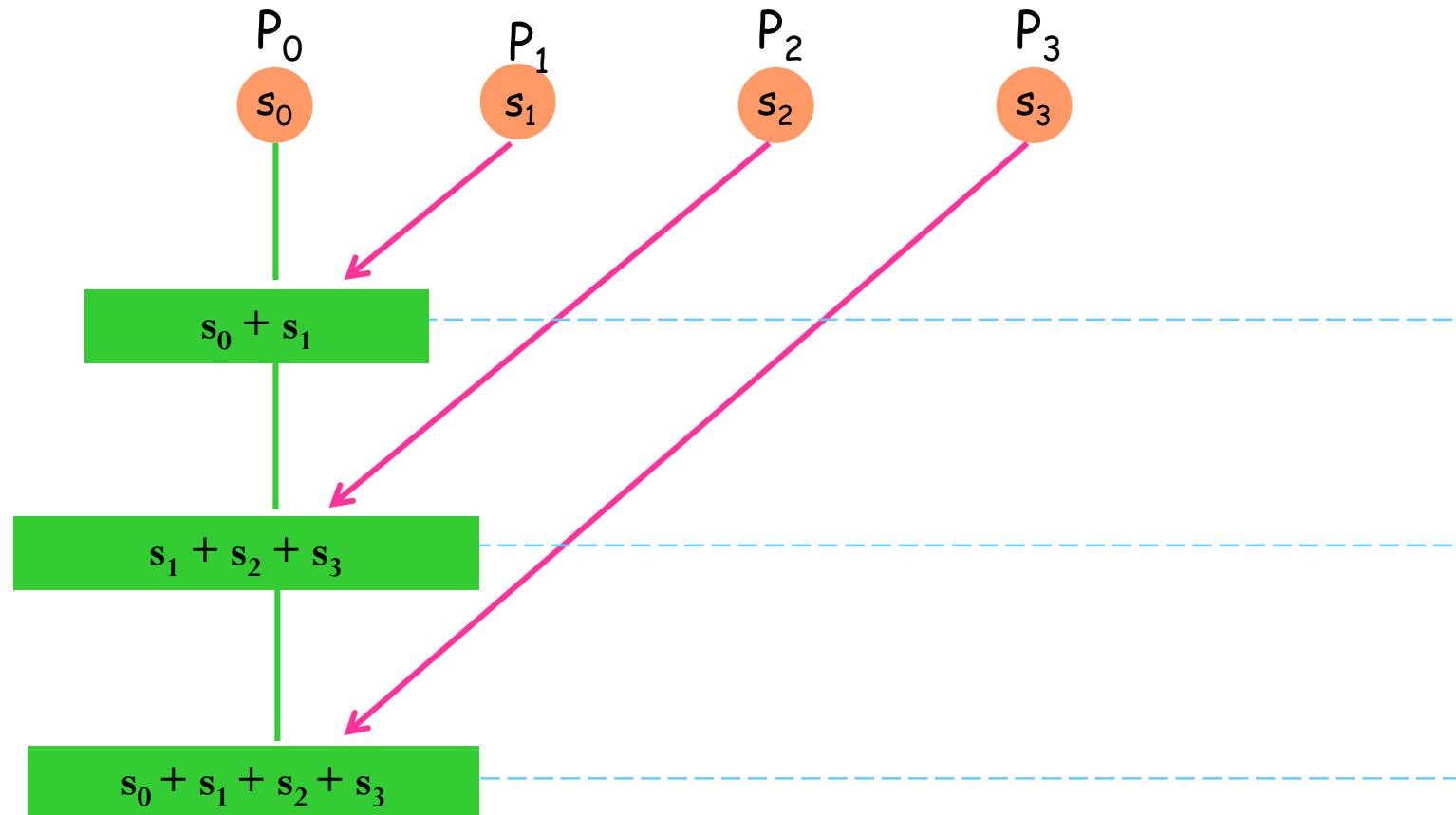
- Recarsi a questa pagina e seguire le istruzioni a video:  
**<https://developer.nvidia.com/cuda-downloads>**
- Tutta la documentazione CUDA è disponibile a questa pagina  
**<https://docs.nvidia.com/cuda/>**

# Calcolo Speed Up ed Efficienza

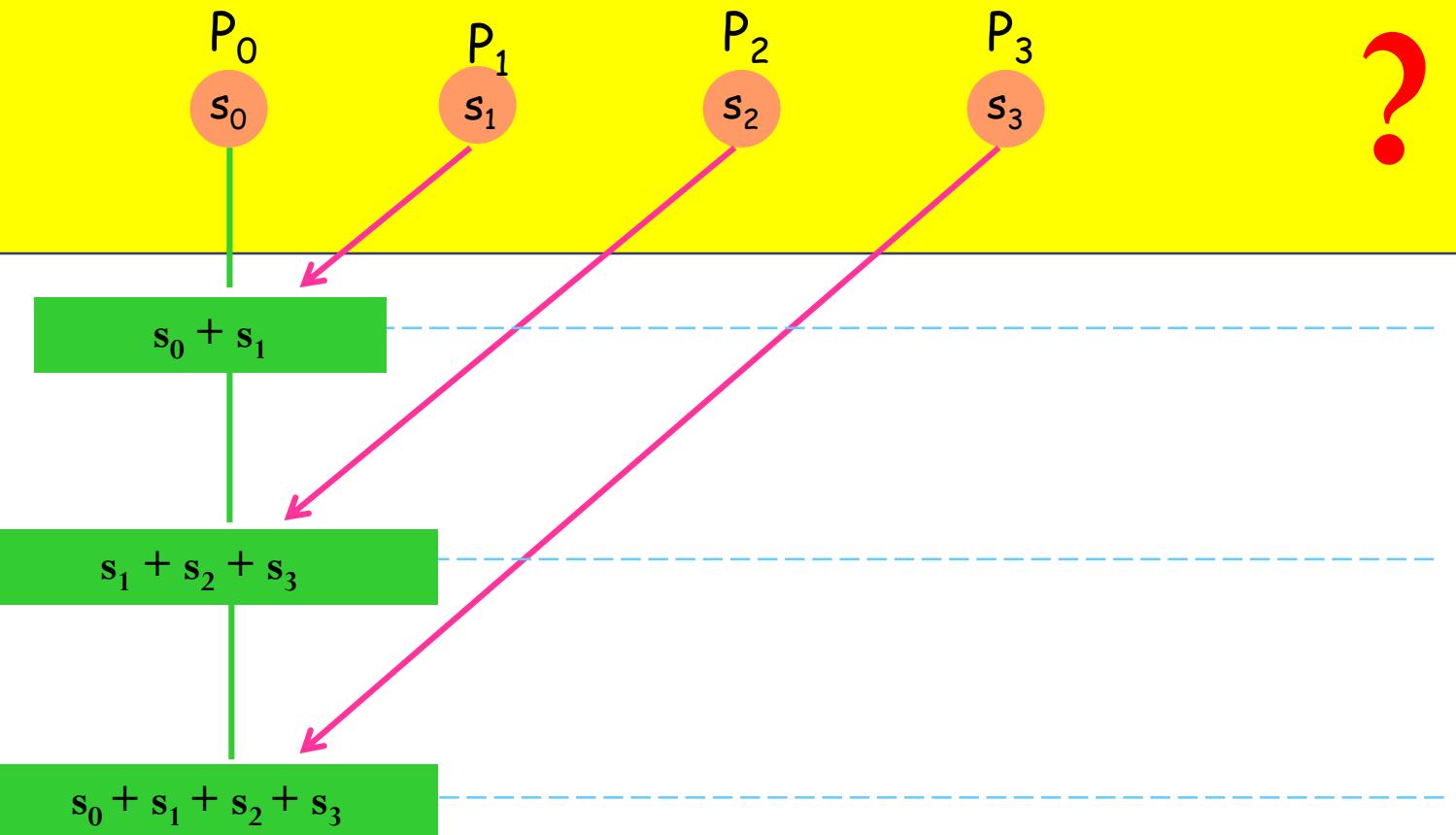
## con la legge di Ware-Amdahl

Somma di N numeri

# Strategia I - Quanti calcoli?



# Strategia I - Quanti calcoli?



# Strategia I - Quanti calcoli?

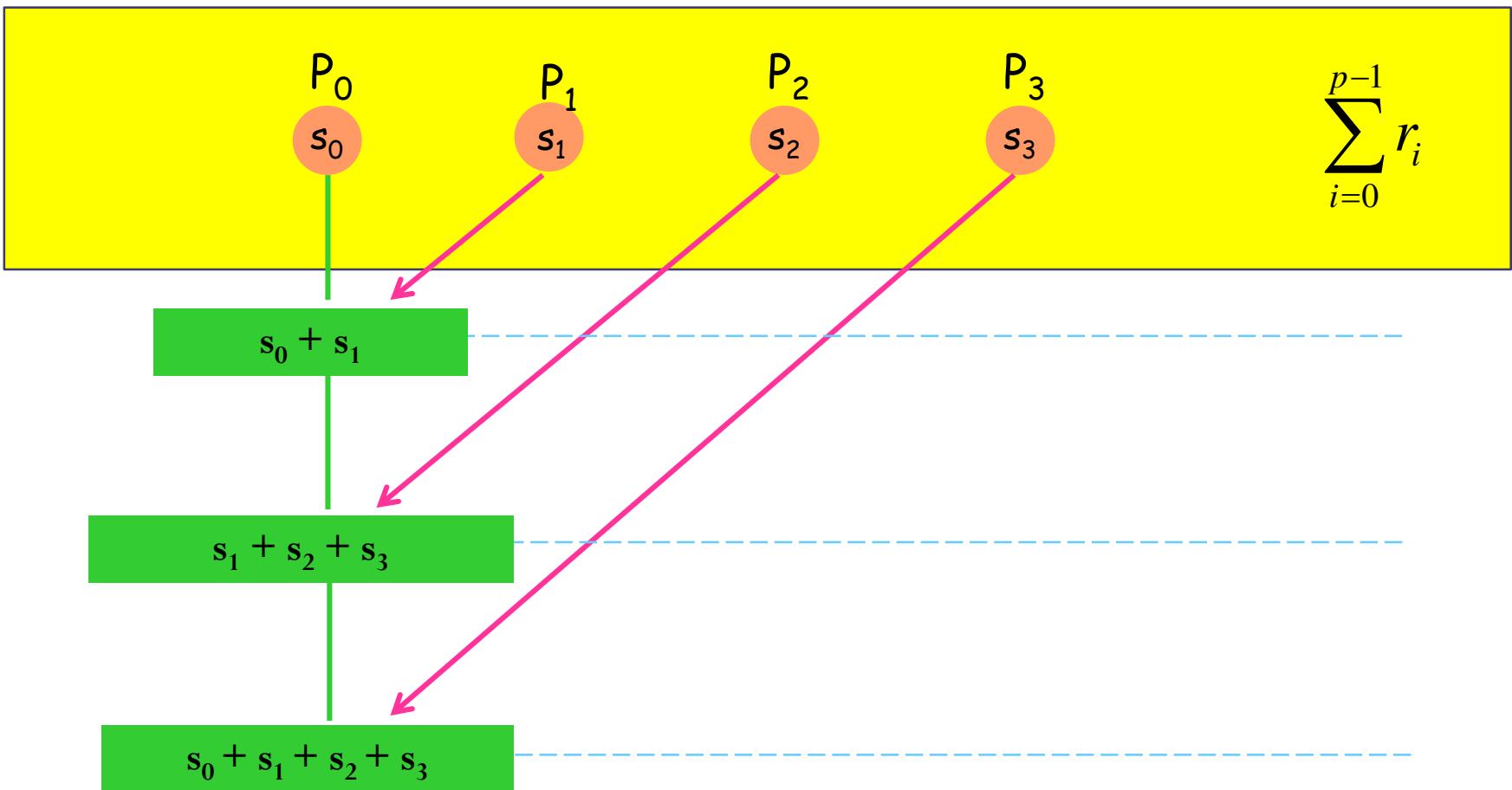
Dati :  $n (\geq 2)$  numeri,  $p$  processi, con  $p \leq 2n$

Per le somme parziali il processore  $i$  esegue un numero di operazioni pari a:

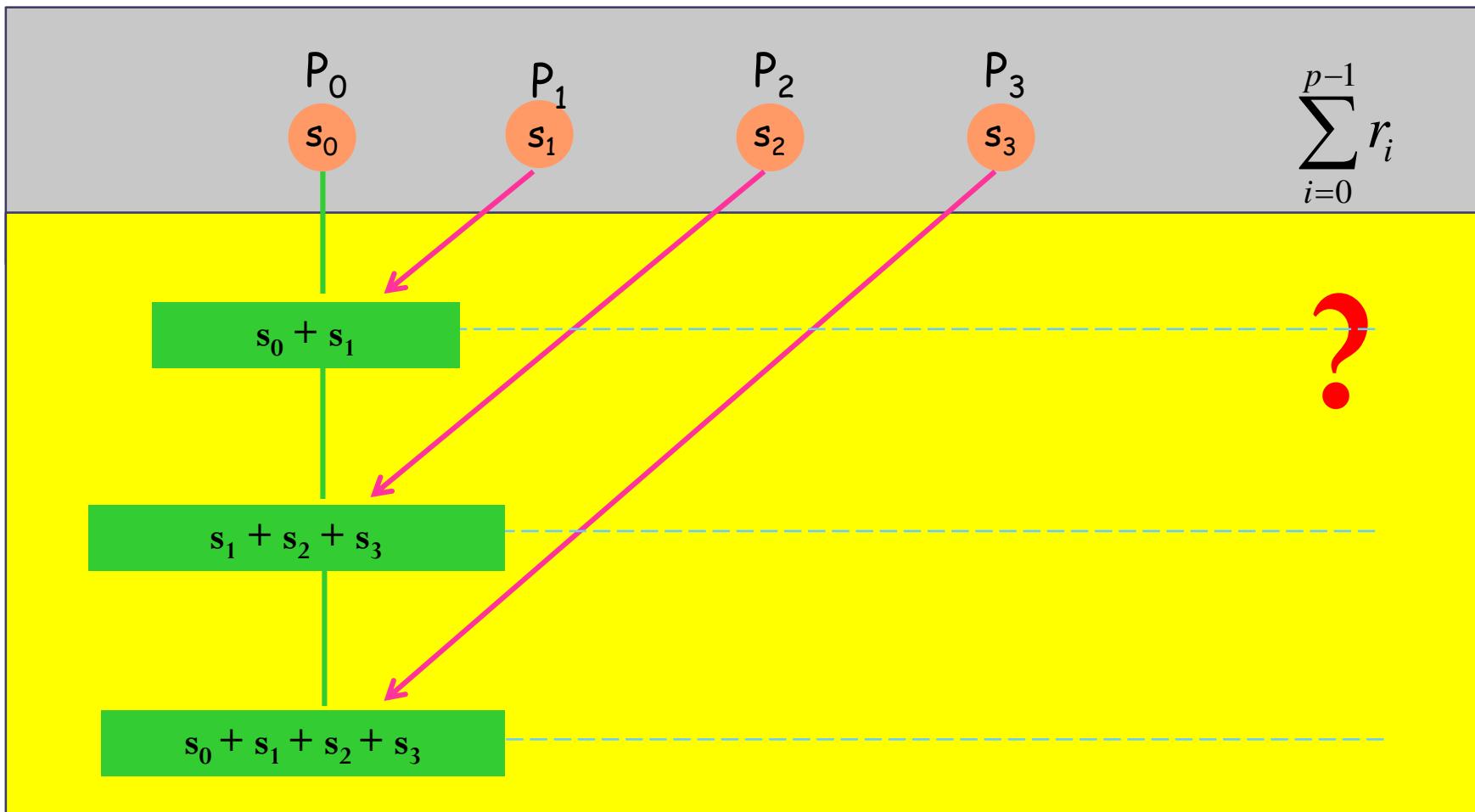
$$r_i = \begin{cases} \left(\frac{n}{p}\right) - 1 & \text{se } i \geq n \% p \\ \left(\frac{n}{p} + 1\right) - 1 & \text{se } i < n \% p \end{cases}$$

Per un totale di  $\sum_{i=0}^{p-1} r_i$  operazioni

# Strategia I - Quanti calcoli?



# Strategia I - Quanti calcoli?



# Strategia I - Quanti calcoli?

Dati : **n** ( $\geq 2$ ) numeri, **p** processi, con  $p \leq 2n$

Seguono  $p-1$  altre fasi di calcolo.

Nella i-ma fase di calcolo un processore solo compie 1 operazione (somma) , per un totale di

$$\sum_{i=1}^{p-1} 1 = p - 1 \text{ operazioni}$$

# Strategia I - Quanti calcoli?

Dati :  $n (\geq 2)$  numeri,  $p$  processi, con  $p \leq 2n$

In totale allora vengono eseguite

$$\sum_{i=0}^{p-1} r_i + (p-1) \text{ operazioni}$$

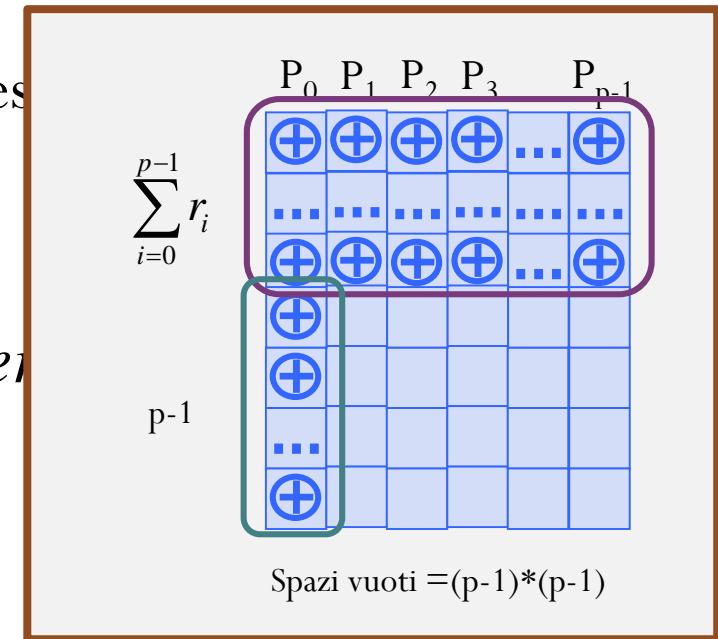
- Di queste:
  - $\sum_{i=0}^{p-1} r_i$  vengono eseguite in parallelo, cioè per la loro esecuzione lavorano contemporaneamente più di un processore
  - $(p-1)$  vengono eseguite invece da un solo processore

# Strategia I - Quanti calcoli?

Dati :  $n (\geq 2)$  numeri,  $p$  processori

In totale allora vengono eseguite

$$\sum_{i=0}^{p-1} r_i + (p-1) \text{ operazioni}$$



- Di queste:
  - $\sum_{i=0}^{p-1} r_i$  vengono eseguite in parallelo, cioè per la loro esecuzione lavorano contemporaneamente più di un processore
  - $(p-1)$  vengono eseguite invece da un solo processore

# Strategia I – Legge di Ware-Amdahl

Per la legge di Ware-Amdahl

$$S(p) = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

$\alpha$  parte sequenziale, ovvero frazione di operazioni eseguite da un solo processore

$1-\alpha$  parte parallela, ovvero frazione di operazioni eseguite contemporaneamente da più di un processore

- Nel nostro caso:

$$\alpha = \frac{p-1}{\sum_{i=0}^{p-1} r_i + (p-1)}$$

$$1-\alpha = 1 - \frac{p-1}{\sum_{i=0}^{p-1} r_i + (p-1)} = \frac{\sum_{i=0}^{p-1} r_i}{\sum_{i=0}^{p-1} r_i + (p-1)}$$

# Strategia I - Esempio

Dati : **50** numeri, 4 processi

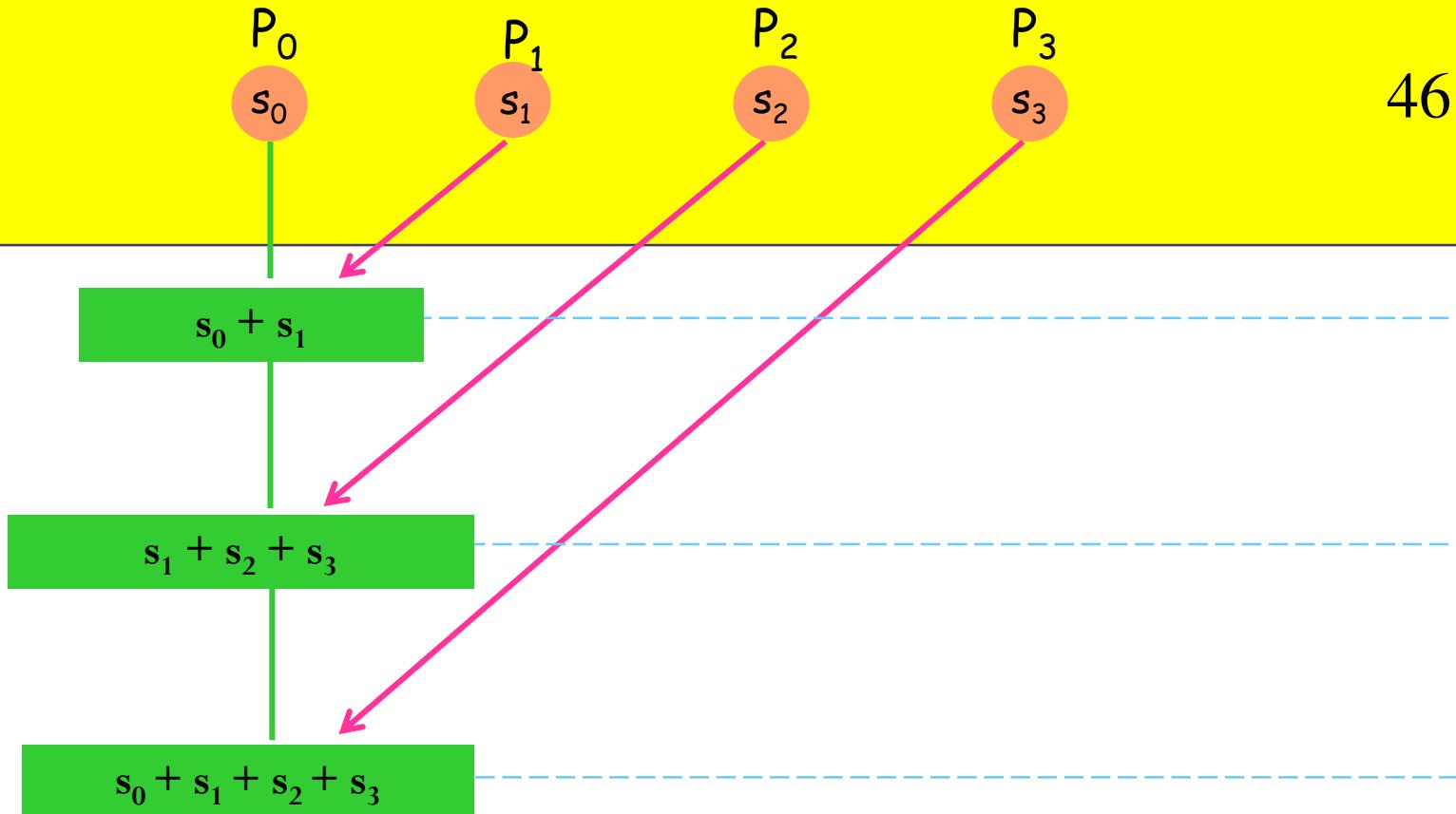
Per le somme parziali il processore i esegue un numero di operazioni pari a:

$$r_i = \begin{cases} \left(\frac{50}{4}\right) - 1 = 11 & se \quad i \geq n \% p = 2 \\ \left(\frac{50}{4} + 1\right) - 1 = 12 & se \quad i < n \% p = 2 \end{cases}$$

Per un totale di  $12 + 12 + 11 + 11 = 46$  *operazioni*

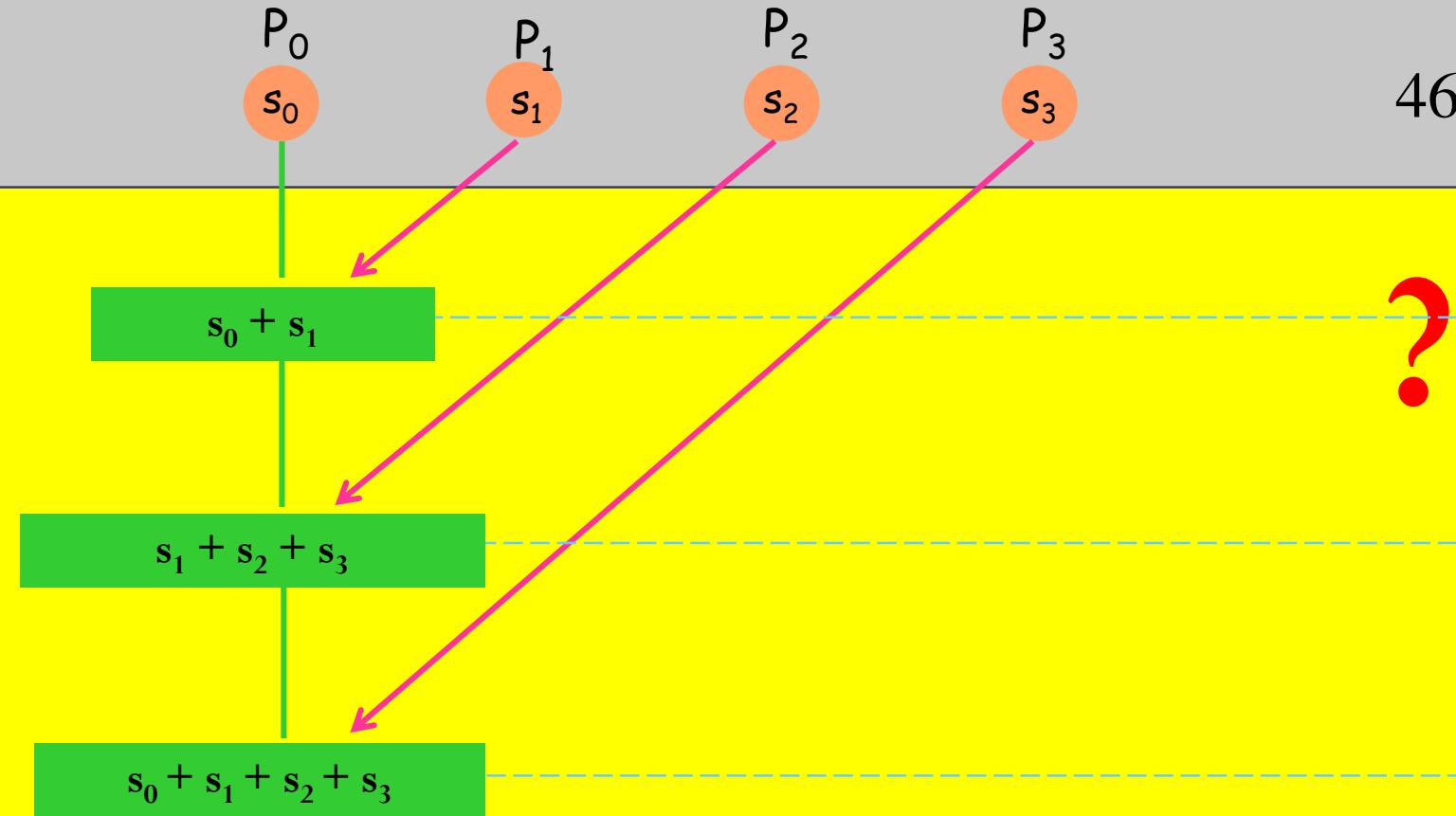
# Strategia I - Quanti calcoli?

46

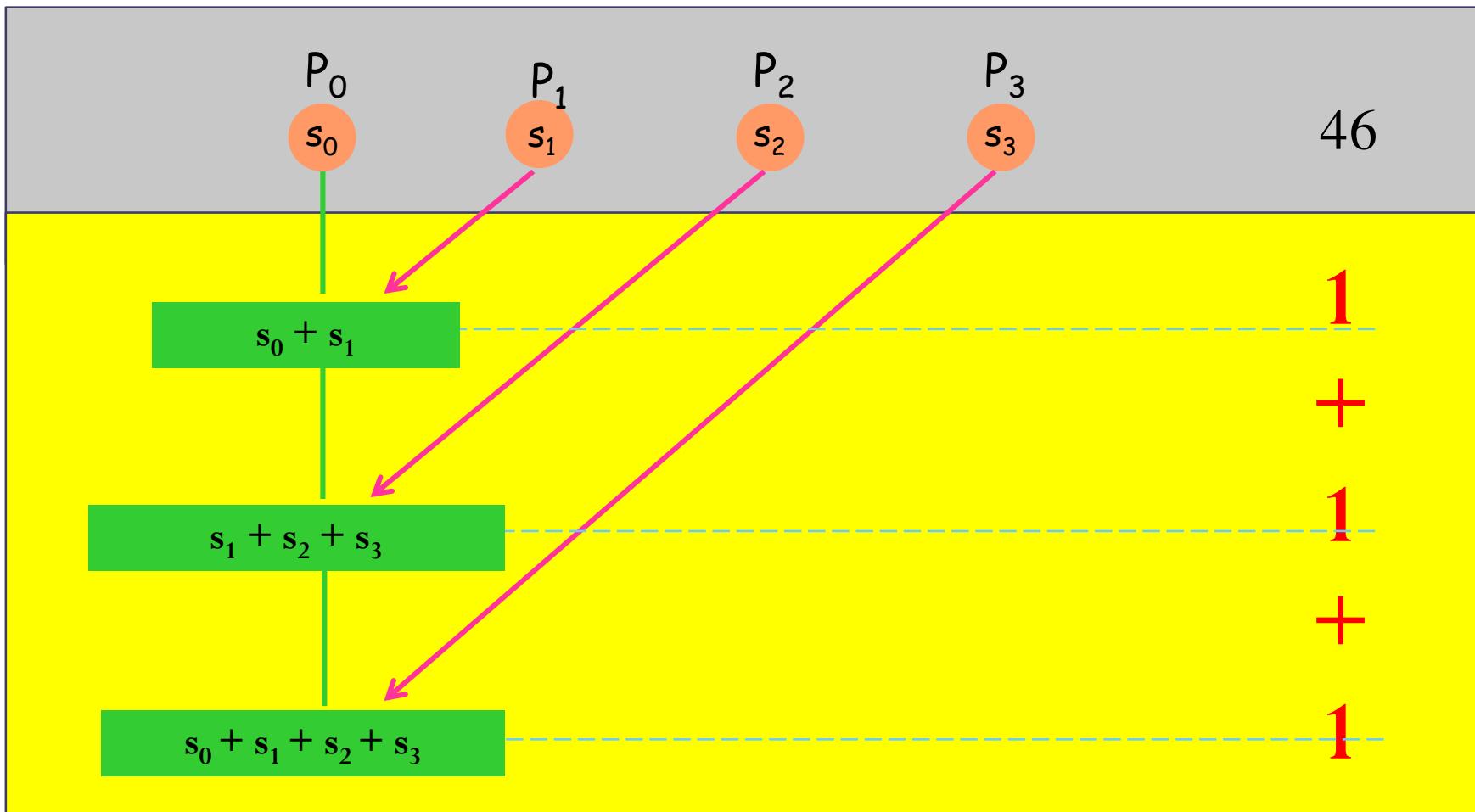


# Strategia I - Quanti calcoli?

46



# Strategia I - Quanti calcoli?



# Strategia I - Quanti calcoli?

Dati : **50** numeri, 4 processi

In totale allora vengono eseguite

49 *operazioni*

- Di queste:
  - 46 vengono eseguite in parallelo, cioè per la loro esecuzione lavorano contemporaneamente più di un processore
  - 3 vengono eseguite invece da un solo processore

# Strategia I – Legge di Ware-Amdahl

Per la legge di Ware-Amdahl

$$S(p) = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

$\alpha$  parte sequenziale, ovvero frazione di operazioni eseguite da un solo processore

$1-\alpha$  parte parallela, ovvero frazione di operazioni eseguite contemporaneamente da più di un processore

- Nel nostro caso:

$$\alpha = \frac{3}{49} \quad 1 - \alpha = \frac{46}{49}$$

# Strategia I – Legge di Ware-Amdahl

Per la legge di Ware-Amdahl

$$S(p) = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

$\alpha$  parte sequenziale, ovvero frazione di operazioni eseguite da un solo processore

$1-\alpha$  parte parallela, ovvero frazione di operazioni eseguite contemporaneamente da più di un processore

- Nel nostro caso:

$$\alpha = \frac{3}{49} = 0,06$$

$$1 - \alpha = \frac{46}{49} = 0,93$$

# Strategia I – Legge di Ware-Amdahl

Per la legge di Ware-Amdahl

$$S(4) = \frac{1}{0,06 + \frac{0,93}{4}}$$

$\alpha$  parte sequenziale, ovvero frazione di operazioni eseguite da un solo processore

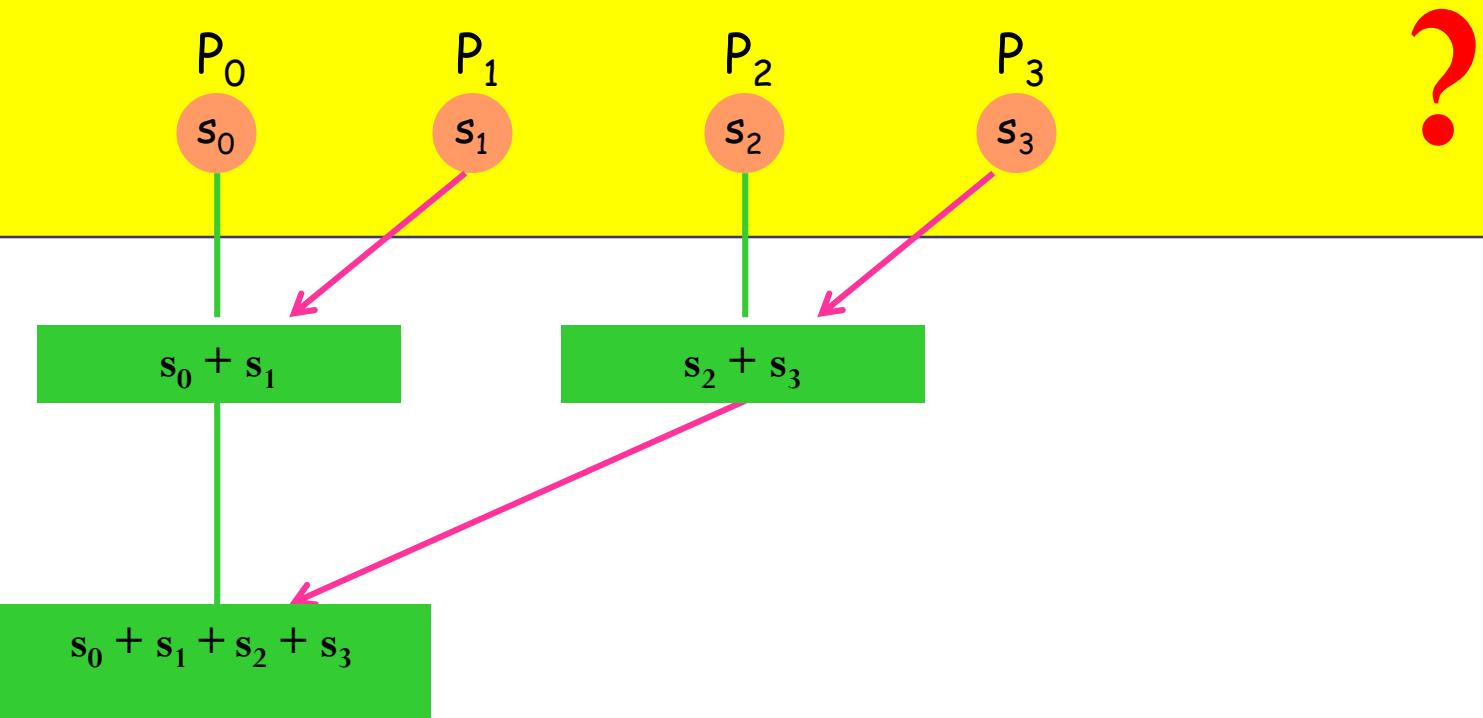
$1-\alpha$  parte parallela, ovvero frazione di operazioni eseguite contemporaneamente da più di un processore

- Nel nostro caso:

$$\alpha = \frac{3}{49} = 0,06$$

$$1 - \alpha = \frac{46}{49} = 0,93$$

# Strategia II - Quanti calcoli?



# Strategia II - Quanti passi di calcolo?

Dati :  $n$  ( $\geq 2$ ) numeri,  $p = 2^k$  processi, con  $k \leq \log_2 n$

Per le somme parziali il processore  $i$  esegue un numero di operazioni pari a:

$$r_i = \begin{cases} \left(\frac{n}{p}\right) - 1 & \text{se } i \geq n \% p \\ \left(\frac{n}{p} + 1\right) - 1 & \text{se } i < n \% p \end{cases}$$

Per un totale di  $\sum_{i=0}^{p-1} r_i$  operazioni

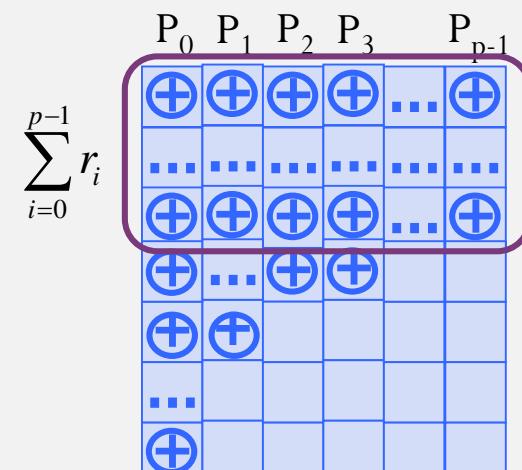
# Strategia II - Quanti passi di calcolo?

Dati :  $n (\geq 2)$  numeri,  $p = 2^k$  processori

Per le somme parziali il processore  $i$  esegue  $r_i$  operazioni pari a:

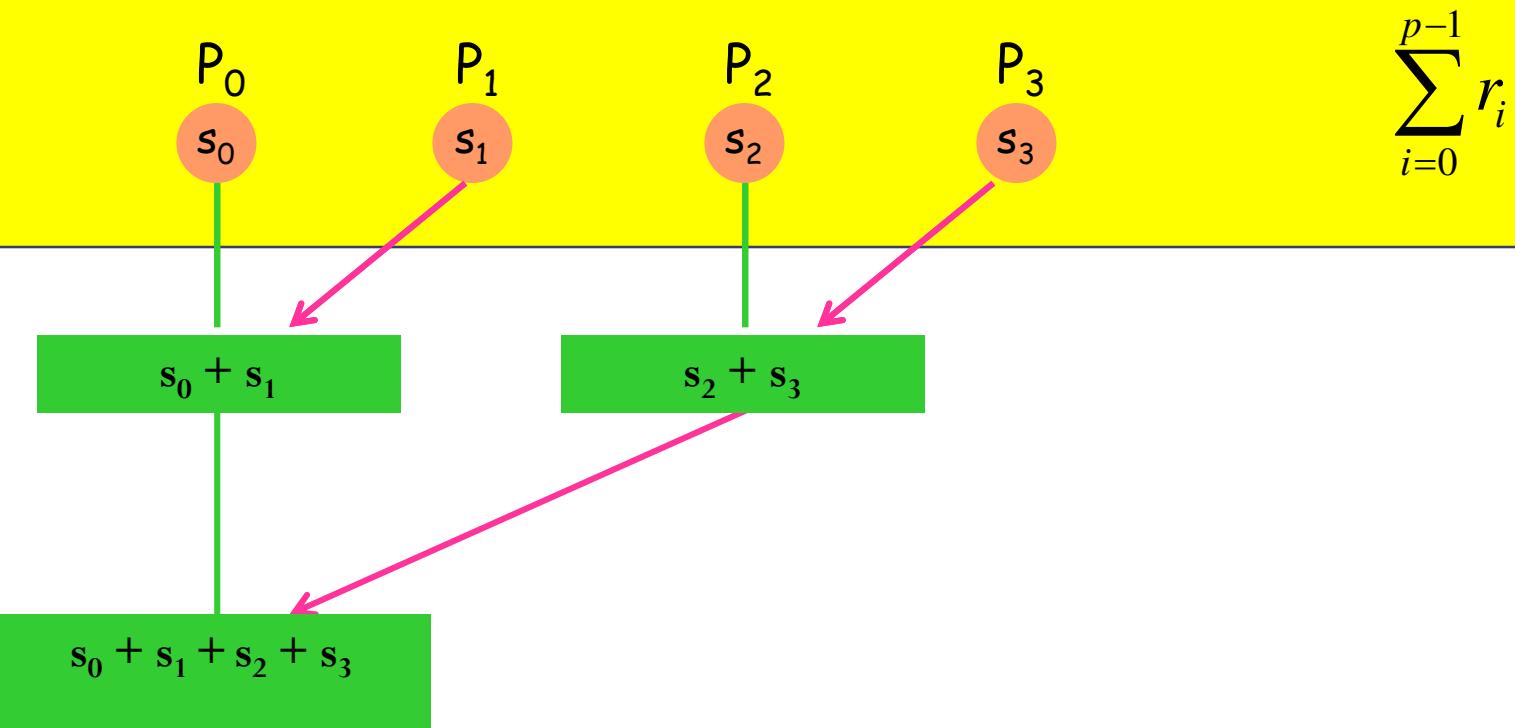
$$r_i = \begin{cases} \left(\frac{n}{p}\right) - 1 & se \quad i \geq n \% p \\ \left(\frac{n}{p} + 1\right) - 1 & se \quad i < n \% p \end{cases}$$

Per un totale di  $\sum_{i=0}^{p-1} r_i$  operazioni

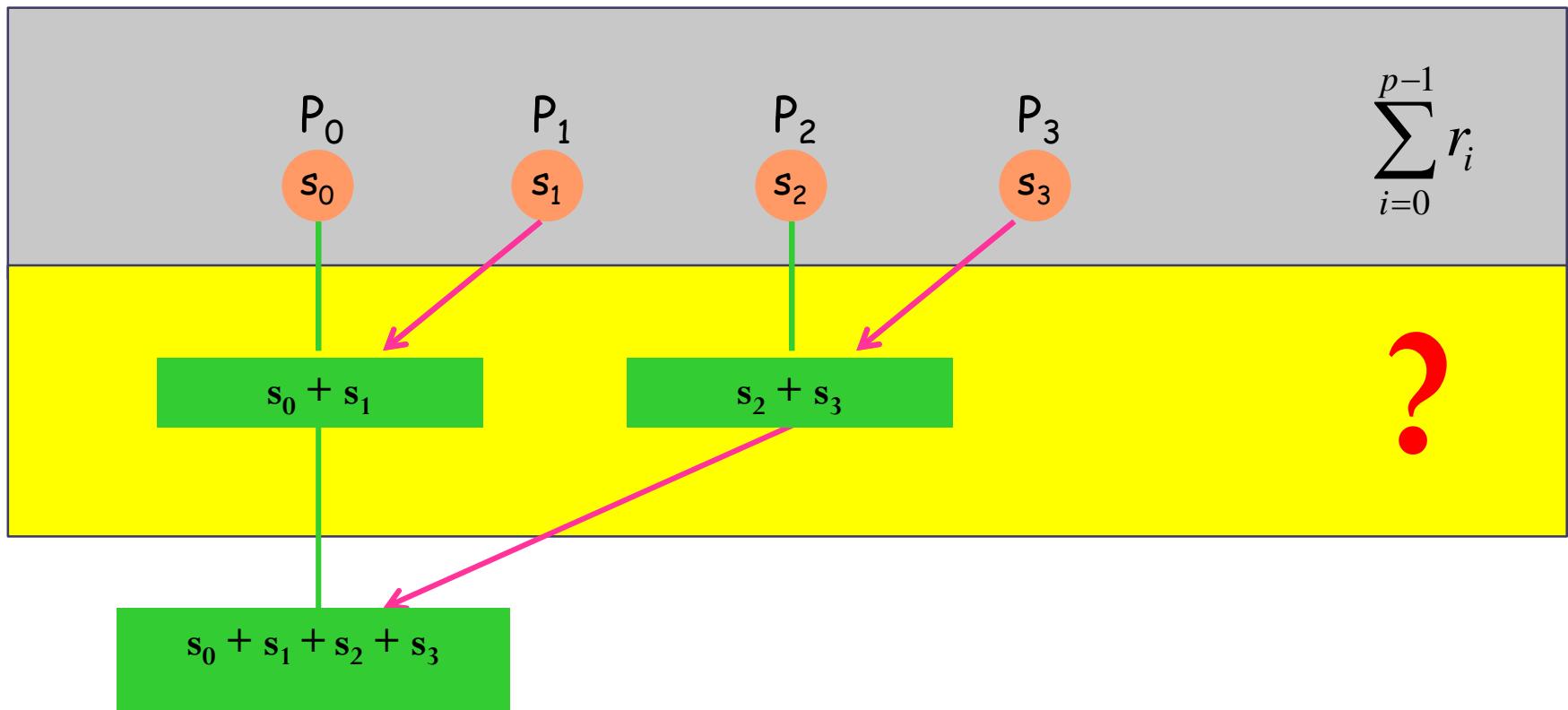


Spazi vuoti =  $p(p-1) - \log_2 p$

# Strategia II - Quanti calcoli?



# Strategia II - Quanti calcoli?



# Strategia II - Quanti calcoli?

Dati :  $n$  ( $\geq 2$ ) numeri,  $p = 2^k$  processi, con  $k \leq \log_2 n$

Seguono  $\log_2 p$  altre fasi di calcolo.

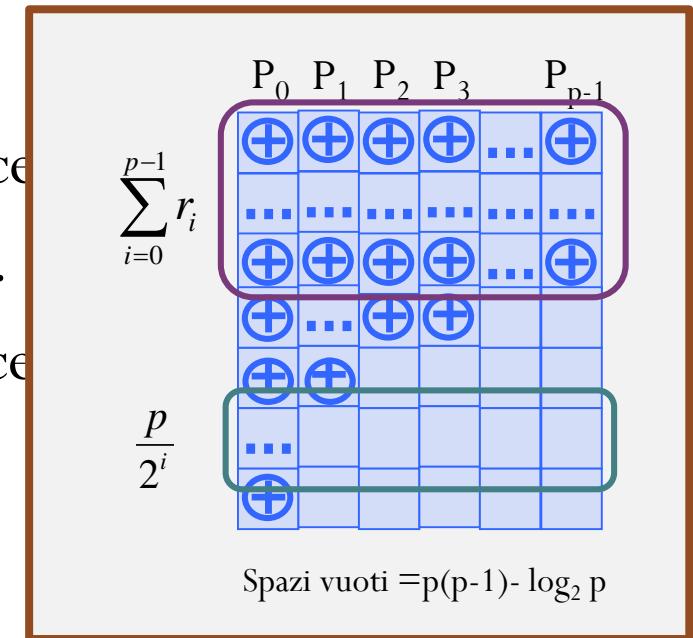
Nella i-ma fase di calcolo  $p/2^i$  processori eseguono 1 operazione (somma)

# Strategia II - Quanti calcoli?

Dati :  $n (\geq 2)$  numeri,  $p = 2^k$  processori

Seguono  $\log_2 p$  altre fasi di calcolo.

Nella i-ma fase di calcolo  $p/2^i$  processori eseguono operazione (somma)

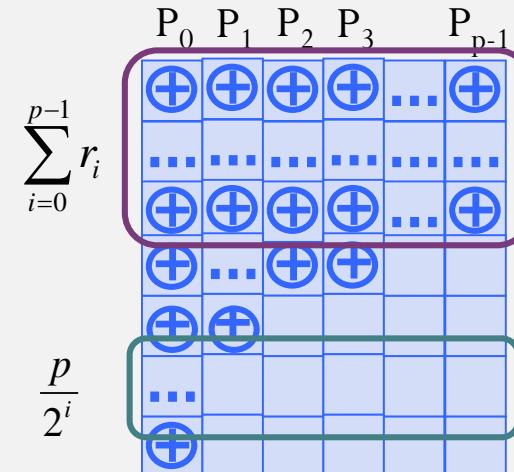


# Strategia II - Quanti calcoli?

Dati :  $n (\geq 2)$  numeri,  $p = 2^k$  processori

Seguono  $\log_2 p$  altre fasi di calcolo.

Nella i-ma fase di calcolo  $p/2^i$  processori eseguono l'operazione (somma)



Spazi vuoti =  $p(p-1) - \log_2 p$

Per un totale di

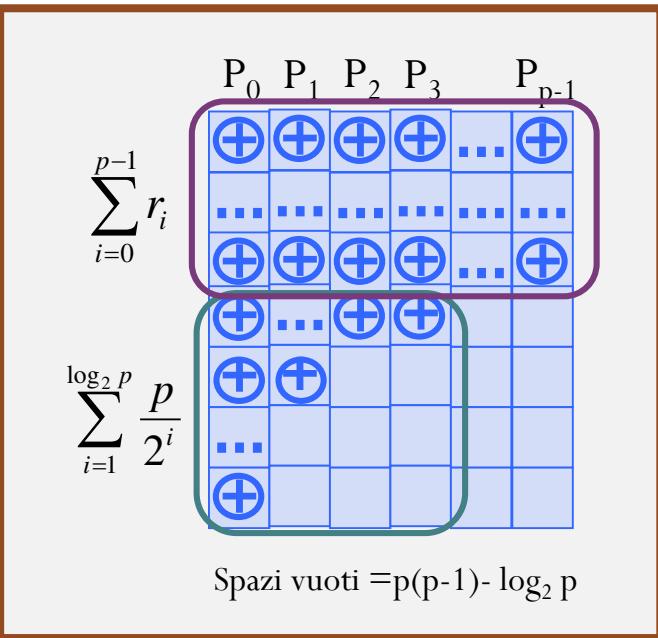
$$\sum_{i=1}^{\log_2 p} \frac{p}{2^i} \text{ operazioni}$$

# Strategia II - Quanti calcoli?

Dati :  $n (\geq 2)$  numeri,  $p = 2^k$  processori

Seguono  $\log_2 p$  altre fasi di calcolo.

Nella i-ma fase di calcolo  $p/2^i$  processori eseguono un'operazione (somma)



Per un totale di

$$\sum_{i=1}^{\log_2 p} \frac{p}{2^i} \quad \text{operazioni}$$

# Strategia II - Quanti calcoli?

Dati :  $n$  ( $\geq 2$ ) numeri,  $p$  processi, con  $p \leq 2n$

In totale allora vengono eseguite

$$\sum_{i=0}^{p-1} r_i + \sum_{i=1}^{\log_2 p} \frac{p}{2^i} \quad \text{operazioni}$$

# Strategia II - Quanti calcoli?

Di queste:

- $\sum_{i=0}^{p-1} r_i$  vengono eseguite da  $p$  processori
- $p/2^i$  vengono eseguite invece da  $p/2^i$  processori
  - In particolare
$$p/2^{\log_2 p} = p/p = 1$$
 operazione viene eseguita da 1 proc

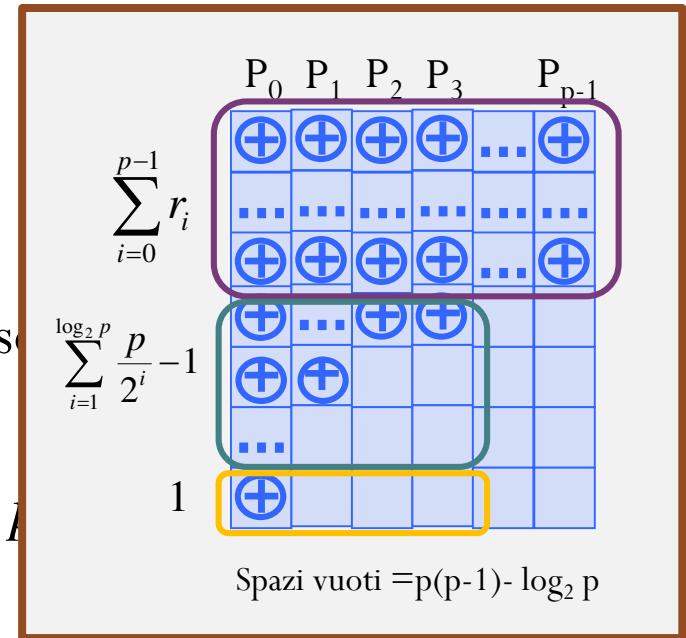
# Strategia II - Quanti calcoli?

Di queste:

- $\sum_{i=0}^{p-1} r_i$  vengono eseguite da p processi
- $p / 2^i$  vengono eseguite invece da 1

- In particolare

$$p / 2^{\log_2 p} = p / p = 1 \quad \text{operazione viene eseguita da 1 proc}$$



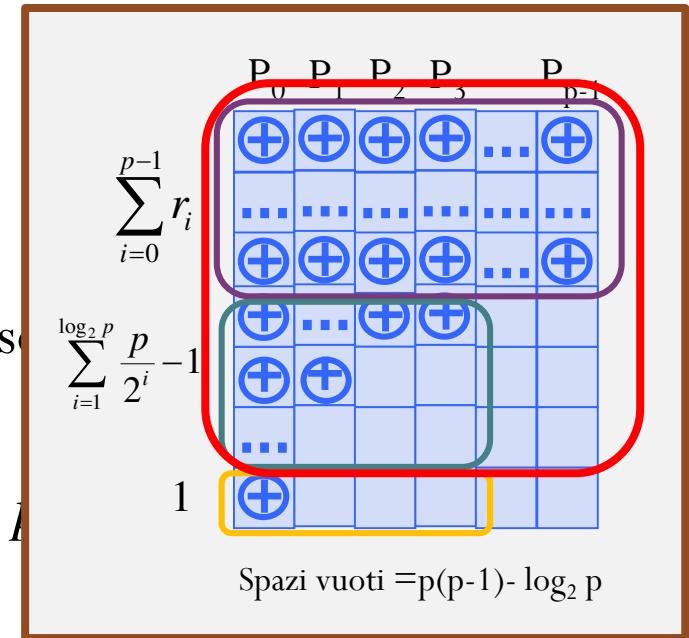
# Strategia II - Quanti calcoli?

Di queste:

- $\sum_{i=0}^{p-1} r_i$  vengono eseguite da  $p$  processori
- $p / 2^i$  vengono eseguite invece da  $1$  processore

- In particolare

$$p / 2^{\log_2 p} = p / p = 1 \quad \text{operazione viene eseguita da 1 proc}$$



# Strategia II – Legge di Ware-Amdahl

Per la legge di Ware-Amdahl

$$S(p) = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

$\alpha$  parte sequenziale, ovvero frazione di operazioni eseguite da un solo processore

$1-\alpha$  parte parallela, ovvero frazione di operazioni eseguite contemporaneamente da più di un processore

- Nel nostro caso:

$$\alpha = \frac{1}{\sum_{i=0}^{p-1} r_i + \sum_{i=1}^{\log_2 p} \frac{p}{2^i}}$$

$$1 - \alpha = \frac{\sum_{i=0}^{p-1} r_i + \sum_{i=1}^{\log_2 p} \frac{p}{2^i} - 1}{\sum_{i=0}^{p-1} r_i + \sum_{i=1}^{\log_2 p} \frac{p}{2^i}}$$

# Strategia II - Esempio

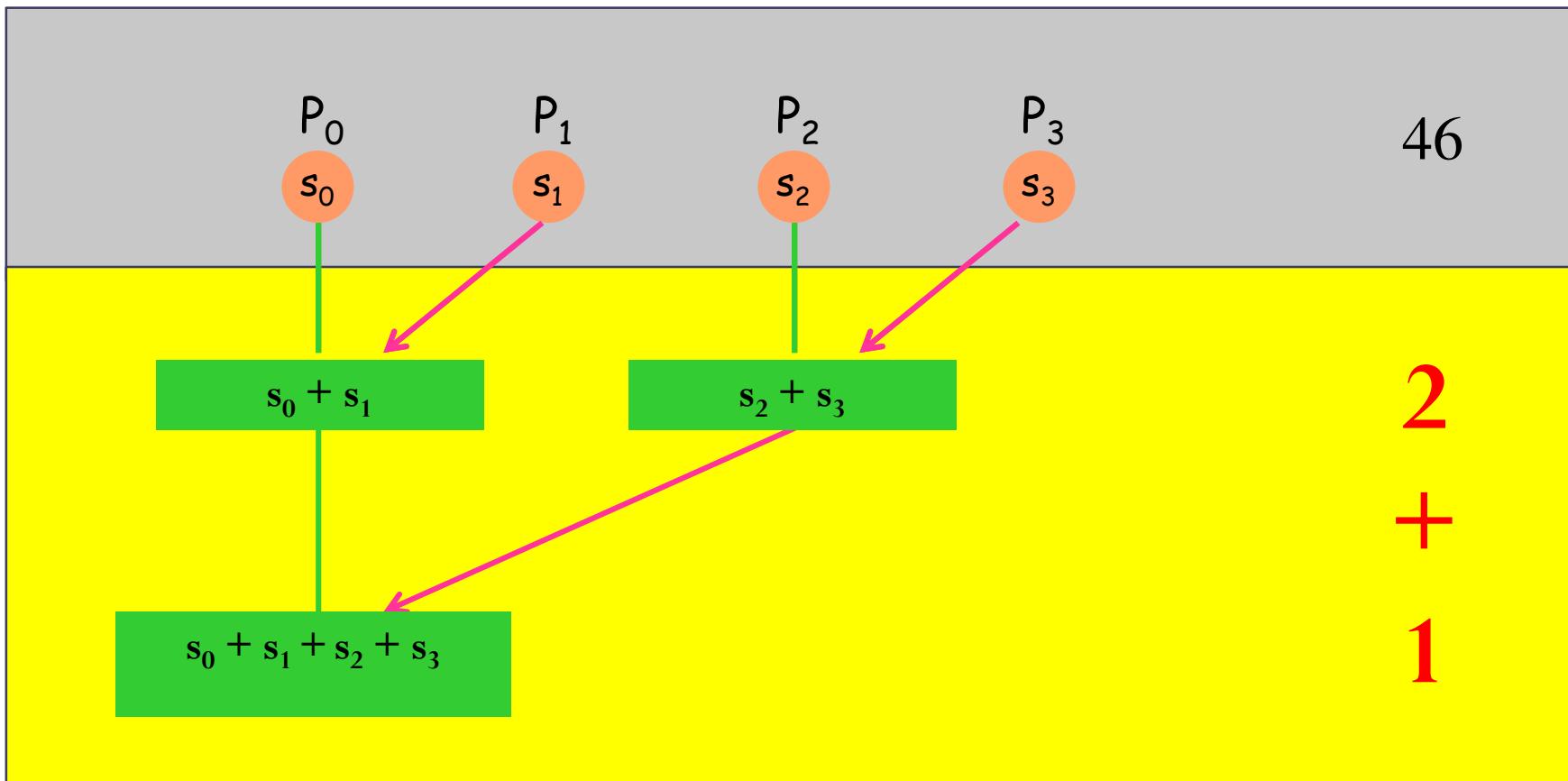
Dati : **50** numeri,  $4 = 2^2$  processi

Per le somme parziali il processore i esegue un numero di operazioni pari a:

$$r_i = \begin{cases} \left(\frac{50}{4}\right) - 1 = 11 & se \quad i \geq n \% p = 2 \\ \left(\frac{50}{4} + 1\right) - 1 = 12 & se \quad i < n \% p = 2 \end{cases}$$

Per un totale di  $12 + 12 + 11 + 11 = 46$  *operazioni*

# Strategia II - Quanti calcoli?



# Strategia II - Quanti calcoli?

Dati : **50** numeri, 4 processi

In totale allora vengono eseguite

49 *operazioni*

- Di queste:
  - 48 vengono eseguite in parallelo, cioè per la loro esecuzione lavorano contemporaneamente più di un processore
  - 1 viene eseguita invece da un solo processore

# Strategia II – Legge di Ware-Amdahl

Per la legge di Ware-Amdahl

$$S(p) = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

$\alpha$  parte sequenziale, ovvero frazione di operazioni eseguite da un solo processore

$1-\alpha$  parte parallela, ovvero frazione di operazioni eseguite contemporaneamente da più di un processore

- Nel nostro caso:

$$\alpha = \frac{1}{49} \qquad \qquad 1 - \alpha = \frac{48}{49}$$

# Strategia II – Legge di Ware-Amdahl

Per la legge di Ware-Amdahl

$$S(p) = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

$\alpha$  parte sequenziale, ovvero frazione di operazioni eseguite da un solo processore

$1-\alpha$  parte parallela, ovvero frazione di operazioni eseguite contemporaneamente da più di un processore

- Nel nostro caso:

$$\alpha = \frac{1}{49} = 0,02$$

$$1 - \alpha = \frac{48}{49} = 0,97$$

# Strategia II – Legge di Ware-Amdahl

Per la legge di Ware-Amdahl

$$S(4) = \frac{1}{0,02 + \frac{0,97}{4}}$$

$\alpha$  parte sequenziale, ovvero frazione di operazioni eseguite da un solo processore

$1-\alpha$  parte parallela, ovvero frazione di operazioni eseguite contemporaneamente da più di un processore

- Nel nostro caso:

$$\alpha = \frac{1}{49} = 0,02$$

$$1 - \alpha = \frac{48}{49} = 0,97$$

# Strategia I – Legge di Ware-Amdahl generalizzata

Per la legge di Ware-Amdahl generalizzata

$$S(p) = \frac{1}{\alpha_1 + \sum_{k=2}^p \frac{\alpha_k}{k}}$$

$\alpha_k$  frazione di operazioni eseguite da k processori

- Nel nostro caso:

$$\alpha_1 = \frac{p-1}{\sum_{i=0}^{p-1} r_i + (p-1)}$$

$$\alpha_p = \frac{\sum_{i=0}^{p-1} r_i}{\sum_{i=0}^{p-1} r_i + (p-1)}$$

Tutti gli altri sono nulli!

# Strategia I – Legge di Ware-Amdahl generalizzata

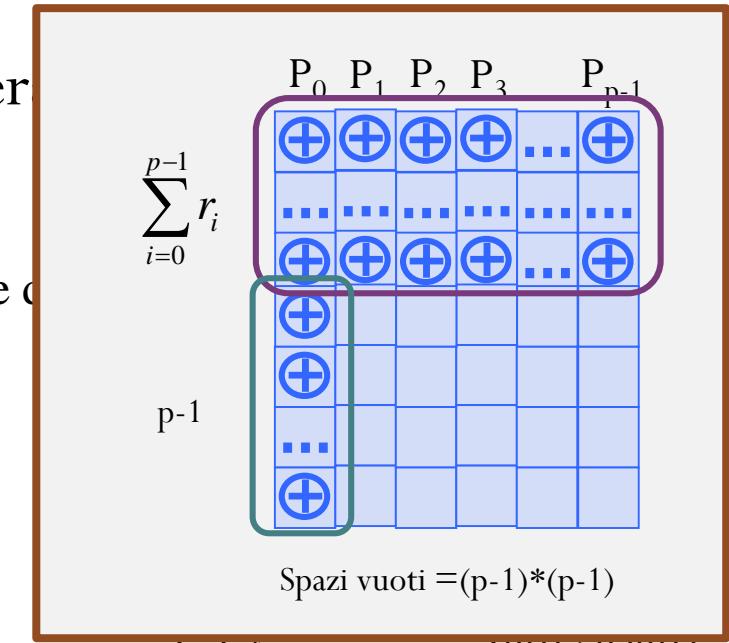
Per la legge di Ware-Amdahl gener

$$S(p) = \frac{1}{\alpha_1 + \sum_{k=2}^p \frac{\alpha_k}{k}}$$

$\alpha_k$  frazione di  
processori

- Nel nostro caso:

$$\alpha_1 = \frac{p-1}{\sum_{i=0}^{p-1} r_i + (p-1)}$$



$$\alpha_p = \frac{\sum_{i=0}^{p-1} r_i}{\sum_{i=0}^{p-1} r_i + (p-1)}$$

# Strategia I – Legge di Ware-Amdahl generalizzata

Per la legge di Ware-Amdahl generalizzata

$$S(p) = \frac{1}{\alpha_1 + \sum_{k=2}^p \frac{\alpha_k}{k}}$$

$\alpha_k$  frazione di operazioni eseguite da k processori

- Nel nostro caso:

$$\alpha_1 = \frac{3}{49} = 0,06$$

$$\alpha_p = \frac{46}{49} = 0,93$$

Tutti gli altri sono nulli!

# Strategia I – Legge di Ware-Amdahl generalizzata

Per la legge di Ware-Amdahl generalizzata

$$S(p) = \frac{1}{0,06 + \frac{0}{2} + \frac{0}{3} + \frac{0,93}{4}}$$

$\alpha_k$  frazione di operazioni eseguite da k processori

- Nel nostro caso:

$$\alpha_1 = \frac{3}{49} = 0,06$$

$$\alpha_p = \frac{46}{49} = 0,93$$

Tutti gli altri sono nulli!

# Strategia I – Legge di Ware-Amdahl generalizzata

Per la legge di Ware-Amdahl generalizzata

$$S(p) = \frac{1}{0,06 + \frac{0,93}{4}}$$

$\alpha_k$  frazione di operazioni eseguite da k processori

- Nel nostro caso:

$$\alpha_1 = \frac{3}{49} = 0,06$$

$$\alpha_p = \frac{46}{49} = 0,93$$

Tutti gli altri sono nulli!

# Strategia I – Legge di Ware-Amdahl generalizzata

Per la legge di Ware-Amdahl generalizzata

$$S(p) = \frac{1}{0,06 + \frac{0,93}{4}}$$

$\alpha_k$  frazione di operazioni eseguite da k processori

- Nel nostro caso:

$$\alpha_1 = \frac{3}{49} = 0,06$$

$$\alpha_p = \frac{46}{49} = 0,93$$

Tutti gli altri sono nulli!

**In questo caso il risultato corrisponde a quello trovato con la legge precedente**

# Strategia II – Legge di Ware-Amdahl generalizzata

Per la legge di Ware-Amdahl generalizzata

$$S(p) = \frac{1}{\alpha_1 + \sum_{k=2}^p \frac{\alpha_k}{k}}$$

$\alpha_k$  frazione di operazioni eseguite da k processori

Tutti gli altri sono nulli!

- Nel nostro caso:

$$\alpha_1 = \frac{1}{\sum_{i=0}^{p-1} r_i + \sum_{i=1}^{\log_2 p} \frac{p}{2^i}}$$

$$\alpha_{2^i} = \frac{\frac{p}{2^i}}{\sum_{i=0}^{p-1} r_i + \sum_{i=1}^{\log_2 p} \frac{p}{2^i}} \quad \forall i \in [1, (\log_2 p) - 1]$$

$$\alpha_p = \frac{\sum_{i=0}^{p-1} r_i}{\sum_{i=0}^{p-1} r_i + \sum_{i=1}^{\log_2 p} \frac{p}{2^i}}$$

# Strategia II – Legge di Ware-Amdahl generalizzata

Per la legge di Ware-Amdahl generalizzata:

$$S(p) = \frac{1}{\alpha_1 + \sum_{k=2}^p \frac{\alpha_k}{k}}$$

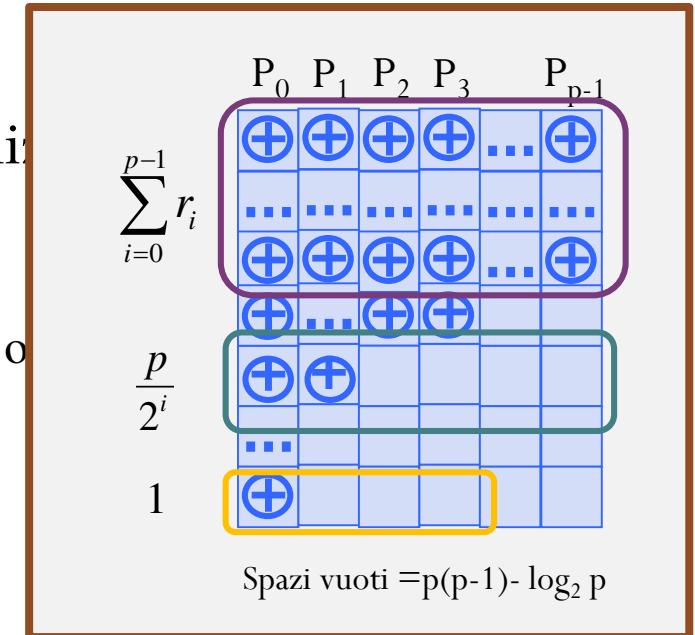
$\alpha_k$  frazione di operazioni eseguite da un singolo processore

- Nel nostro caso:

$$\alpha_1 = \frac{1}{\sum_{i=0}^{p-1} r_i + \sum_{i=1}^{\log_2 p} \frac{p}{2^i}}$$

$$\alpha_{2^i} = \frac{\frac{p}{2^i}}{\sum_{i=0}^{p-1} r_i + \sum_{i=1}^{\log_2 p} \frac{p}{2^i}} \quad \forall i \in [1, (\log_2 p) - 1]$$

$$\alpha_p = \frac{\sum_{i=0}^p r_i}{\sum_{i=0}^{p-1} r_i + \sum_{i=1}^{\log_2 p} \frac{p}{2^i}}$$



# Strategia II – Legge di Ware-Amdahl generalizzata

Per la legge di Ware-Amdahl generalizzata:

$$S(p) = \frac{1}{\alpha_1 + \sum_{k=2}^p \frac{\alpha_k}{k}}$$

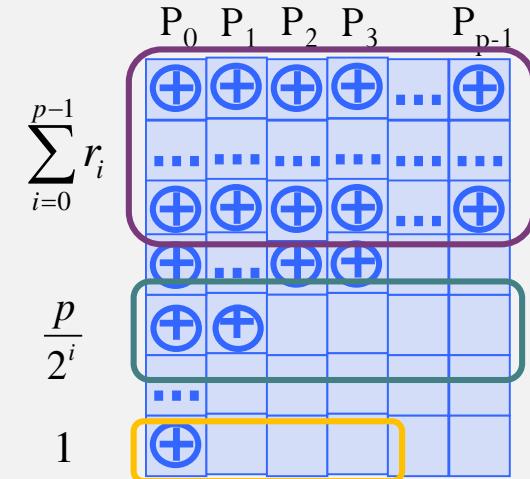
$\alpha_k$  frazione di operazioni eseguite da  $k$  processori

- Nel nostro caso:

$$\alpha_1 = \frac{1}{\sum_{i=0}^{p-1} r_i + \sum_{i=1}^{\log_2 p} \frac{p}{2^i}}$$

$$\alpha_{2^i} = \frac{\frac{p}{2^i}}{\sum_{i=0}^{p-1} r_i + \sum_{i=1}^{\log_2 p} \frac{p}{2^i}} \quad \forall i \in [1, (\log_2 p) - 1]$$

$$\alpha_p = \frac{\sum_{i=0}^p r_i}{\sum_{i=0}^{p-1} r_i + \sum_{i=1}^{\log_2 p} \frac{p}{2^i}}$$



Consideriamo le operazioni separatamente in base al numero di processori che le esegue

# Strategia II – Legge di Ware-Amdahl generalizzata

Per la legge di Ware-Amdahl generalizzata

$$S(p) = \frac{1}{\alpha_1 + \sum_{k=2}^p \frac{\alpha_k}{k}}$$

$\alpha_k$  frazione di operazioni eseguite da k processori

Tutti gli altri sono nulli!

- Nel nostro caso:

$$\alpha_1 = \frac{1}{49} = 0,02$$

$$\alpha_2 = \frac{2}{49} = 0,04$$

$$\alpha_4 = \frac{46}{49} = 0,93$$

# Strategia II – Legge di Ware-Amdahl generalizzata

Per la legge di Ware-Amdahl generalizzata

$$S(p) = \frac{1}{0,02 + \frac{0,04}{2} + \frac{0}{3} + \frac{0,93}{4}}$$

$\alpha_k$  frazione di operazioni eseguite da k processori

Tutti gli altri sono nulli!

- Nel nostro caso:

$$\alpha_1 = \frac{1}{49} = 0,02$$

$$\alpha_2 = \frac{2}{49} = 0,04$$

$$\alpha_4 = \frac{46}{49} = 0,93$$

# Strategia II – Legge di Ware-Amdahl generalizzata

Per la legge di Ware-Amdahl generalizzata

$$S(p) = \frac{1}{0,02 + \frac{0,04}{2} + \frac{0,93}{4}}$$

$\alpha_k$  frazione di operazioni eseguite da k processori

- Nel nostro caso:

Tutti gli altri sono nulli!

$$\alpha_1 = \frac{1}{49} = 0,02$$

$$\alpha_2 = \frac{2}{49} = 0,04$$

$$\alpha_4 = \frac{46}{49} = 0,93$$

# Strategia II – Legge di Ware-Amdahl generalizzata

Per la legge di Ware-Amdahl generalizzata

$$S(p) = \frac{1}{0,02 + \frac{0,04}{2} + \frac{0,93}{4}}$$

$\alpha_k$  frazione di operazioni eseguite da k processori

Tutti gli altri sono nulli!

- Nel nostro caso:

$$\alpha_1 = \frac{1}{49} = 0,02$$

$$\alpha_2 = \frac{2}{49} = 0,04$$

$$\alpha_4 = \frac{46}{49} = 0,93$$

In questo caso il risultato NON corrisponde a quello trovato con la legge precedente

# Calcolo Speed Up ed Efficienza

Prodotto Matrice-Vettore

# Strategia I - Quanti passi di calcolo?

$$Ax = b \quad A \in \mathbb{R}^{n \times m} \quad x \in \mathbb{R}^m \quad b \in \mathbb{R}^n$$

- Distribuzione della matrice per righe: ad ogni processore va un insieme di righe intere
- Il vettore X viene dato a tutti i processori
- Ogni processore calcola un prodotto matrice-vettore più piccolo

$$A_i x = b_i \quad A_i \in \mathbb{R}^{r \times m} \quad x \in \mathbb{R}^m \quad b_i \in \mathbb{R}^r$$

- Non si deve ricompilare il risultato

# Strategia I - Quanti passi di calcolo?

Dati : p processi, n ( $\geq p$ )

$$r = \left\lceil \frac{n}{p} \right\rceil = \begin{cases} \frac{n}{p} & se \quad n \% p = 0 \\ \frac{n}{p} + 1 & se \quad n \% p \neq 0 \end{cases}$$

Algoritmo sequenziale

$$T(1) = n \cdot (2m - 1)t_{calc}$$

$$T(p) = r \cdot (2m - 1)t_{calc}$$



Solo più piccolo!

```
for i=0,n-1 do
    yi=0
    for j=0,n-1 do
        yi=yi+aij xj
    endfor
endfor
```

# Strategia I – Speed up?

$$S(p) = \frac{T(1)}{T(p)}$$

$$T(1) = n \cdot (2m - 1)t_{calc}$$

$$T(p) = r \cdot (2m - 1)t_{calc}$$

# Strategia I – Speed up?

$$S(p) = \frac{T(1)}{T(p)} = \frac{n \cdot (2m-1)t_{calc}}{r \cdot (2m-1)t_{calc}}$$

$$T(1) = n \cdot (2m-1)t_{calc}$$

$$T(p) = r \cdot (2m-1)t_{calc}$$

# Strategia I – Speed up?

$$S(p) = \frac{T(1)}{T(p)} = \frac{n}{r}$$

$$T(1) = n \cdot (2m - 1)t_{calc}$$

$$T(p) = r \cdot (2m - 1)t_{calc}$$

# Strategia I – Quanti passi di comun.?



# Strategia I – Quanti passi di comun.?

Dati : p processi, n ( $\geq p$ )

$$r = \left\lceil \frac{n}{p} \right\rceil = \begin{cases} \frac{n}{p} & \text{se } n \% p = 0 \\ \frac{n}{p} + 1 & \text{se } n \% p \neq 0 \end{cases}$$



Se li vogliamo considerare per rimettere insieme il risultato

$$T(1) = n \cdot (2m - 1)t_{calc}$$

$$T(p) = r \cdot (2m - 1)t_{calc}$$

# Strategia I – Quanti passi di comun.?

Dati : p processi, n ( $\geq p$ )

$$r = \left\lceil \frac{n}{p} \right\rceil = \begin{cases} \frac{n}{p} & \text{se } n \% p = 0 \\ \frac{n}{p} + 1 & \text{se } n \% p \neq 0 \end{cases}$$



Se li vogliamo considerare per rimettere insieme il risultato

$$T(1) = n \cdot (2m - 1)t_{calc}$$

$$T(p) = r \cdot (2m - 1)t_{calc} + r \cdot \log_2 p t_{com}$$

# Strategia I – Quanti passi di comun.?

Dati : p processi, n ( $\geq p$ )

$$r = \left\lceil \frac{n}{p} \right\rceil = \begin{cases} \frac{n}{p} & \text{se } n \% p = 0 \\ \frac{n}{p} + 1 & \text{se } n \% p \neq 0 \end{cases}$$



Se li vogliamo considerare per rimettere insieme il risultato

$$T(1) = n \cdot (2m - 1)t_{calc}$$

$t_{com} = ht_{calc}$

$$T(p) = r \cdot (2m - 1)t_{calc} + hr \log_2 pt_{calc}$$

# Strategia I – Speed up?

$$S(p) = \frac{T(1)}{T(p)} = \frac{n \cdot (2m-1)t_{calc}}{[r \cdot (2m-1) + hr \log_2 p]t_{calc}}$$

$$T(1) = n \cdot (2m-1)t_{calc}$$

$$t_{com} = ht_{calc}$$

$$T(p) = r \cdot (2m-1)t_{calc} + hr \log_2 pt_{calc}$$

# Strategia I – Speed up?

$$S(p) = \frac{T(1)}{T(p)} = \frac{n \cdot (2m - 1)}{r \cdot (2m - 1) + hr \log_2 p}$$

$$T(1) = n \cdot (2m - 1)t_{calc}$$

$$t_{com} = ht_{calc}$$

$$T(p) = r \cdot (2m - 1)t_{calc} + hr \log_2 pt_{calc}$$

# Strategia II - Quanti passi di calcolo?

$$Ax = b \quad A \in \Re^{n \times m} \quad x \in \Re^m \quad b \in \Re^n$$

- Distribuzione della matrice per colonne: ad ogni processore va un insieme di colonne intere
- Il vettore X viene distribuito tra tutti i processori
- Ogni processore calcola un prodotto matrice-vettore più piccolo

$$A_i x_i = s_i \quad A_i \in \Re^{n \times c} \quad x_i \in \Re^c \quad s_i \in \Re^n$$

- Si deve ricompilare il risultato

$$b = \sum_i s_i$$

# Strategia II - Quanti passi di calcolo?

Dati : q processi,  $m (\geq q)$

$$c = \left\lceil \frac{m}{q} \right\rceil = \begin{cases} \frac{m}{q} & se \quad m \% q = 0 \\ \frac{m}{q} + 1 & se \quad m \% q \neq 0 \end{cases}$$

Algoritmo a blocchi

$$T(1) = n \cdot (2m - 1)t_{calc}$$

$$T(q) = n \cdot (2c - 1)t_{calc} + n \cdot \log_2 q \cdot t_{calc}$$

Prodotto più piccolo!

```
begin
    y=0
    for i=0 to p-1 do
        ri=Aixi
        y=y+ri
    endfor
end
```

# Strategia II - Quanti passi di calcolo?

Dati : q processi,  $\mathbf{m} (\geq q)$

$$c = \left\lceil \frac{m}{q} \right\rceil = \begin{cases} \frac{m}{q} & se \quad m \% q = 0 \\ \frac{m}{q} + 1 & se \quad m \% q \neq 0 \end{cases}$$

Algoritmo a blocchi

$$T(1) = n \cdot (2m - 1)t_{calc}$$

$$T(q) = n \cdot (2c - 1)t_{calc} + n \cdot \log_2 q \cdot t_{calc}$$

N somme per  $\log_2(q)$  volte

```
begin
    y=0
    for i=0 to p-1 do
        ri=Aixi
        y=y+ri
    endfor
end
```

# Strategia II – Speed up?

$$S(q) = \frac{T(1)}{T(q)} = \frac{n \cdot (2m-1)t_{calc}}{n \cdot (2c-1)t_{calc} + n \cdot \log_2 q \cdot t_{calc}}$$

$$T(1) = n \cdot (2m-1)t_{calc}$$

$$T(q) = n \cdot (2c-1)t_{calc} + n \cdot \log_2 q \cdot t_{calc}$$

# Strategia II – Speed up?

$$S(q) = \frac{T(1)}{T(q)} = \frac{(2m-1)}{(2c-1) + \log_2 q}$$

$$T(1) = n \cdot (2m-1)t_{calc}$$

$$T(q) = n \cdot (2c-1)t_{calc} + n \cdot \log_2 q \cdot t_{calc}$$

# Strategia II – Quanti passi di comun.?

Dati : q processi,  $\mathbf{m} (\geq \mathbf{q})$

$$c = \left\lceil \frac{m}{q} \right\rceil = \begin{cases} \frac{m}{q} & se \quad m \% q = 0 \\ \frac{m}{q} + 1 & se \quad m \% q \neq 0 \end{cases}$$

$$T(1) = n \cdot (2m - 1)t_{calc}$$

$$T(q) = n \cdot (2c - 1)t_{calc} + n \cdot \log_2 q \cdot t_{calc}$$

# Strategia II – Quanti passi di comun.?

Dati : q processi,  $\mathbf{m} (\geq \mathbf{q})$

$$c = \left\lceil \frac{m}{q} \right\rceil = \begin{cases} \frac{m}{q} & se \quad m \% q = 0 \\ \frac{m}{q} + 1 & se \quad m \% q \neq 0 \end{cases}$$

$$T(1) = n \cdot (2m - 1)t_{calc}$$

$$T(q) = n \cdot (2c - 1)t_{calc} + n \cdot \log_2 q \cdot t_{calc} + n \cdot \log_2 q \cdot t_{com}$$

# Strategia II – Quanti passi di comun.?

Dati : q processi,  $\mathbf{m} (\geq q)$

$$c = \left\lceil \frac{m}{q} \right\rceil = \begin{cases} \frac{m}{q} & se \quad m \% q = 0 \\ \frac{m}{q} + 1 & se \quad m \% q \neq 0 \end{cases}$$

$$T(1) = n \cdot (2m - 1)t_{calc}$$

$$T(q) = n \cdot (2c - 1)t_{calc} + n \cdot \log_2 q \cdot t_{calc} + h \cdot n \cdot \log_2 q \cdot t_{calc}$$

$$t_{com} = ht_{calc}$$

# Strategia II – Speed up?

$$S(q) = \frac{T(1)}{T(q)} = \frac{n \cdot (2m-1)t_{calc}}{[n \cdot (2c-1) + n \cdot \log_2 q + h \cdot n \cdot \log_2 q]t_{calc}}$$

$$T(1) = n \cdot (2m-1)t_{calc}$$

$$T(q) = n \cdot (2c-1)t_{calc} + n \cdot \log_2 q \cdot t_{calc} + h \cdot n \cdot \log_2 q \cdot t_{calc}$$

# Strategia II – Speed up?

$$S(q) = \frac{T(1)}{T(q)} = \frac{n \cdot (2m - 1)}{n \cdot (2c - 1) + n \cdot \log_2 q + h \cdot n \cdot \log_2 q}$$

$$T(1) = n \cdot (2m - 1)t_{calc}$$

$$T(q) = n \cdot (2c - 1)t_{calc} + n \cdot \log_2 q \cdot t_{calc} + h \cdot n \cdot \log_2 q \cdot t_{calc}$$

# Strategia III - Quanti passi di calcolo?

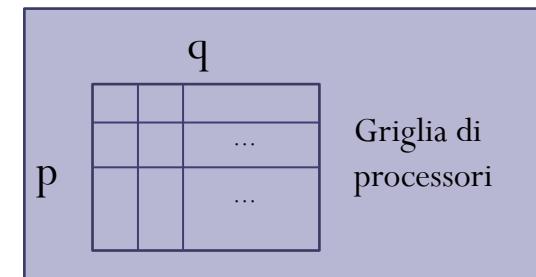
$$Ax = b \quad A \in \mathbb{R}^{n \times m} \quad x \in \mathbb{R}^m \quad b \in \mathbb{R}^n$$

- Processori idealmente disposti in una griglia pxq
- Distribuzione della matrice per blocchi: ad ogni processore va un blocco rettangolare
- Il vettore X viene distribuito tra i processori sulla stessa colonna della griglia
- Ogni processore calcola un prodotto matrice-vettore più piccolo  
$$A_i x_i = s_i \quad A_i \in \mathbb{R}^{r \times c} \quad x_i \in \mathbb{R}^c \quad s_i \in \mathbb{R}^r$$
- Si deve ricompilare il risultato solo lungo le righe della griglia

# Strategia III - Quanti passi di calcolo?

Dati : pxq processi,  $n (\geq p)$ ,  $m (\geq q)$

$$r = \left\lceil \frac{n}{p} \right\rceil = \begin{cases} \frac{n}{p} & se \quad n \% p = 0 \\ \frac{n}{p} + 1 & se \quad n \% p \neq 0 \end{cases}$$
$$c = \left\lceil \frac{m}{q} \right\rceil = \begin{cases} \frac{m}{q} & se \quad m \% q = 0 \\ \frac{m}{q} + 1 & se \quad m \% q \neq 0 \end{cases}$$



$$T(1) = n \cdot (2m - 1)t_{calc}$$

Albero per la somma dei vettori di  $r$  elementi tra i  $q$  processori di una riga della griglia

$$T(pxq) = r \cdot (2c - 1)t_{calc} + (r \cdot \log_2 q)t_{calc}$$

# Strategia III – Speed up?

$$S(pxq) = \frac{T(1)}{T(pxq)} = \frac{n \cdot (2m-1)t_{calc}}{r \cdot (2c-1)t_{calc} + (r \cdot \log q)t_{calc}}$$

$$T(1) = n \cdot (2m-1)t_{calc}$$

$$T(pxq) = r \cdot (2c-1)t_{calc} + (r \cdot \log_2 q)t_{calc}$$

# Strategia III – Speed up?

$$S(pxq) = \frac{T(1)}{T(pxq)} = \frac{n \cdot (2m - 1)}{r \cdot (2c - 1) + r \cdot \log q}$$

$$T(1) = n \cdot (2m - 1)t_{calc}$$

$$T(pxq) = r \cdot (2c - 1)t_{calc} + (r \cdot \log_2 q)t_{calc}$$

# Strategia III – Quanti passi di comun.?

Dati : pxq processi,  $n (\geq p)$ ,  $m (\geq q)$

$$r = \left\lceil \frac{n}{p} \right\rceil = \begin{cases} \frac{n}{p} & se \quad n \% p = 0 \\ \frac{n}{p} + 1 & se \quad n \% p \neq 0 \end{cases}$$
$$c = \left\lceil \frac{m}{q} \right\rceil = \begin{cases} \frac{m}{q} & se \quad m \% q = 0 \\ \frac{m}{q} + 1 & se \quad m \% q \neq 0 \end{cases}$$

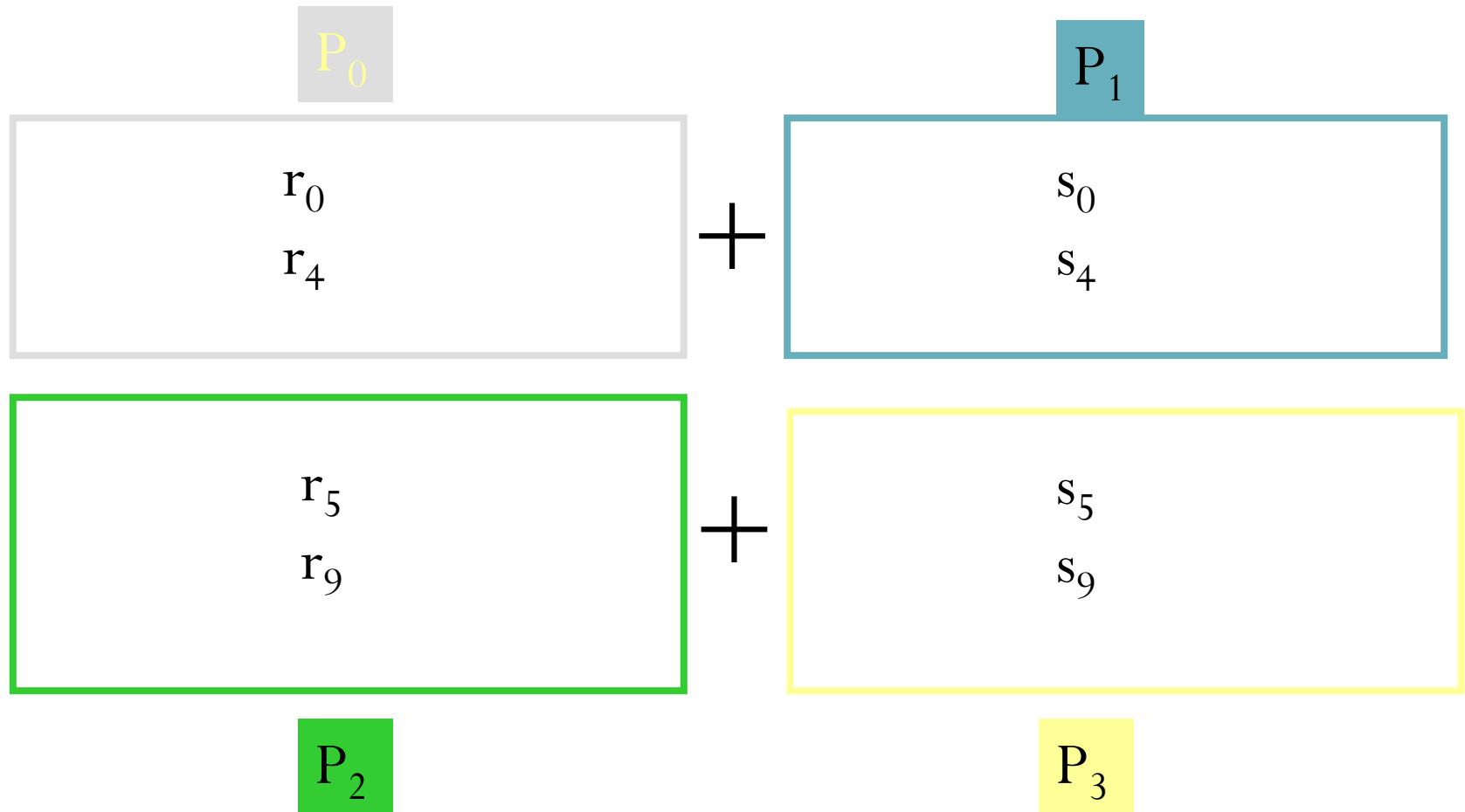
$$T(1) = n \cdot (2m - 1)t_{calc}$$

$$T(pxq) = r \cdot (2c - 1)t_{calc} + (r \cdot \log_2 q)t_{calc}$$

# Strategia III – Quanti passi di comun.?

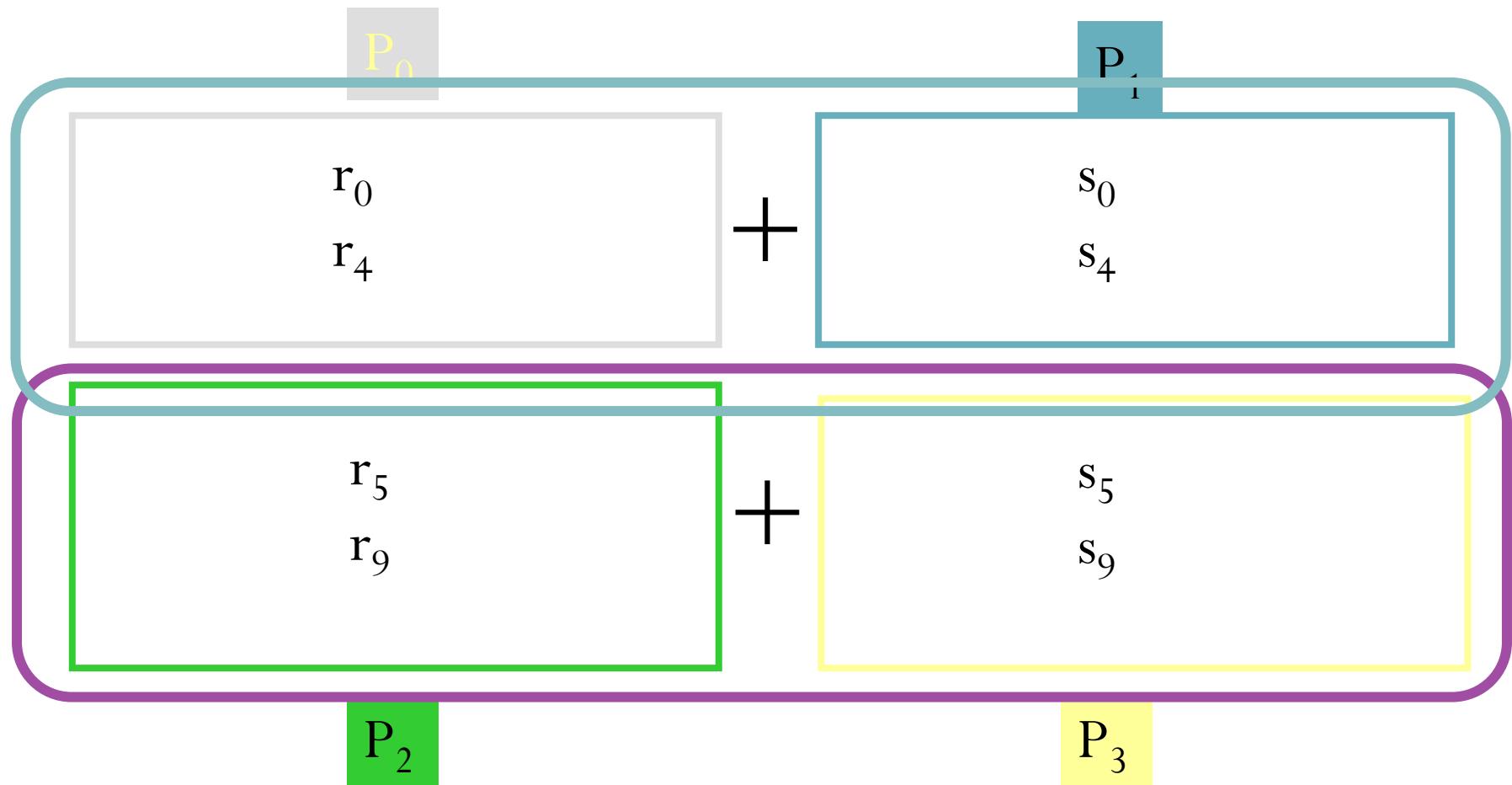
Ricordiamo chi comunica cosa a chi???

# Esempio $N = 9$ , $P=4$



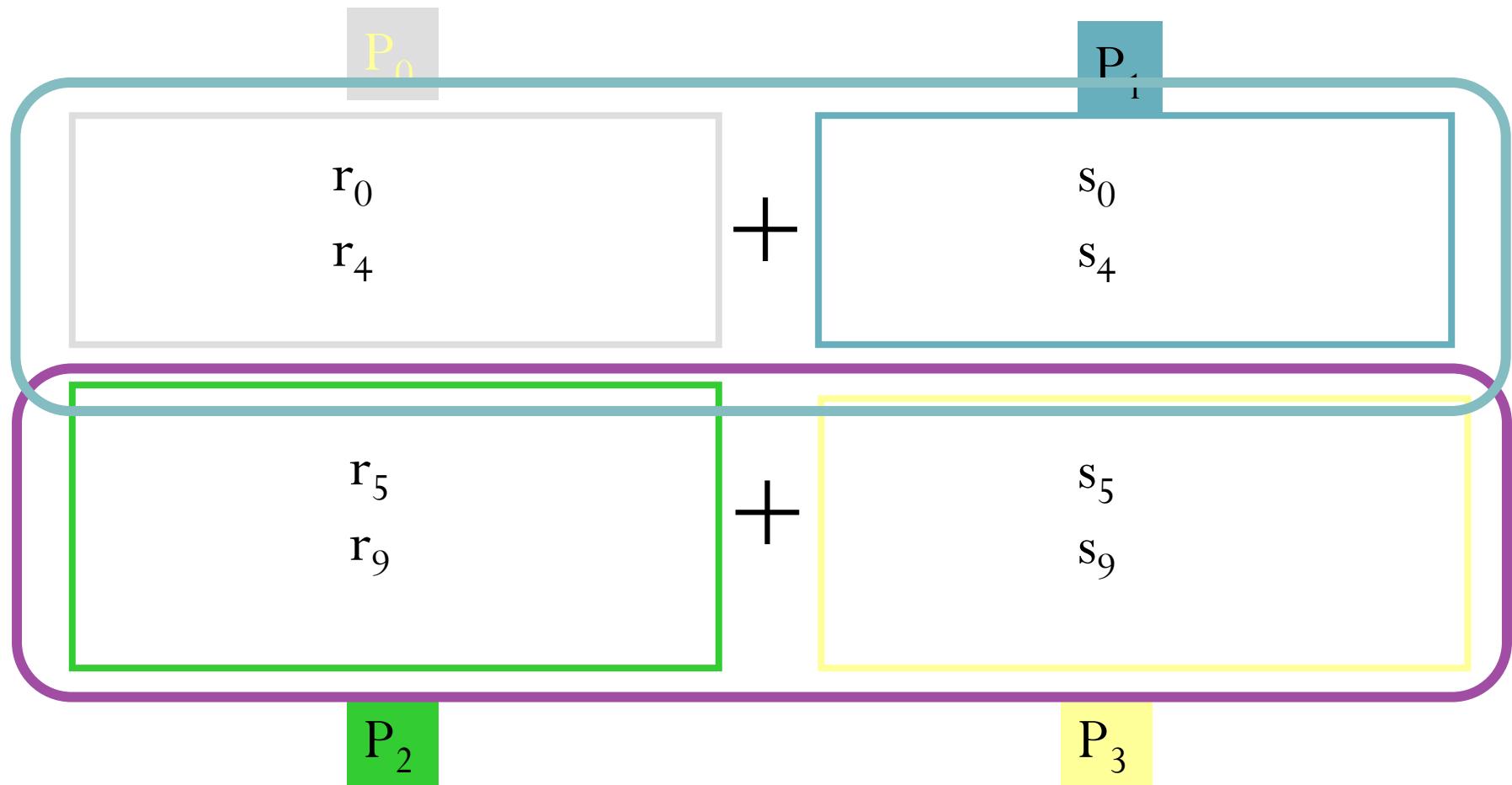
Comunicazione: somma in parallelo

# Esempio $N = 9$ , $P=4$



Comunicazione: somma in parallelo

# Esempio $N = 9$ , $P=4$



La comunicazione avviene solo riga per riga (della griglia)

# Strategia III – Quanti passi di comun.?

Dati : pxq processi,  $n (\geq p)$ ,  $m (\geq q)$

$$r = \left\lceil \frac{n}{p} \right\rceil = \begin{cases} \frac{n}{p} & se \quad n \% p = 0 \\ \frac{n}{p} + 1 & se \quad n \% p \neq 0 \end{cases}$$
$$c = \left\lceil \frac{m}{q} \right\rceil = \begin{cases} \frac{m}{q} & se \quad m \% q = 0 \\ \frac{m}{q} + 1 & se \quad m \% q \neq 0 \end{cases}$$

Albero per la somma di vettori di r elementi tra processori  
sulla stessa riga della griglia

$$T(1) = n \cdot (2m - 1)t_{calc}$$

$$T(pxq) = r \cdot (2c - 1)t_{calc} + (r \cdot \log_2 q)t_{calc} + (r \cdot \log_2 q)t_{com}$$

# Strategia III – Quanti passi di comun.?

Dati : pxq processi,  $n (\geq p)$ ,  $m (\geq q)$

$$r = \left\lceil \frac{n}{p} \right\rceil = \begin{cases} \frac{n}{p} & \text{se } n \% p = 0 \\ \frac{n}{p} + 1 & \text{se } n \% p \neq 0 \end{cases}$$

$$c = \left\lceil \frac{m}{q} \right\rceil = \begin{cases} \frac{m}{q} & \text{se } m \% q = 0 \\ \frac{m}{q} + 1 & \text{se } m \% q \neq 0 \end{cases}$$

Se li vogliamo considerare per rimettere insieme il risultato

$$T(1) = n \cdot (2m - 1)t_{calc}$$

$$T(pxq) = r \cdot (2c - 1)t_{calc} + (r \cdot \log_2 q)t_{calc} + (r \cdot \log_2 q)t_{com}$$

# Strategia III – Quanti passi di comun.?

Dati : pxq processi,  $n (\geq p)$ ,  $m (\geq q)$

$$r = \left\lceil \frac{n}{p} \right\rceil = \begin{cases} \frac{n}{p} & \text{se } n \% p = 0 \\ \frac{n}{p} + 1 & \text{se } n \% p \neq 0 \end{cases}$$

$$c = \left\lceil \frac{m}{q} \right\rceil = \begin{cases} \frac{m}{q} & \text{se } m \% q = 0 \\ \frac{m}{q} + 1 & \text{se } m \% q \neq 0 \end{cases}$$

Se li vogliamo considerare per rimettere insieme il risultato

$$T(1) = n \cdot (2m - 1)t_{calc}$$

Albero per la riunione di vettori di r elementi in uno da n elementi, tra i processori di una colonna della griglia

$$T(pxq) = r \cdot (2c - 1)t_{calc} + (r \cdot \log_2 q)t_{calc} + (r \cdot \log_2 q)t_{com} + (r \cdot \log_2 p)t_{com}$$

# Strategia III – Quanti passi di comun.?

Dati : pxq processi,  $n (\geq p)$ ,  $m (\geq q)$

$$r = \left\lceil \frac{n}{p} \right\rceil = \begin{cases} \frac{n}{p} & se \quad n \% p = 0 \\ \frac{n}{p} + 1 & se \quad n \% p \neq 0 \end{cases}$$

$$c = \left\lceil \frac{m}{q} \right\rceil = \begin{cases} \frac{m}{q} & se \quad m \% q = 0 \\ \frac{m}{q} + 1 & se \quad m \% q \neq 0 \end{cases}$$

$$T(1) = n \cdot (2m - 1)t_{calc}$$

$$t_{com} = ht_{calc}$$

$$T(pxq) = r \cdot (2c - 1)t_{calc} + (r \cdot \log_2 q)t_{calc} + h(r \cdot \log_2 q)t_{calc} + h(r \cdot \log_2 p)t_{calc}$$

# Strategia III – Speed up?

$$S(pxq) = \frac{T(1)}{T(pxq)} = \frac{n \cdot (2m-1)t_{calc}}{r \cdot (2c-1)t_{calc} + (r \cdot \log_2 q)t_{calc} + h(r \cdot \log_2 q)t_{calc} + h(r \cdot \log_2 p)t_{calc}}$$

$$T(1) = n \cdot (2m-1)t_{calc}$$

$$T(pxq) = r \cdot (2c-1)t_{calc} + (r \cdot \log_2 q)t_{calc} + h(r \cdot \log_2 q)t_{calc} + h(r \cdot \log_2 p)t_{calc}$$

# Strategia III – Speed up?

$$S(pxq) = \frac{T(1)}{T(pxq)} = \frac{n \cdot (2m-1)}{r \cdot (2c-1) + r \cdot \log_2 q + hr \cdot \log_2 q + hr \cdot \log_2 p}$$

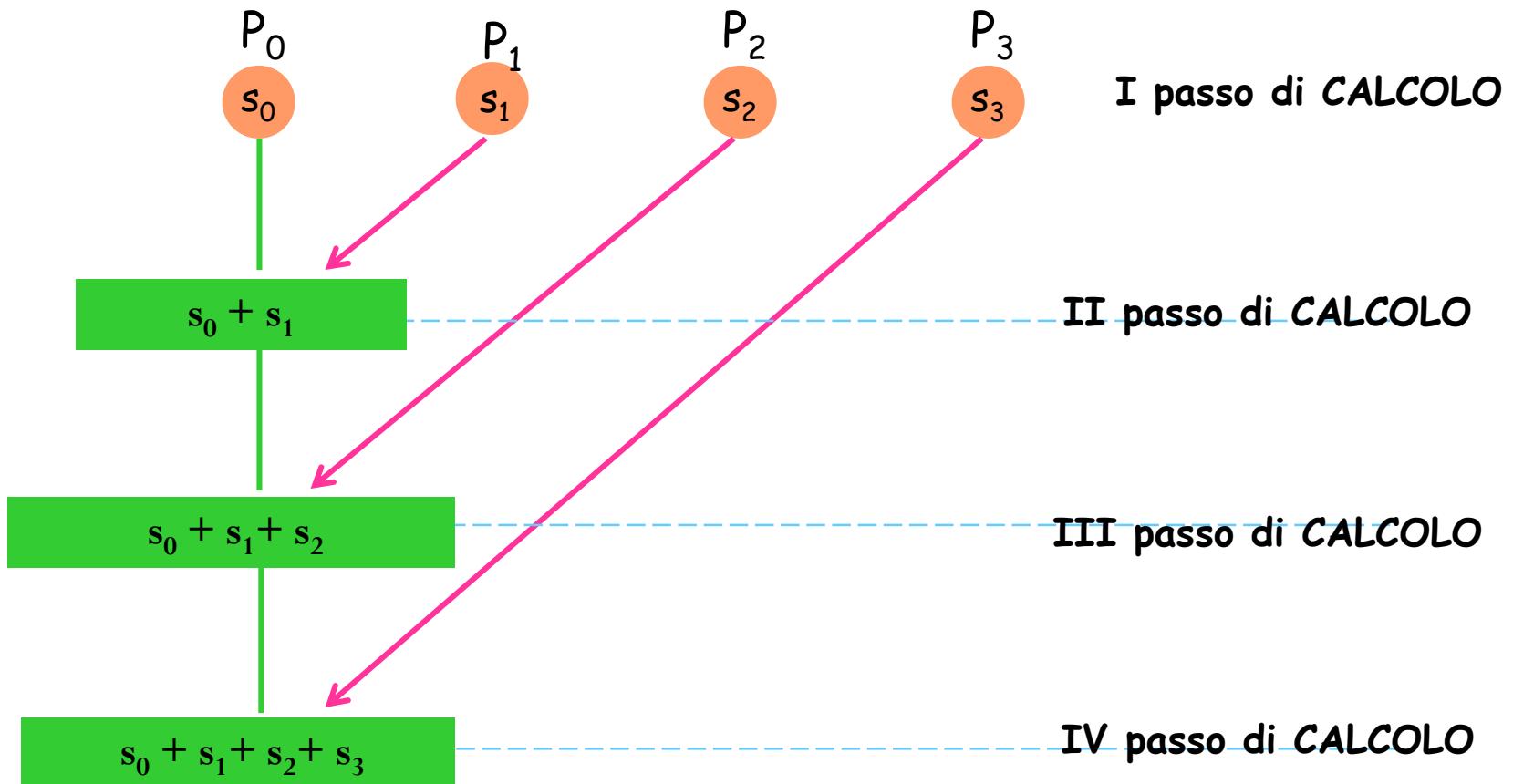
$$T(1) = n \cdot (2m-1)t_{calc}$$

$$T(pxq) = r \cdot (2c-1)t_{calc} + (r \cdot \log_2 q)t_{calc} + h(r \cdot \log_2 q)t_{calc} + h(r \cdot \log_2 p)t_{calc}$$

# Calcolo Speed Up ed Efficienza

Somma di N numeri

# Strategia I - Quanti passi di calcolo?



# Strategia I - Quanti passi di calcolo?

Dati :  $n$  ( $\geq 2$ ) numeri,  $p$  processi, con  $p \leq 2n$  e

$$r = \left\lceil \frac{n}{p} \right\rceil = \begin{cases} \frac{n}{p} & se \quad n \% p = 0 \\ \frac{n}{p} + 1 & se \quad n \% p \neq 0 \end{cases}$$

$$T(1) = (n-1)t_{calc}$$

$$T(p) = (r-1)t_{calc} + (p-1)t_{calc}$$

$t_{calc}$  = tempo di esecuzione di 1 addizione

# Strategia I - Quanti passi di calcolo?

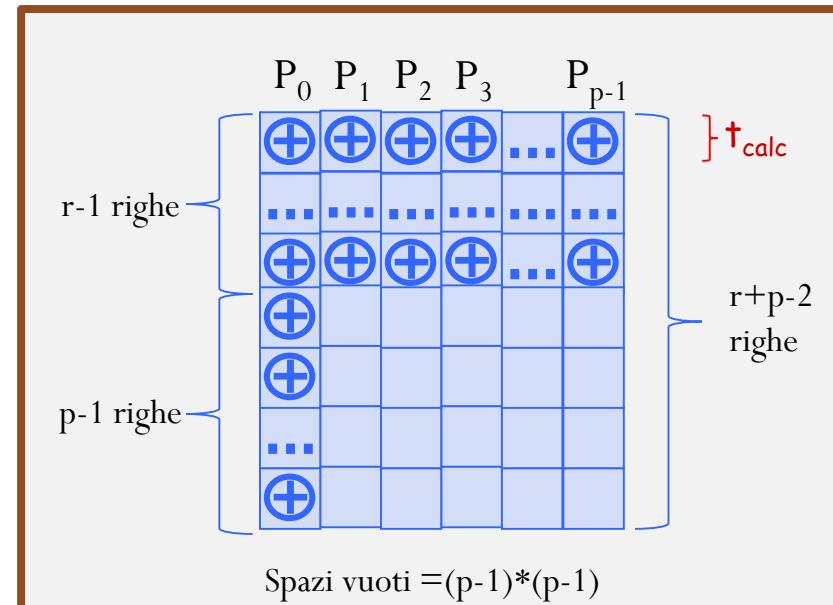
Dati :  $n (\geq 2)$  numeri,  $p$  processi, con  $p \leq 2n$  e

$$r = \left\lceil \frac{n}{p} \right\rceil = \begin{cases} \frac{n}{p} & \text{se } n \% p = 0 \\ \frac{n}{p} + 1 & \text{se } n \% p \neq 0 \end{cases}$$

$$T(1) = (n-1)t_{calc}$$

$$T(p) = (r-1)t_{calc} + (p-1)t_{calc}$$

$t_{calc}$ = tempo di esecuzione di 1 addizione

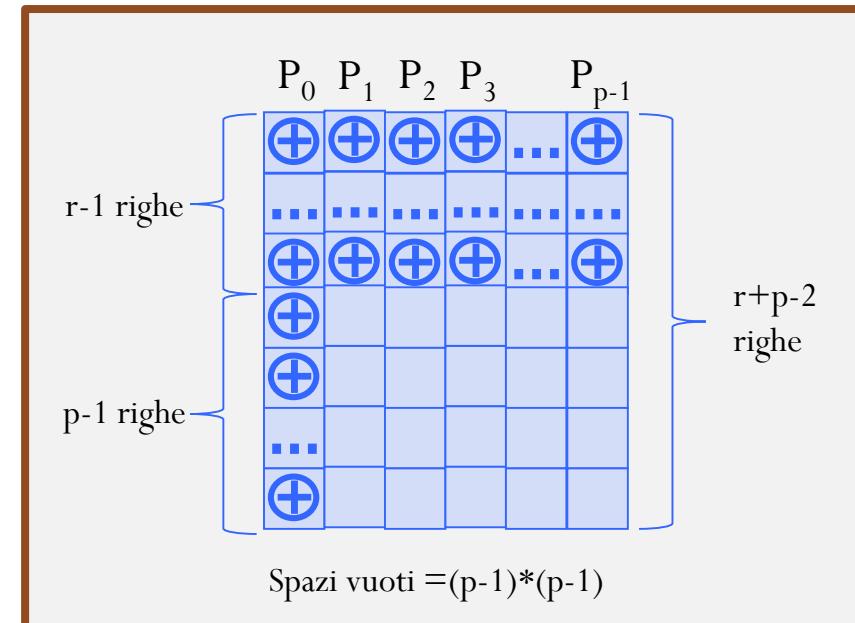


# Strategia I – Speed up?

$$S(p) = \frac{T(1)}{T(p)} = \frac{(n-1)t_{calc}}{(r+p-2)t_{calc}}$$

$$T(1) = (n-1)t_{calc}$$

$$T(p) = (r-1)t_{calc} + (p-1)t_{calc}$$

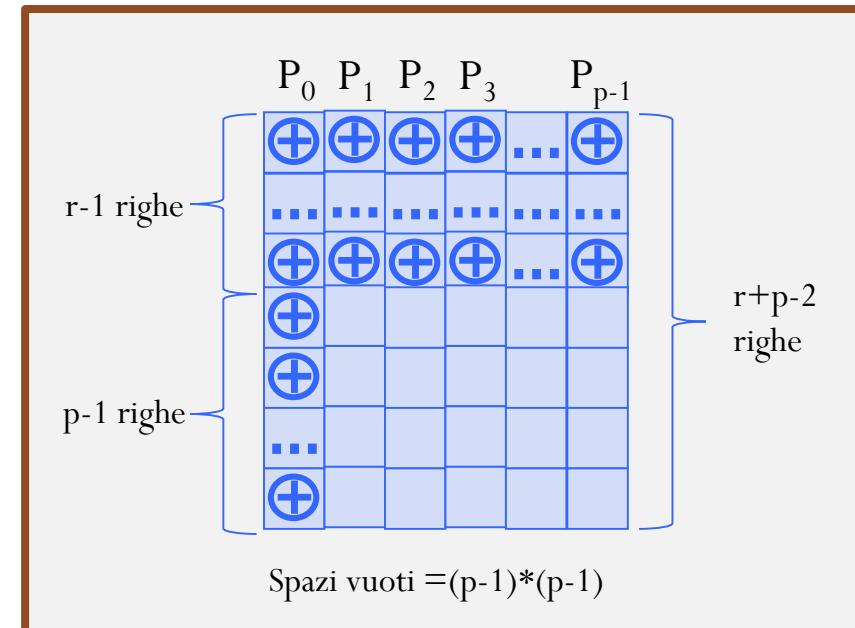


# Strategia I – Speed up?

$$S(p) = \frac{T(1)}{T(p)} = \frac{(n-1)t_{calc}}{(r+p-2)t_{calc}}$$

$$T(1) = (n-1)t_{calc}$$

$$T(p) = (r-1)t_{calc} + (p-1)t_{calc}$$

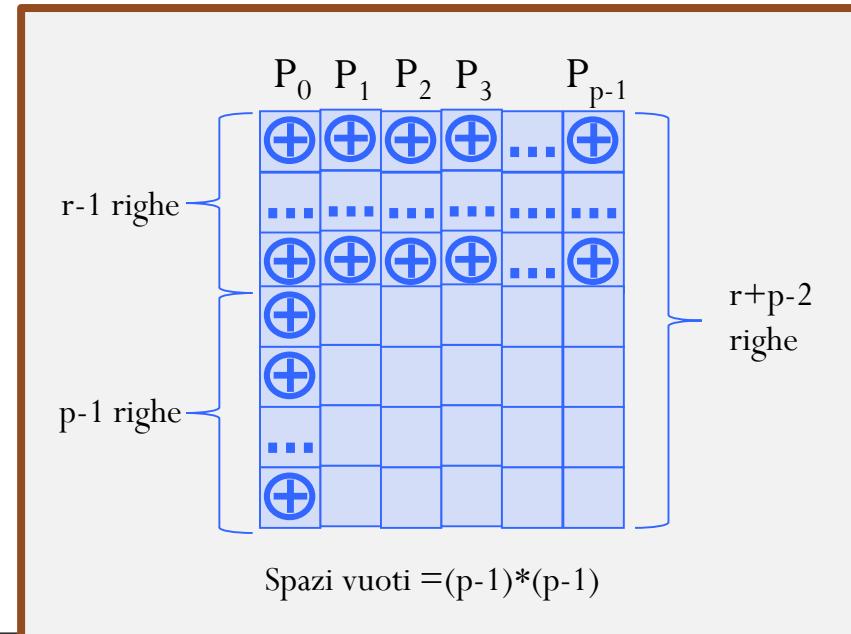


# Strategia I – Efficienza?

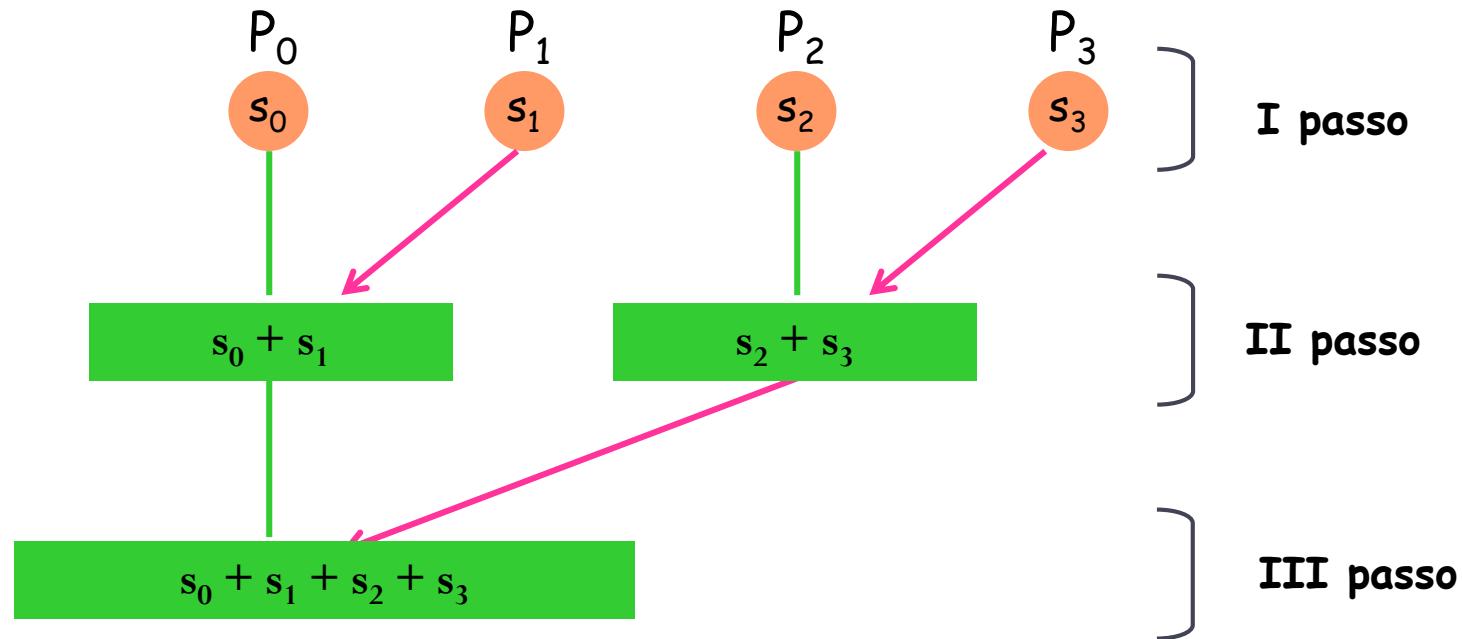
$$E(p) = \frac{S(p)}{p}$$

$$T(1) = (n-1)t_{calc}$$

$$T(p) = (r-1)t_{calc} + (p-1)t_{calc}$$



# Strategia II - Quanti passi di calcolo?



# Strategia II - Quanti passi di calcolo?

Dati :  $n$  ( $\geq 2$ ) numeri,  $p = 2^k$  processi, con  $k \leq \log_2 n$  e

$$r = \left\lceil \frac{n}{p} \right\rceil = \begin{cases} \frac{n}{p} & se \quad n \% p = 0 \\ \frac{n}{p} + 1 & se \quad n \% p \neq 0 \end{cases}$$

$$T(1) = (n - 1)t_{calc}$$

$$T(p) = (r - 1)t_{calc} + (\log_2 p)t_{calc}$$

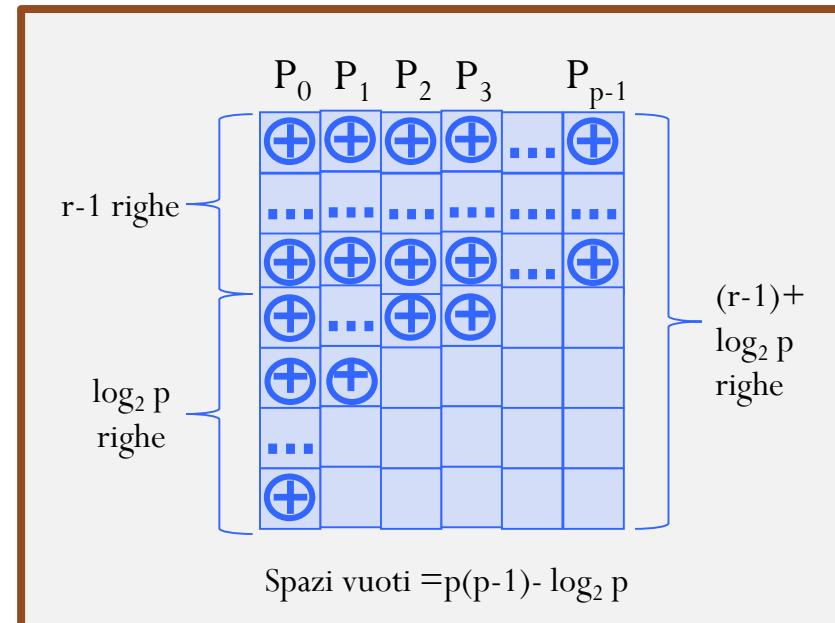
# Strategia II - Quanti passi di calcolo?

Dati :  $n (\geq 2)$  numeri,  $p = 2^k$  processi, con  $k \leq \log_2 n$  e

$$r = \left\lceil \frac{n}{p} \right\rceil = \begin{cases} \frac{n}{p} & se \quad n \% p = 0 \\ \frac{n}{p} + 1 & se \quad n \% p \neq 0 \end{cases}$$

$$T(1) = (n-1)t_{calc}$$

$$T(p) = (r-1)t_{calc} + (\log_2 p)t_{calc}$$

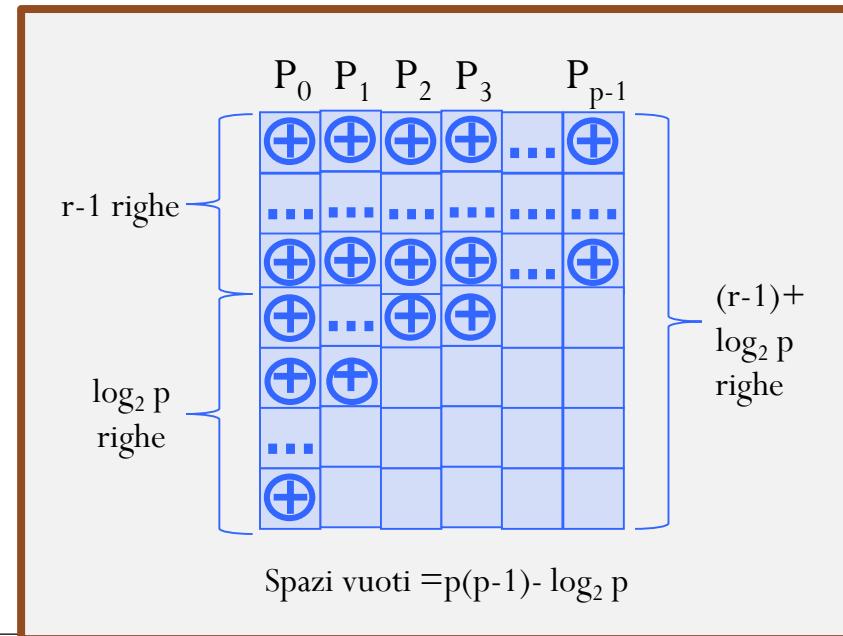


# Strategia II – Speed up?

$$S(p) = \frac{T(1)}{T(p)} = \frac{(n-1)t_{calc}}{(r-1 + \log_2 p)t_{calc}}$$

$$T(1) = (n-1)t_{calc}$$

$$T(p) = (r-1)t_{calc} + (\log_2 p)t_{calc}$$

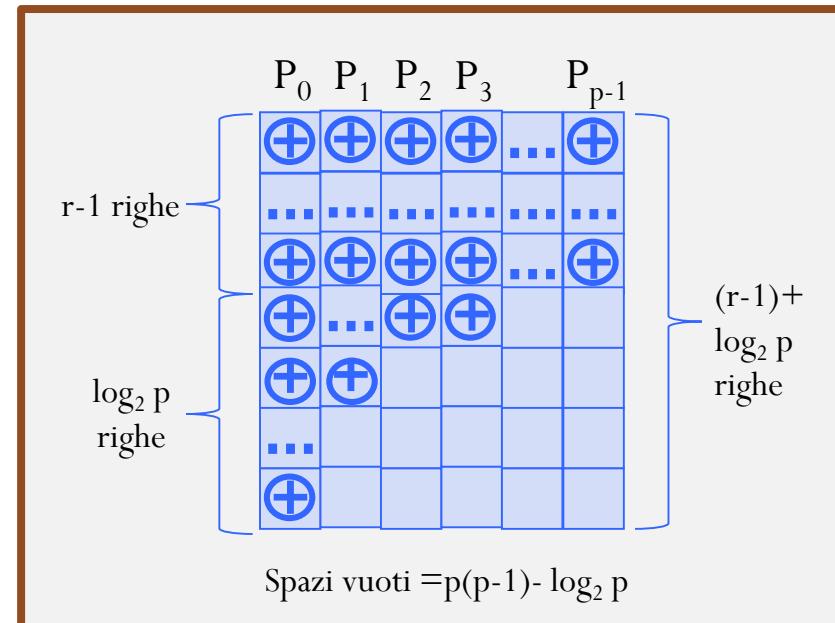


# Strategia II – Speed up?

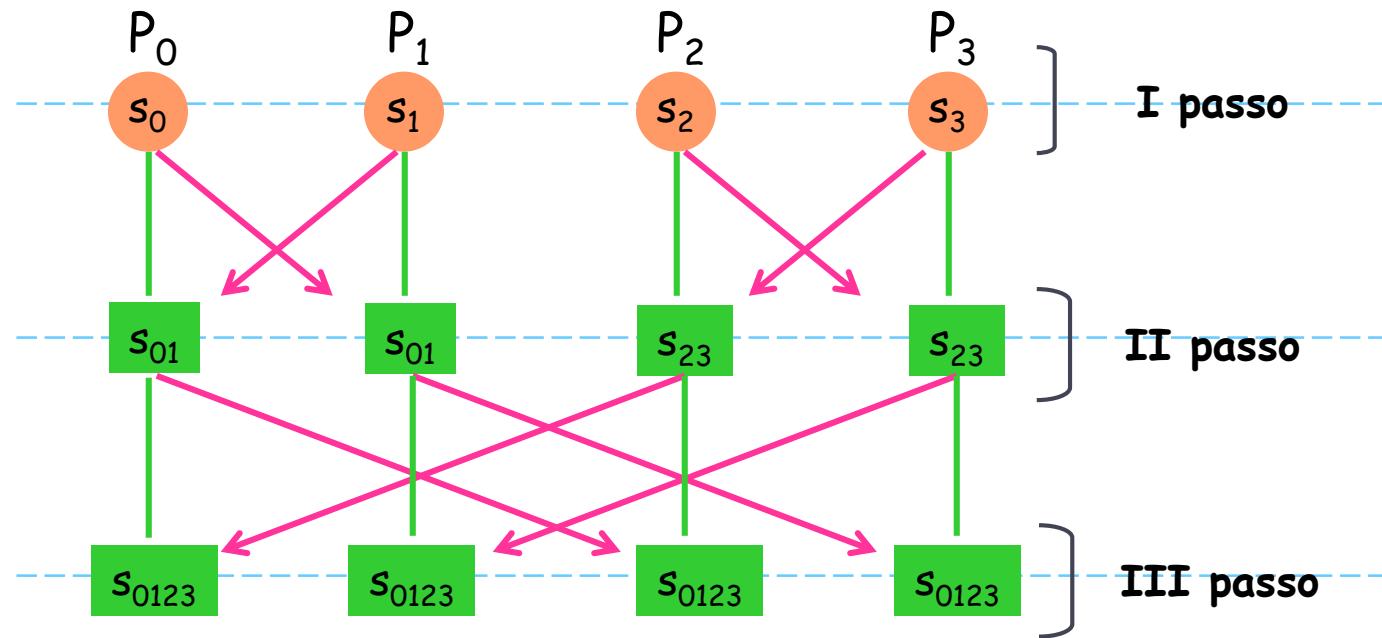
$$S(p) = \frac{T(1)}{T(p)} = \frac{(n-1)t_{calc}}{(r-1 + \log_2 p)t_{calc}}$$

$$T(1) = (n-1)t_{calc}$$

$$T(p) = (r-1)t_{calc} + (\log_2 p)t_{calc}$$



# Strategia III - Quanti passi di calcolo?



# Strategia III - Quanti passi di calcolo?

Dati :  $n$  ( $\geq 2$ ) numeri,  $p = 2^k$  processi, con  $k \leq \log_2 n$  e

$$r = \left\lceil \frac{n}{p} \right\rceil = \begin{cases} \frac{n}{p} & se \quad n \% p = 0 \\ \frac{n}{p} + 1 & se \quad n \% p \neq 0 \end{cases}$$

$$T(1) = (n - 1)t_{calc}$$

$$T(p) = (r - 1)t_{calc} + (\log_2 p)t_{calc}$$

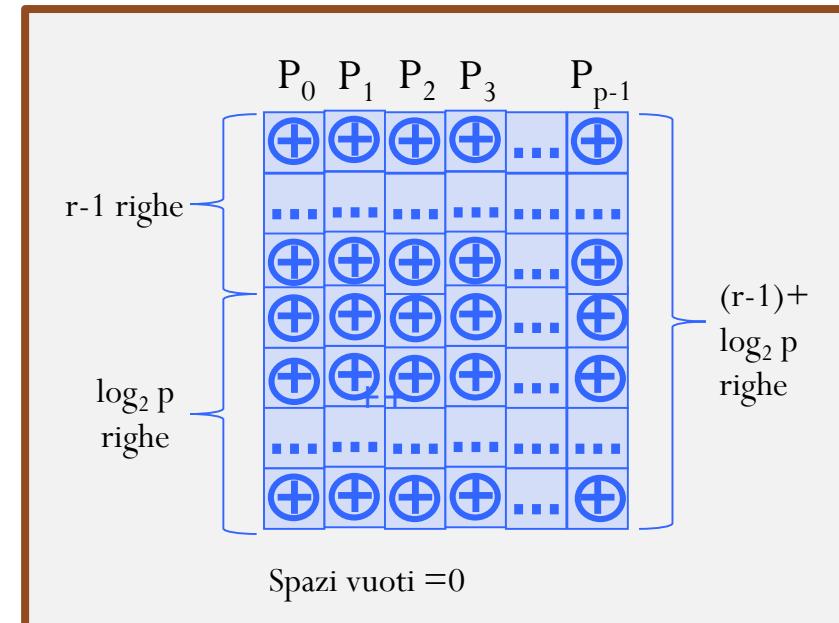
# Strategia III - Quanti passi di calcolo?

Dati :  $n (\geq 2)$  numeri,  $p = 2^k$  processi, con  $k \leq \log_2 n$  e

$$r = \left\lceil \frac{n}{p} \right\rceil = \begin{cases} \frac{n}{p} & se \quad n \% p = 0 \\ \frac{n}{p} + 1 & se \quad n \% p \neq 0 \end{cases}$$

$$T(1) = (n-1)t_{calc}$$

$$T(p) = (r-1)t_{calc} + (\log_2 p)t_{calc}$$

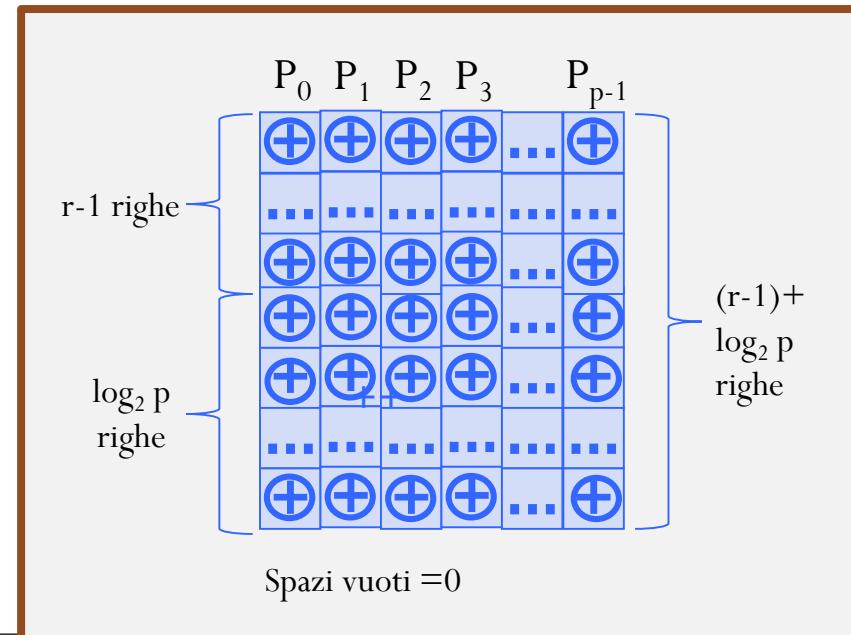


# Strategia III – Speed up?

$$S(p) = \frac{T(1)}{T(p)} = \frac{(n-1)t_{calc}}{(r-1 + \log_2 p)t_{calc}}$$

$$T(1) = (n-1)t_{calc}$$

$$T(p) = (r-1)t_{calc} + (\log_2 p)t_{calc}$$

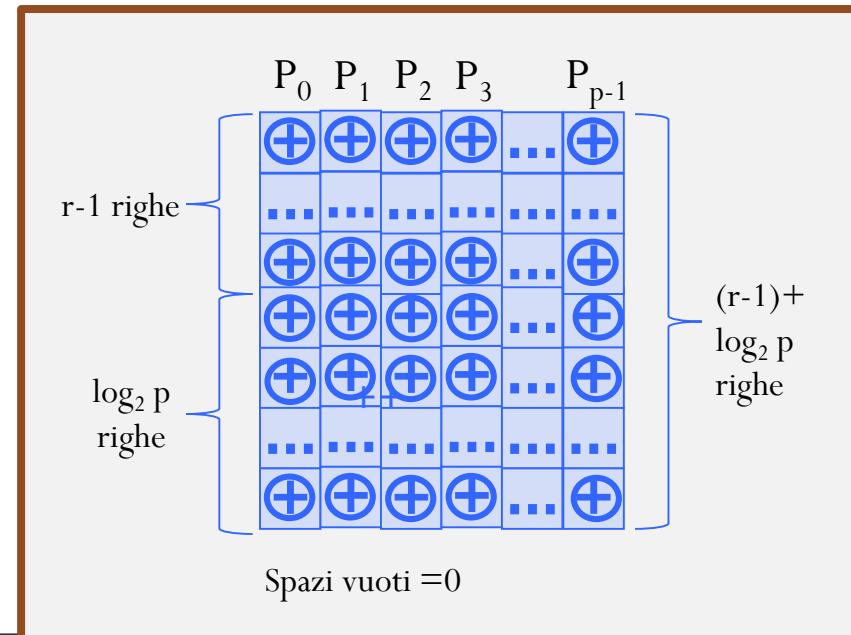


# Strategia III – Speed up?

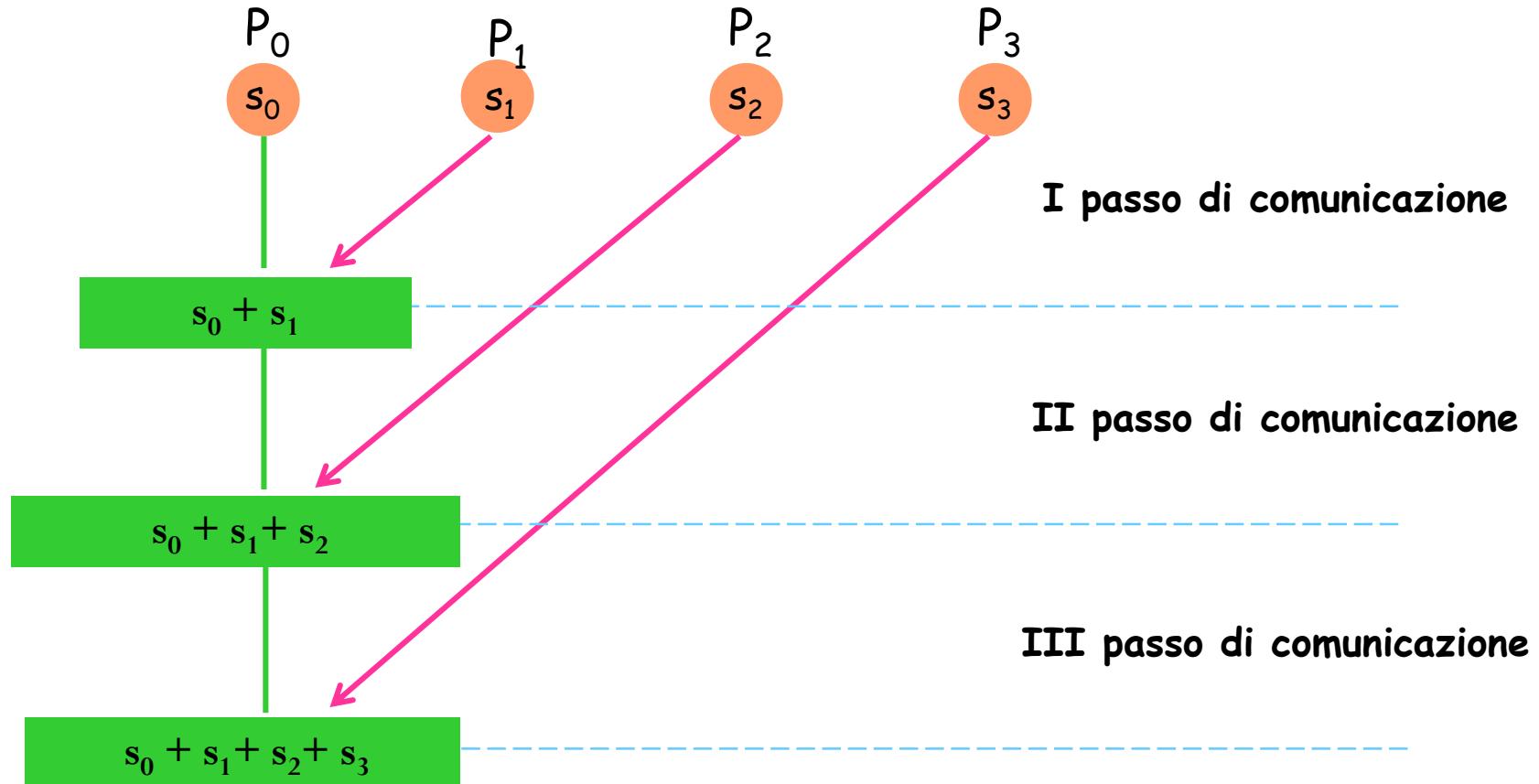
$$S(p) = \frac{T(1)}{T(p)} = \frac{(n-1)t_{calc}}{(r-1 + \log_2 p)t_{calc}}$$

$$T(1) = (n-1)t_{calc}$$

$$T(p) = (r-1)t_{calc} + (\log_2 p)t_{calc}$$



# Strategia I – Passi di comunicazione?



# Strategia I – Passi di comunicazione?

Dati :  $\mathbf{n}$  ( $\geq 2$ ) numeri,  $\mathbf{p}$  processi, con  $p \leq 2n$  e

$$r = \left\lceil \frac{n}{p} \right\rceil = \begin{cases} \frac{n}{p} & se \quad n \% p = 0 \\ \frac{n}{p} + 1 & se \quad n \% p \neq 0 \end{cases}$$

$$T(1) = (n - 1)t_{calc}$$

$$T(p) = (r - 1)t_{calc} + (p - 1)t_{calc} + (p - 1)t_{com}$$

$t_{com}$  = tempo di comunicazione di 1 numero

# Strategia I – Speed up?

$$S(p) = \frac{T(1)}{T(p)} = \frac{(n-1)t_{calc}}{(r+p-2)t_{calc} + (p-1)t_{com}}$$

$$T(1) = (n-1)t_{calc}$$

$$T(p) = (r-1)t_{calc} + (p-1)t_{calc} + (p-1)t_{com}$$

$t_{com}$  = tempo di comunicazione di 1 numero

# Strategia I – Speed up?

$$S(p) = \frac{T(1)}{T(p)} = \frac{(n-1)t_{calc}}{(r+p-2)t_{calc} + h(p-1)t_{calc}}$$

$$t_{com} = ht_{calc}$$

$$T(1) = (n-1)t_{calc}$$

$$T(p) = (r-1)t_{calc} + (p-1)t_{calc} - h(p-1)t_{calc}$$

$t_{com}$  = tempo di comunicazione di 1 numero

# Strategia I – Speed up?

$$S(p) = \frac{T(1)}{T(p)} = \frac{(n-1)t_{calc}}{(r+p-2)t_{calc} + 2(p-1)t_{calc}}$$

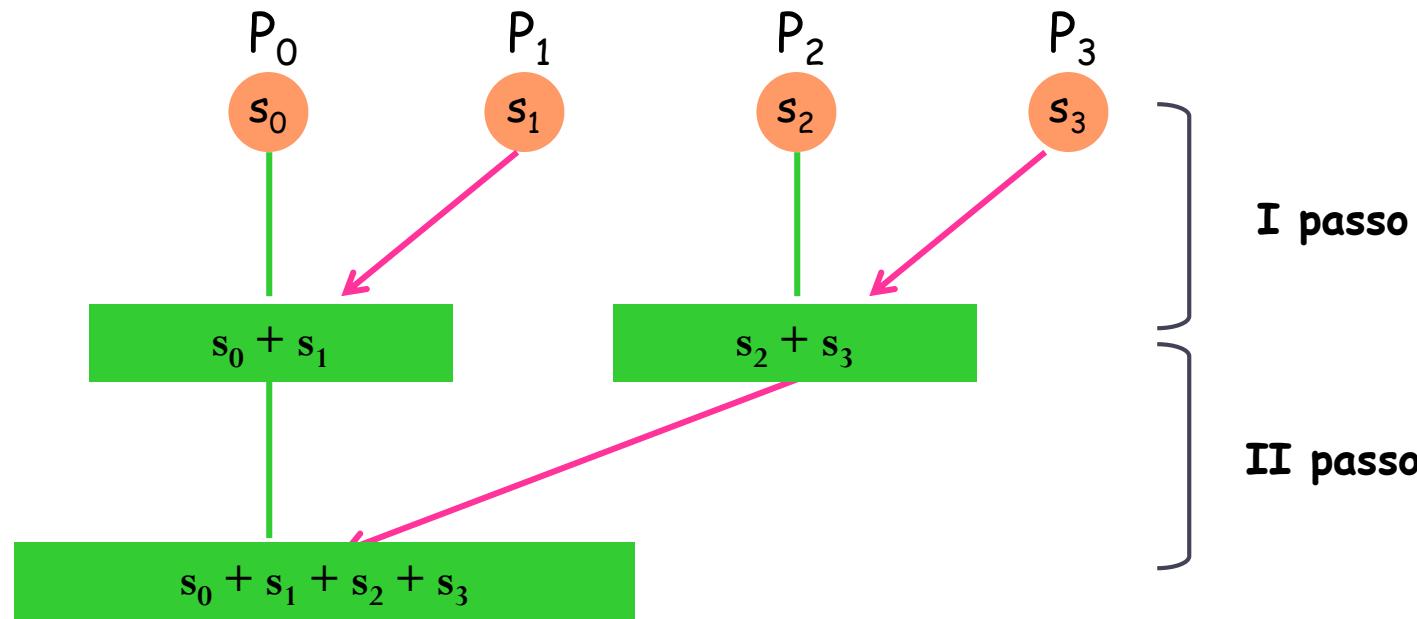
$$t_{com} = 2t_{calc}$$

$$T(1) = (n-1)t_{calc}$$

$$T(p) = (r-1)t_{calc} + (p-1)t_{calc} - 2(p-1)t_{calc}$$

$t_{com}$  = tempo di comunicazione di 1 numero

# Strategia II - Passi di comunicazione?



# Strategia II – Passi di comunicazione?

Dati :  $n$  ( $\geq 2$ ) numeri,  $p = 2^k$  processi, con  $k \leq \log_2 n$  e

$$r = \left\lceil \frac{n}{p} \right\rceil = \begin{cases} \frac{n}{p} & se \quad n \% p = 0 \\ \frac{n}{p} + 1 & se \quad n \% p \neq 0 \end{cases}$$

$$T(1) = (n - 1)t_{calc}$$

$$T(p) = (r - 1)t_{calc} + (\log_2 p)t_{calc} + \log_2 pt_{com}$$

# Strategia II – Speed up?

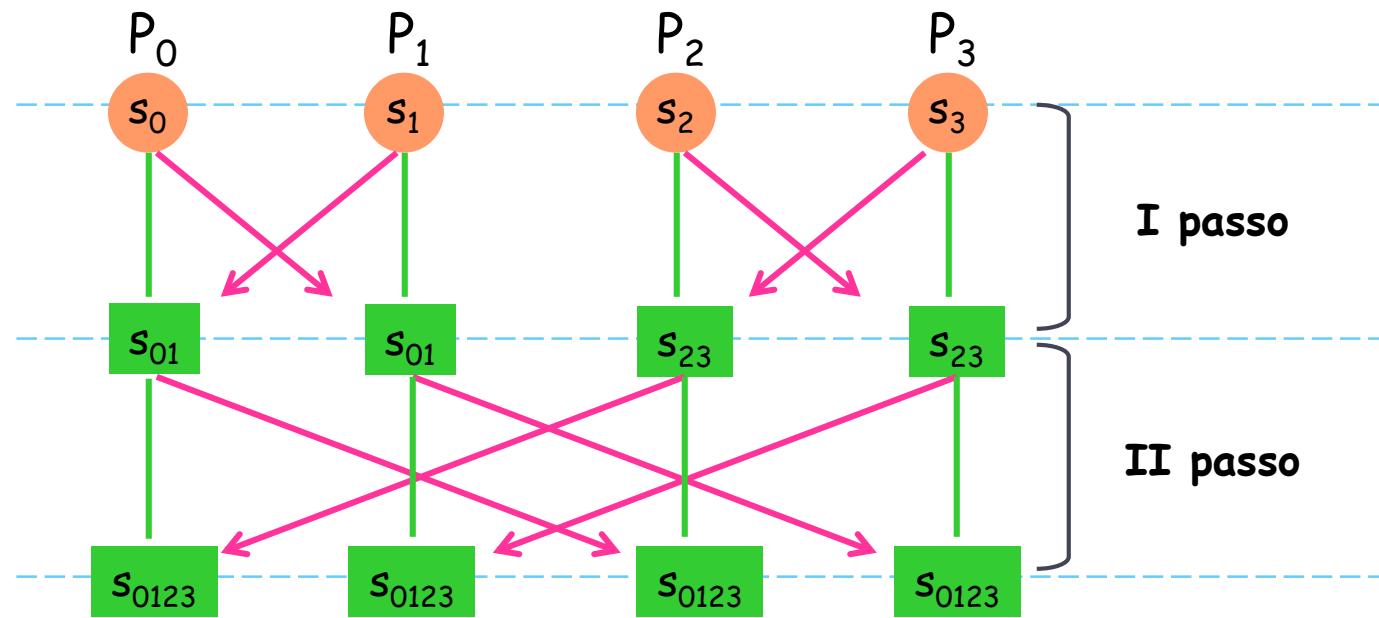
$$S(p) = \frac{T(1)}{T(p)} = \frac{(n-1)t_{calc}}{(r-1 + \log_2 p)t_{calc} + h \log_2 p t_{calc}}$$

$$t_{com} = ht_{calc}$$

$$T(1) = (n-1)t_{calc}$$

$$T(p) = (r-1)t_{calc} + (\log_2 p)t_{calc} - h \log_2 p t_{calc}$$

# Strategia III – Passi di comunicazione?



# Strategia III – Passi di comunicazione?

Dati :  $n$  ( $\geq 2$ ) numeri,  $p = 2^k$  processi, con  $k \leq \log_2 n$  e

$$r = \left\lceil \frac{n}{p} \right\rceil = \begin{cases} \frac{n}{p} & se \quad n \% p = 0 \\ \frac{n}{p} + 1 & se \quad n \% p \neq 0 \end{cases}$$

$$T(1) = (n - 1)t_{calc}$$

$$T(p) = (r - 1)t_{calc} + (\log_2 p)t_{calc} + \log_2 pt_{com}$$

# Strategia III – Speed Up?

$$S(p) = \frac{T(1)}{T(p)} = \frac{(n-1)t_{calc}}{(r-1 + \log_2 p)t_{calc} + h \log_2 p t_{calc}}$$

$$t_{com} = ht_{calc}$$

$$T(1) = (n-1)t_{calc}$$

$$T(p) = (r-1)t_{calc} + (\log_2 p)t_{calc} + h \log_2 p t_{calc}$$

# Cenni sulla costruzione

Algoritmo per la Somma di n numeri

# Problema

**Scrivere un algoritmo parallelo per il calcolo della somma di n numeri**

# Scrivere un algoritmo parallelo per il calcolo della somma di n numeri

- Descriviamo 4 fasi fondamentali separatamente, costruendo una bozza per il codice implementativo:
  - Lettura e distribuzione dei dati
  - Calcolo del sottoproblema
  - Eventuali comunicazioni per il calcolo del risultato finale
  - Stampa del risultato

# Dichiarazioni

/\* SCOPO: Calcola la somma di un insieme di interi

Consideriamo:

numero di processi  $\geq 1$

numero di elementi  $\geq$  numero di processi \*/

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char *argv[]){
    int menum, nproc,... ;
    int n, nloc, tag, i,...;
    int *x, *xloc;
    MPI_Status status;
    ...
}
```

# Lettura e distribuzione dei dati (1)

## (prima versione)

```
MPI_Init(&argv, &argc);
MPI_Comm_rank(MPI_COMM_WORLD, &menu);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
if (menu==0){
    “Lettura dei dati di input: n e x”
    for(i=1;i<nproc;i++){
        tag=10+i
        MPI_Send(&n, 1, MPI_INT, i, tag, MPI_COMM_WORLD);
    }
}else{
    tag=10+menu
    MPI_Recv(&n, 1, MPI_INT, 0, tag, MPI_COMM_WORLD,&status);
```

# Lettura e distribuzione dei dati (1)

## (seconda versione)

```
MPI_Init(&argv, &argc);
MPI_Comm_rank(MPI_COMM_WORLD, &menu);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
if (menu==0){
    “Lettura dei dati di input: n e x”
}
MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
```

# Lettura e distribuzione dei dati (1)

```
MPI_Init(&argv, &argc);
MPI_Comm_rank(MPI_COMM_WORLD, &menu);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
if (menu==0){
    “Lettura dei dati di input: n e x”
}
MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
nloc=n/nproc
rest=n%nproc
if (menu<rest) nloc=nloc+1
“allocazione di xloc”
if (menu==0){
    xloc=x
```

# Lettura e distribuzione dei dati (2)

```
tmp=nloc
start=0
for (i=1;i<nproc;i++){
    start=start+tmp
    tag=22+i;
    if(i==rest) tmp=tmp-1
    MPI_Send(&x[start],tmp,MPI_INT,i,tag,MPI_COMM_WORLD);
}
}/*endif*/
else{
    tag=22+menu
    MPI_Recv(xloc,nloc,MPI_INT,0,tag, MPI_COMM_WORLD,&status);
}
```

# Lettura e distribuzione dei dati

## (NOTE)

- Attenzione all'allocazione di **x** ed **xloc**: in Po non sono necessarie entrambi!! Si può decidere di usare solo uno dei due vettori (il maggiore risparmio di memoria si ha usando solo **xloc**, ma bisogna spezzare in più parti le fasi di lettura-invio)
- La variabile **tmp** serve a gestire la situazione in cui i primi **rest-1** processori hanno un elemento in più degli altri
- La variabile **start**, ad ogni passo di invio di elementi, contiene il numero di elementi già spediti e quindi l'indice da cui ripartire per il prossimo invio.
- Non è l'unico modo di scrivere questo algoritmo: si possono scrivere i cicli in modo diverso senza necessariamente cambiare il numero di operazioni, e si può certamente scriverne una versione ottimizzata.
- Alla fine di questa fase si può procedere con il calcolo locale delle somme parziali.

# Calcolo Locale

...

...

...

```
/*tutti i processori*/  
sum=0  
for(i=0;i<nloc;i++)  
    sum=sum+xloc[i]
```

...

...

...

# Calcolo della somma totale

## (I strategia di comunicazione)

...  
...  
...

```
if (menum==0){  
    for(i=1;i<nproc;i++){  
        tag=80+i  
        MPI_Recv(&sum_parz,1,MPI_int,i,tag,MPI_COMM_WORLD,&status);  
        sum=sum+sum_parz  
    }  
}else{  
    tag=menum+80  
    MPI_Send(&sum,1,MPI_INT,0,tag, MPI_COMM_WORLD);  
}
```

# Calcolo della somma totale

## (Il strategia di comunicazione)

...

```
for(i=0;i<log2nproc;i++){ /*passi di comunicazione*/
    if ((menum%2i)==0){ /*chi partecipa alla comunicazione*/
        if ((menum%2i+1)==0){ /*chi riceve*/
            “Ricevi da menum+2i”
        }else{
            “Spedisci a menum-2i”
        }
    }
}
```

...

# Calcolo della somma totale

## (III strategia di comunicazione)

...

```
for(i=0;i<log2nproc;i++){ /*passi di comunicazione*/
/*tutti partecipano ad ogni passo*/
    if ((menum%2i+1)<2i){ /*decidiamo solo a chi si invia e da chi si riceve*/
        “Ricevi da menum+2i”
        “Spedisci a menum+2i”
    }else{
        “Ricevi da menum-2i”
        “Spedisci a menum-2i”
    }
}
...
...
```

# Stampa del risultato (l'versione)

...

...

...

```
/*se ci basta che la stampi solo un processore (P0)*/
if (menum==0)
    printf("\nSomma totale=%d\n", sum);
```

...

...

...

# Stampa del risultato

## (Il versione)

...

...

...

/\*se vogliamo che la stampino tutti i processori\*/

```
printf("\nSono il processo %d: Somma totale=%d\n", menum,sum);
```

...

...

...

# Parallel and Distributed Computing

Esempio di script PBS  
Somma N con numeri memoria condivisa

Prof. Giuliano Laccetti

a.a. 2021-2022

```
#!/bin/bash

#####
#      #
# The PBS directives #
#      #
#####

#PBS -q studenti
#PBS -l nodes=1:ppn=8          # si riservano così 8 processori su un nodo. Se ne servono meno, si può mettere un numero minore.
#PBS -N somma
#PBS -o somma.out
#PBS -e somma.err
#####

# -q coda su cui va eseguito il job #
# -l numero di nodi richiesti #
# -N nome job(stesso del file pbs) #
# -o, -e nome files contenente l'output #

#####
#      #
#      qualche informazione sul job   #
#      #
#####

echo 'Job is running on node(s):'
cat $PBS_NODEFILE

PBS_O_WORKDIR=$PBS_O_HOME/ProgettoSommaOpenMP
echo -----
echo PBS: qsub is running on $PBS_O_HOST
```

```
echo PBS: originating queue is $PBS_O_QUEUE
echo PBS: executing queue is $PBS_QUEUE
echo PBS: working directory is $PBS_O_WORKDIR
echo PBS: execution mode is $PBS_ENVIRONMENT
echo PBS: job identifier is $PBS_JOBID
echo PBS: job name is $PBS_JOBNAME
echo PBS: node file is $PBS_NODEFILE
echo PBS: current home directory is $PBS_O_HOME
echo PBS: PATH = $PBS_O_PATH
echo -----
export OMP_NUM_THREADS=2          # numero di thread generati di default durante le regioni parallele
export PSC_OMP_AFFINITY=TRUE      # per legare i thread a particolari processori
echo "Compilo..."
gcc -fopenmp -lgomp -o $PBS_O_WORKDIR/somma $PBS_O_WORKDIR/sommaOpenMP.c

# nell'esempio il primo argomento è il numero di thread da utilizzare e il secondo la dimensione n
# il numero di thread deve essere minore o uguale del numero di processori riservati sul nodo
echo "Eseguo..."
$PBS_O_WORKDIR/somma 4 500000000
```

# **ESERCITAZIONE**

Codice parallelo per il prodotto  
matrice–vettore

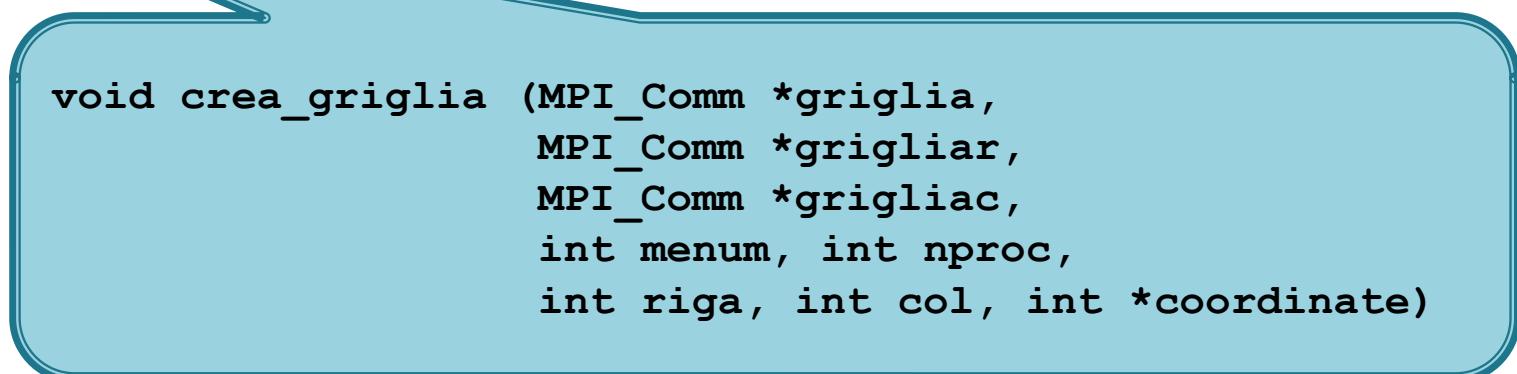
```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{
    int menum, nproc;
    int n,m; //dimensioni della matrice
    int ncol, mcol; //dimensioni delle sottomatrici
    int col, righe; //dimensioni della griglia
    int coord[2];
    int i,j;
    float *x,*A,*b;
    float *xloc, *bloc, *Aloc, *sumbloc;
    ...
    MPI_Status status;
    MPI_Comm *griglia, *grigliar, *grigliac;
    ...
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &menum);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{
    int menum, nproc;
    int n,m; //dimensioni della matrice
    int ncol, mcol; //dimensioni delle sottomatrici
    int col, righe; //dimensioni della griglia
    int coord[2];
    int i,j;
    float *x,*A,*b;
    float *xloc, *bloc, *Aloc, *sumbloc;
    ...
    MPI_Status status;
    MPI_Comm *griglia, *grigliar, *grigliac;
    ...
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &menum);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

Codice  
puramente  
indicativo!!  
Non scritto  
per  
funzionare  
né per  
essere  
copiato e  
funzionare!

```
tag = 1;  
if (menum==0)  
{  
    printf("Inserire il numero di righe della matrice:\n");  
    scanf("%d", &m);  
    printf("Inserire il numero di colonne della matrice:\n");  
    scanf("%d", &n);  
}  
MPI_Bcast(&m,1,MPI_INT, 0, MPI_COMM_WORLD);  
MPI_Bcast(&n,1,MPI_INT, 0, MPI_COMM_WORLD);  
...  
ASSEGNAZIONE col E righe (dimensioni griglia)  
...  
MPI_Bcast(&col,1,MPI_INT, 0, MPI_COMM_WORLD);  
MPI_Bcast(&righe,1,MPI_INT, 0, MPI_COMM_WORLD);  
...  
ALLOCAZIONI MEMORIA NECESSARIE  
...
```

```
...
/*CREAZIONE DELLA GRIGLIA DI PROCESSORI*/
crea_griglia(griglia, grigliar, grigliac, menum, nproc,
              righe,col, coord);
```



```
void crea_griglia (MPI_Comm *griglia,
                   MPI_Comm *grigliar,
                   MPI_Comm *grigliac,
                   int menum, int nproc,
                   int riga, int col, int *coordinate)
```

```
...
mloc = m/righe;
nloc = n/col;
...
DISTRIBUZIONE DEI DATI TRA I PROCESSORI
...
```

```
...
/*Fase di calcolo*/
for (i=0;i<mloc;i++) {
    for (j=0;j<nloc;j++) {
        bloc[i] += Aloc[i*nloc+j] * xloc[j];
    }
}
```

...

...

...

```
for (i=0;i<mloc;i++) {
    for (j=0;j<nloc;j++) {
        bloc[i]+= aloc[i][j] * xloc[j];
    }
}
```

...

```
/*Fase di comunicazione e raccolta del risultato*/
```

```
MPI_Allreduce(bloc,  
              sumbloc,  
              mloc,  
              MPI_FLOAT,  
              MPI_SUM,  
              grigliar); //somma sulle righe
```

```
MPI_Allgather(sumbloc,  
              mloc,  
              MPI_FLOAT,  
              b,mloc,  
              MPI_FLOAT,  
              grgliac); //collezione sulle colonne
```

...

```
int MPI_Allreduce(void *sendbuf,  
                  void *recvbuf,  
                  int count,  
                  MPI_Datatype datatype,  
                  MPI_Op op,  
                  MPI_Comm comm)
```

```
int MPI_Allgather(void *sendbuf,  
                  int sendcount,  
                  MPI_Datatype sendtype,  
                  void *recvbuf,  
                  int recvcount,  
                  MPI_Datatype recvtype,  
                  MPI_Comm comm)
```

```
...
/*Output*/
if (menum==0) {
    for (i=0;i<m;i++)
        printf("b(%d) = %f",i,b[i]);
}

...
MPI_Finalize();
return 0;
}
```

# Esercizi

Scrivere, compilare ed eseguire sul cluster attraverso il PBS i seguenti esercizi. Inviare il codice, il PBS e uno o più esempi di output a [valeria.mele@unina.it](mailto:valeria.mele@unina.it)

1. Sviluppare un algoritmo per il calcolo del **prodotto matrice-vettore**, in ambiente di calcolo parallelo su architettura MIMD a memoria distribuita, che utilizzi la libreria MPI.  
La matrice  $A \in \mathbb{R}^{m \times n}$  deve essere distribuita a  $p \times q$  processi, disposti secondo una topologia a griglia bidimensionale.  
L'algoritmo deve essere organizzato in modo da costruire e utilizzare una griglia bidimensionale dei processi.
- Utilizzare quanto sviluppato per gli esercizi precedenti!

# Esercizi

Scrivere, compilare ed eseguire sul cluster attraverso il PBS i seguenti esercizi. Inviare il codice, il PBS e uno o più esempi di output a [valeria.mele@unina.it](mailto:valeria.mele@unina.it)

1. Sviluppare un algoritmo per il calcolo del **prodotto matrice-vettore**, in ambiente di calcolo parallelo su architettura MIMD a memoria distribuita, che utilizzi la libreria MPI.  
La matrice  $A \in \mathbb{R}^{mxn}$  deve essere distribuita a  $pxq$  processi, disposti secondo una topologia a griglia bidimensionale.  
L'algoritmo deve essere organizzato in modo da costruire e utilizzare una griglia bidimensionale dei processi.
- Utilizzare quanto sviluppato per gli esercizi precedenti!

Entro la prossima  
settimana!!

# **ESERCITAZIONE**

Codice per la creazione di una  
griglia virtuale di processori

```
void crea_griglia (MPI_Comm *griglia, MPI_Comm *grigliar,
                    MPI_Comm *grigliac, int menum, int nproc,
                    int riga, int col, int *coordinate)

{
    int dim, *ndim, reorder, *period, vc[2];
    dim = 2;
    ndim = (int*) calloc (dim, sizeof(int));
    ndim[0] = riga;
    ndim[1] = col;
    period = (int*) calloc (dim, sizeof(int));
    period[0] = period [1] = 0;
    reorder = 0;
```

```
MPI_Cart_create(MPI_COMM_WORLD, dim, ndim,
                  period, reorder, griglia);
// creazione griglia
MPI_Cart_coords (*griglia, menum, 2, coordinate);
// assegnazione coordinate dei nodi
vc[0] = 0;
vc[1] = 1;
MPI_Cart_sub(*griglia, vc, grigliar);
// divisione in righe del comunicator
vc[0] = 1;
vc[1] = 0;
MPI_Cart_sub(*griglia, vc, grigliac);
// divisione in colonne del comunicator

return;
}
```

# Esercizi

Scrivere, compilare ed eseguire sul cluster attraverso il PBS i seguenti esercizi. Inviare il codice, il PBS e uno o più esempi di output a [valeria.mele@unina.it](mailto:valeria.mele@unina.it)

1. P processi. Creare una griglia di  $pxq$  processi, dove  $pxq=P$ , e far stampare ad ogni processo le proprie coordinate nella griglia. Input: P, p.
2. P processi, vettore V di dimensione N. Scrivere una funzione che distribuisca “equamente”<sup>1</sup> gli elementi di V tra i processi.
3. P processi, matrice M di dimensioni NxM. Scrivere una funzione che
  - I. Crei una griglia  $pxq$  dei processori
  - II. Individui  $pxq$  sottoblocchi rettangolari della matrice M
  - III. Assegni ad ogni processore il sottoblocco di M che ha le coordinate corrispondenti.

<sup>1</sup>Facendo in modo che il carico sia più bilanciato possibile

# Esercizi

Scrivere, compilare ed eseguire sul cluster attraverso il PBS i seguenti esercizi. Inviare il codice, il PBS e uno o più esempi di output a [valeria.mele@unina.it](mailto:valeria.mele@unina.it)

1. P processi. Creare una griglia di  $pxq$  processi, dove  $pxq=P$ , e far stampare ad ogni processo le proprie coordinate nella griglia. Input: P, p.
2. P processi, vettore V di dimensione N. Scrivere una funzione che distribuisca “equamente”<sup>1</sup> gli elementi di V tra i processi.
3. P processi, matrice M di dimensioni NxM. Scrivere una funzione che
  - I. Crei una griglia  $pxq$  dei processori
  - II. Individui  $pxq$  sottoblocchi rettangolari della matrice M
  - III. Assegni ad ogni processore il sottoblocco di M che ha le coordinate corrispondenti.

<sup>1</sup>Facendo in modo

**Entro la prossima  
settimana!!**

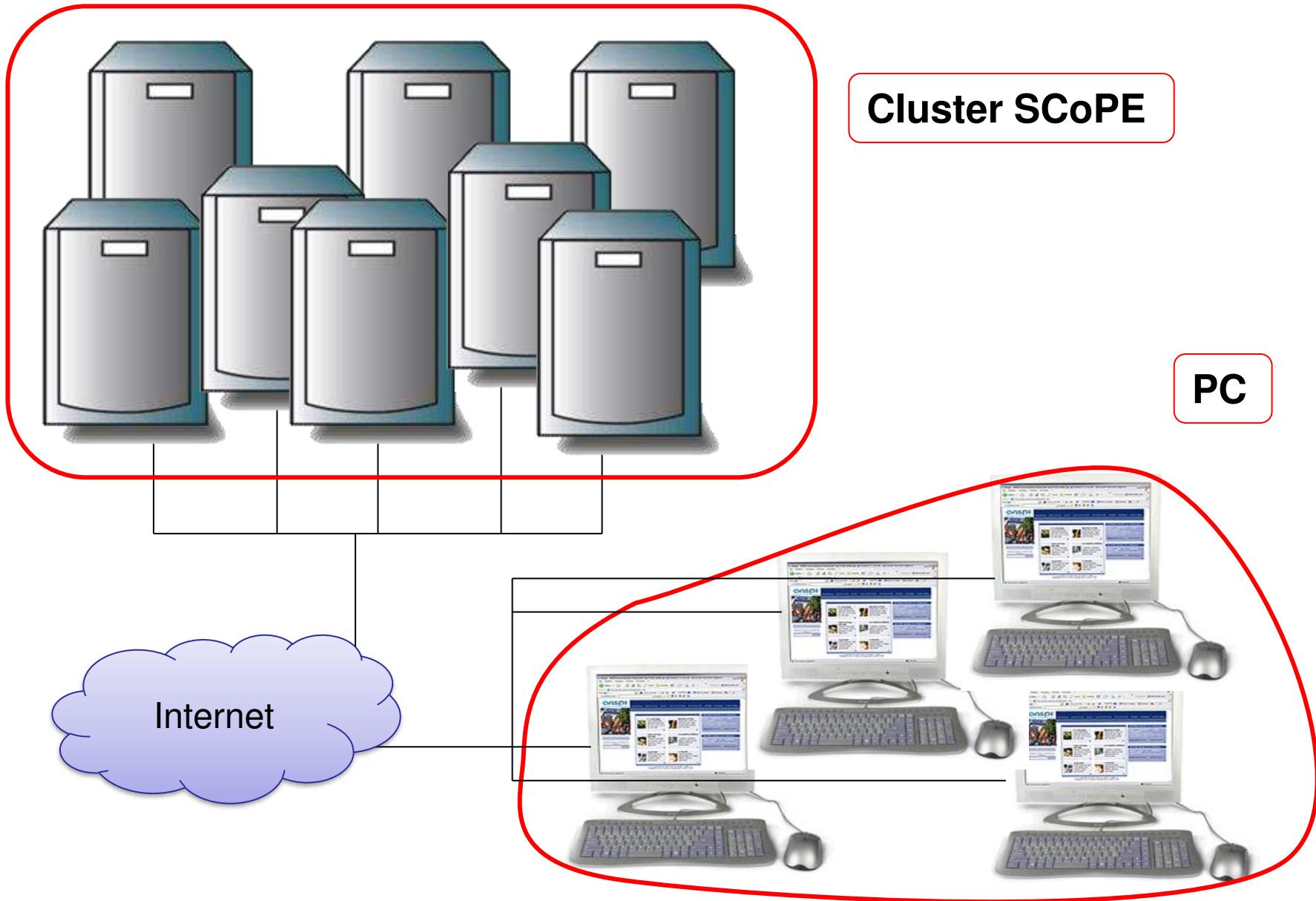
# Esercizi – Suggerimenti

Scrivere, compilare ed eseguire sul cluster attraverso il PBS i seguenti esercizi. Inviare il codice, il PBS e uno o più esempi di output a [valeria.mele@unina.it](mailto:valeria.mele@unina.it)

1. Sperimentare l'utilizzo delle sole funzioni di comunicazione uno a uno
2. Sperimentare l'utilizzo delle funzioni di comunicazione collettive viste (e non viste!) a lezione:
  - MPI\_Scatter
  - MPI\_Gather
  - MPI\_Allgather
  - MPI\_Scatterv
  - MPI\_Gatherv
  - ...
3. (Eventualmente) sperimentare l'utilizzo di tipi di dati più complessi (e delle funzioni fornite da MPI per gestirli) come:
  - MPI\_Type\_vector
  - MPI\_Type\_Datatype
  - ...

**Introduzione  
all'architettura parallela  
messa a disposizione dal  
*Progetto SCoPE***

# A disposizione



# A disposizione



**Cluster SCoPE**

Ogni nodo è **un server Dell PowerEdge M600** ciascuna dotato di:

- due processori quad core Intel Xeon E5410@2.33GHz (Architettura a 64 bit)
- 8 Gb di RAM
- 2 dischi SATA da 80GB in configurazione RAID1
- due schede Gigabit Ethernet configurate in bonding

# Come accedere al cluster

# Accesso al cluster

**Da un qualsiasi PC dove sia disponibile il protocollo ssh da riga di comando (Linux, Unix...)**

**ssh [login@ui-studenti.scope.unina.it](mailto:login@ui-studenti.scope.unina.it)**

**(la password all'inizio è uguale alla login, ma al primo accesso verrà chiesto di cambiarla)**

**Se siete su Windows l'operazione è simile ma vi è utile un client ssh, ed eventualmente un client sftp/scp**



# Accesso al cluster

E' necessario un client SSH.

Se siete su una piattaforma Windows, va sicuramente bene l'emulatore di terminale PuTTY, scaricabile gratuitamente da

<http://www.putty.org/>

**host name** ui-studenti.scope.unina.it  
**port** 22  
**connection type** SSH  
**login** personale, fornita a lezione

# Accesso al cluster

E' necessario un client SSH.

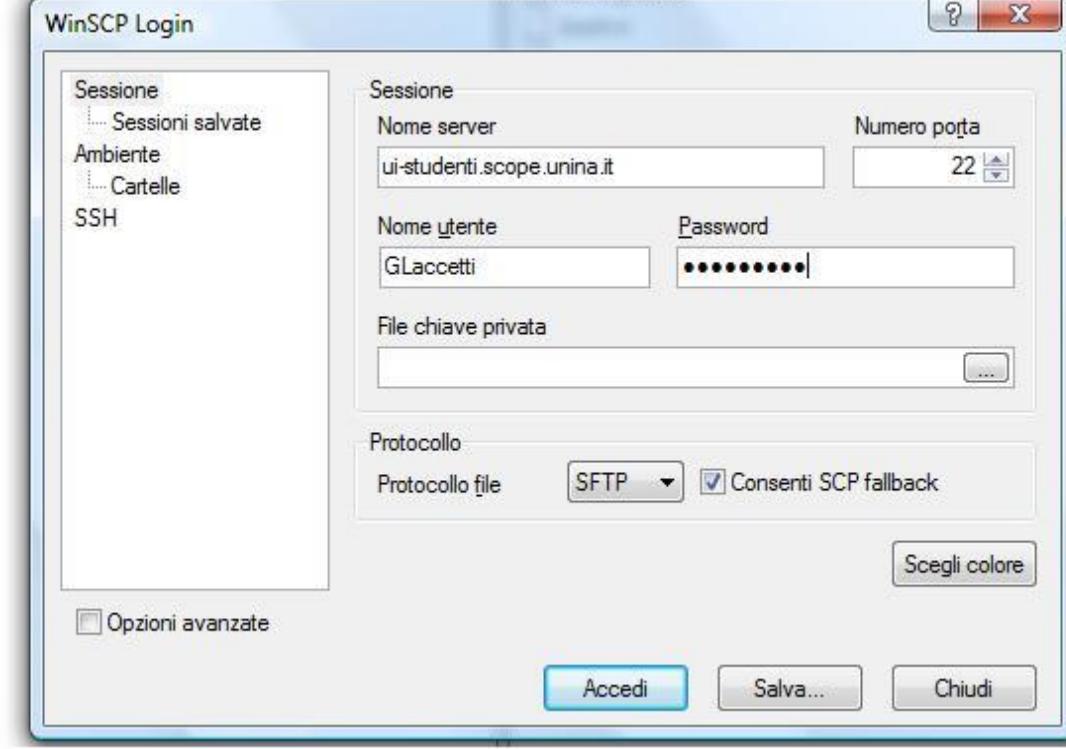
Se siete su una piattaforma Windows, va sicuramente bene il programma WinSCP, scaricabile gratuitamente da  
<https://winscp.net/eng/download.php>

Per spostare i file da e verso il cluster, può essere utile un client SFTP/FTP/SCP.

**host name** ui-studenti.scope.unina.it  
**port** 22  
**connection type** SSH  
**login personale, fornita a lezione**

# Accesso al cluster

E' necessario un client SSH.

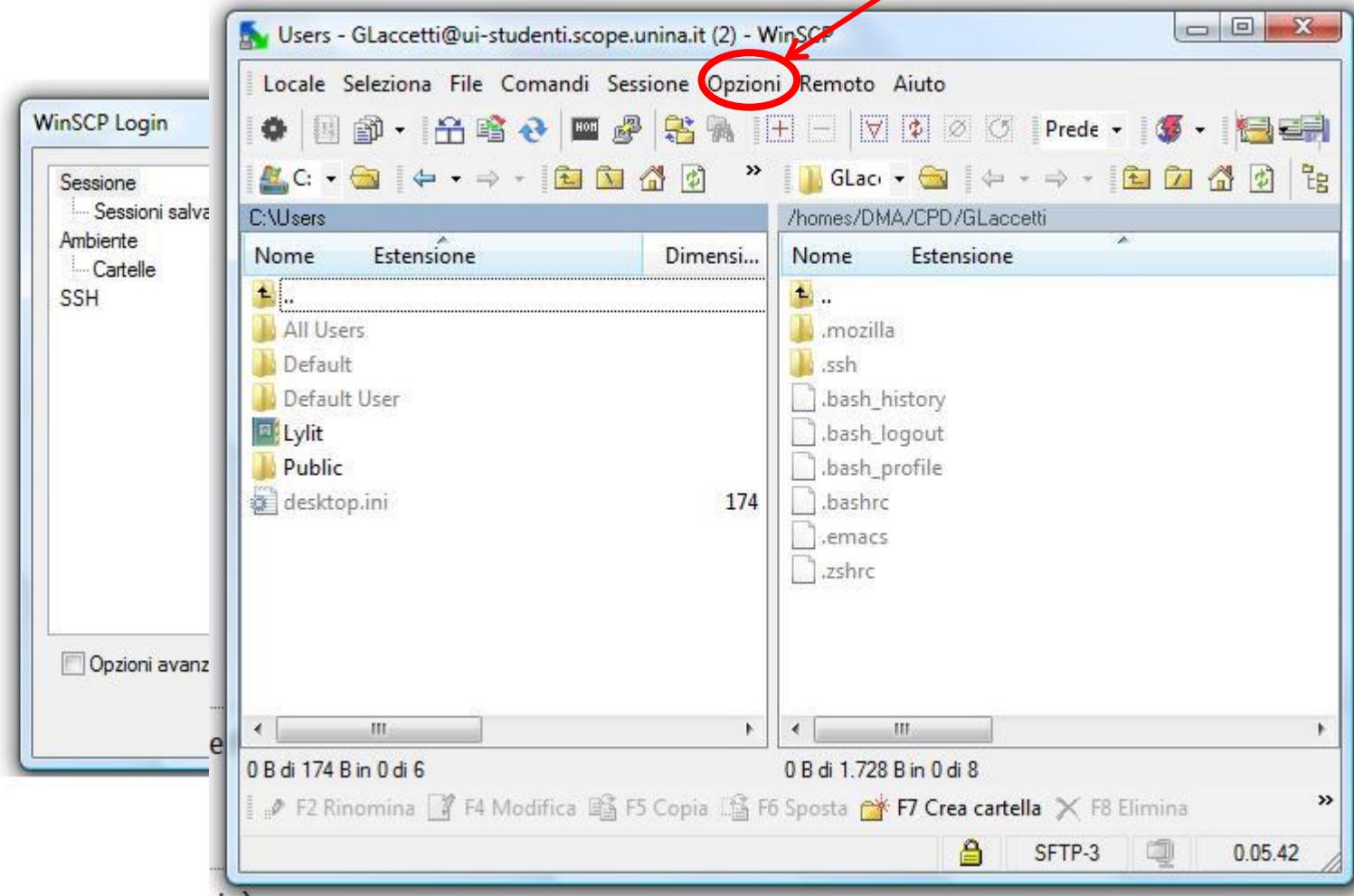


Se si sta su una piattaforma Windows, va  
nne il programma WinSCP,  
ile gratuitamente da  
<http://winscp.net/eng/download.php>

**t name** ui-studenti.scope.unina.it  
**t 22**  
**nection type** SSH  
**n personale, fornita a lezione**

# Accesso al cluster

E' necessario un client SSH.



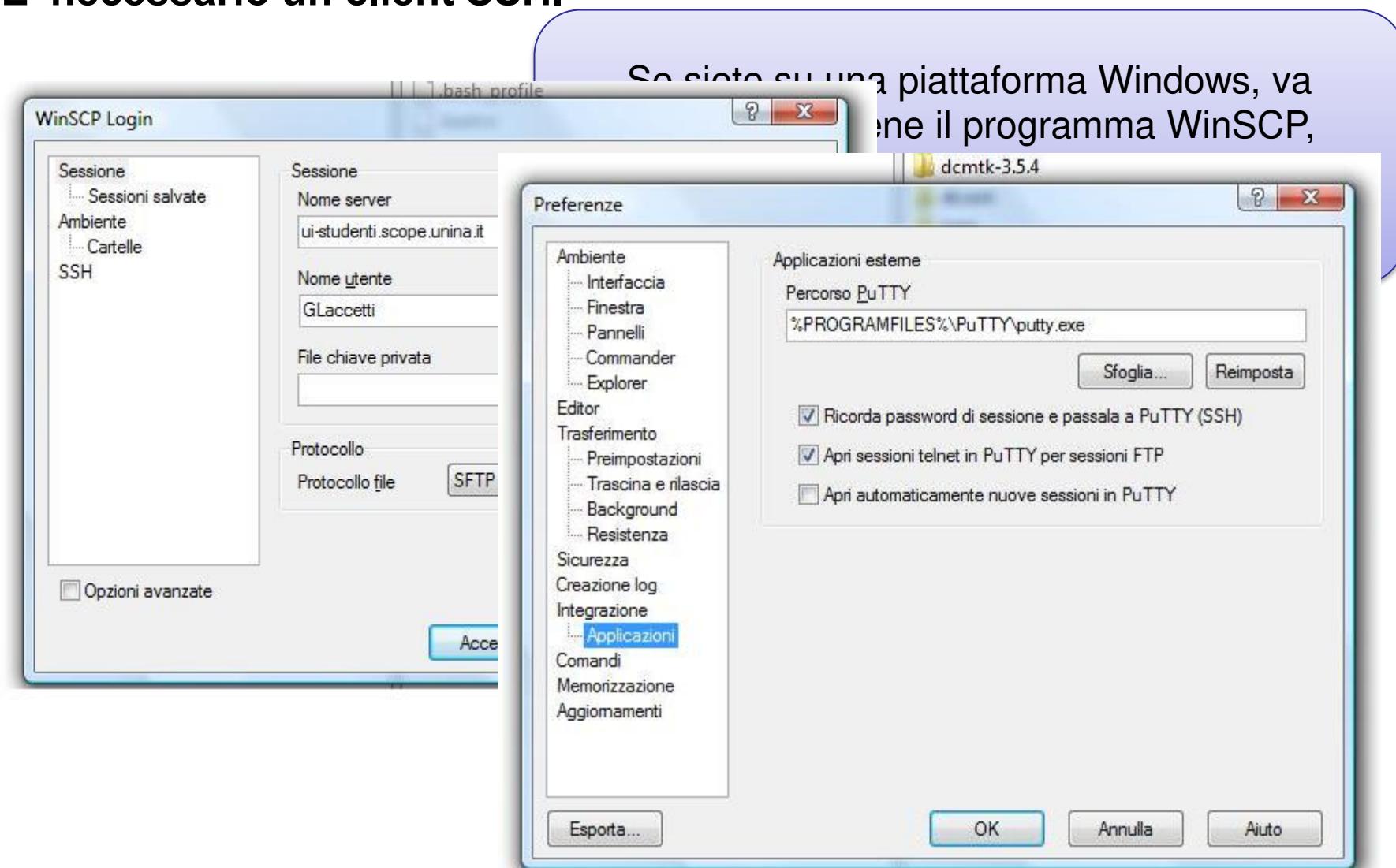
Preferenze

vs, va  
nSCP,  
php

unina.it  
ne

# Accesso al cluster

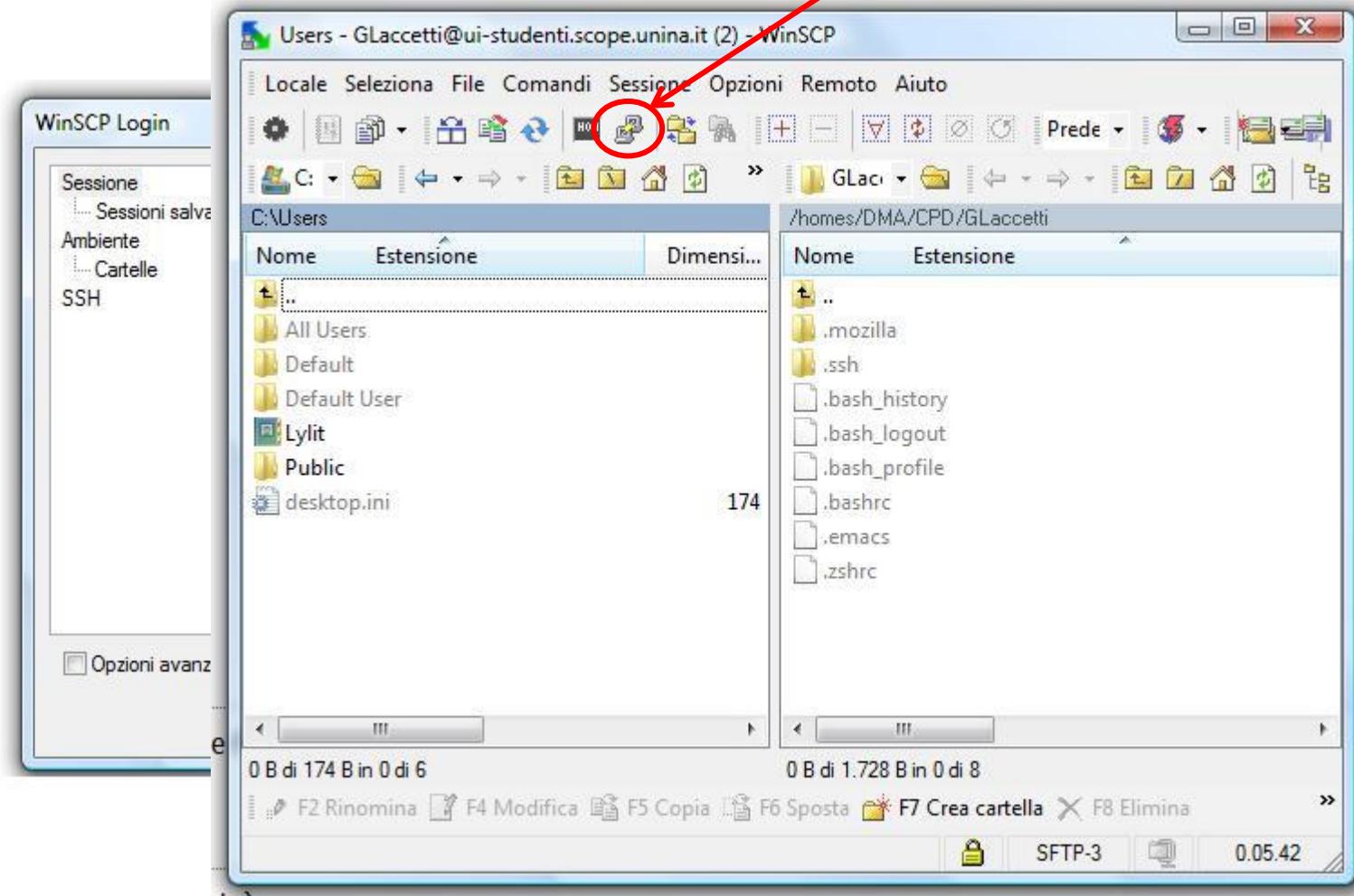
E' necessario un client SSH.



# Accesso al cluster

E' necessario un client SSH.

Sessione PuTTY



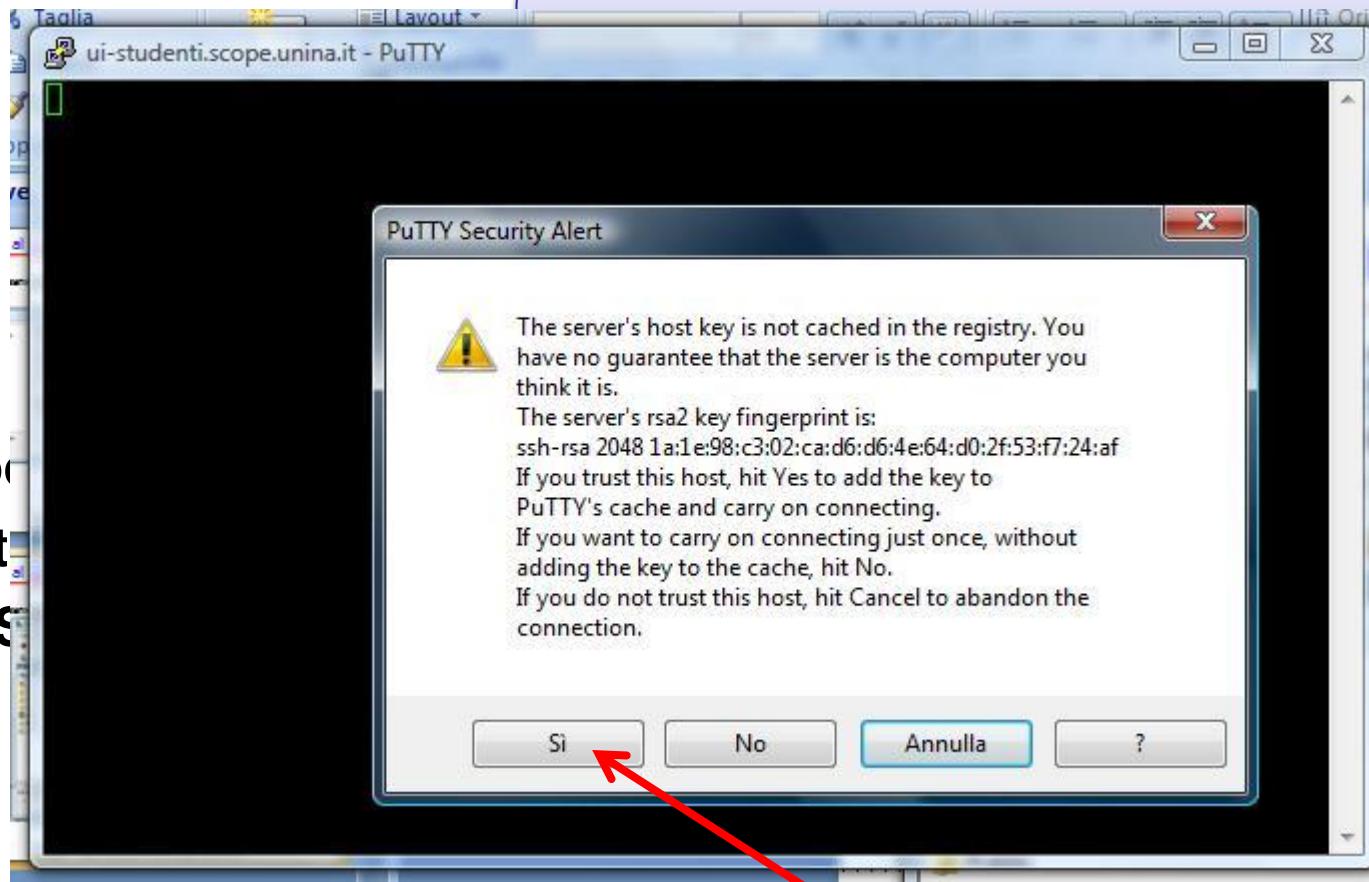
vs, va  
nSCP,  
[d.php](#)

unina.it  
ne

# Accesso al cluster

E' necessario un client SSH.

Sessione PuTTY



Per saperne di più su come accedere al cluster con un client SSH.

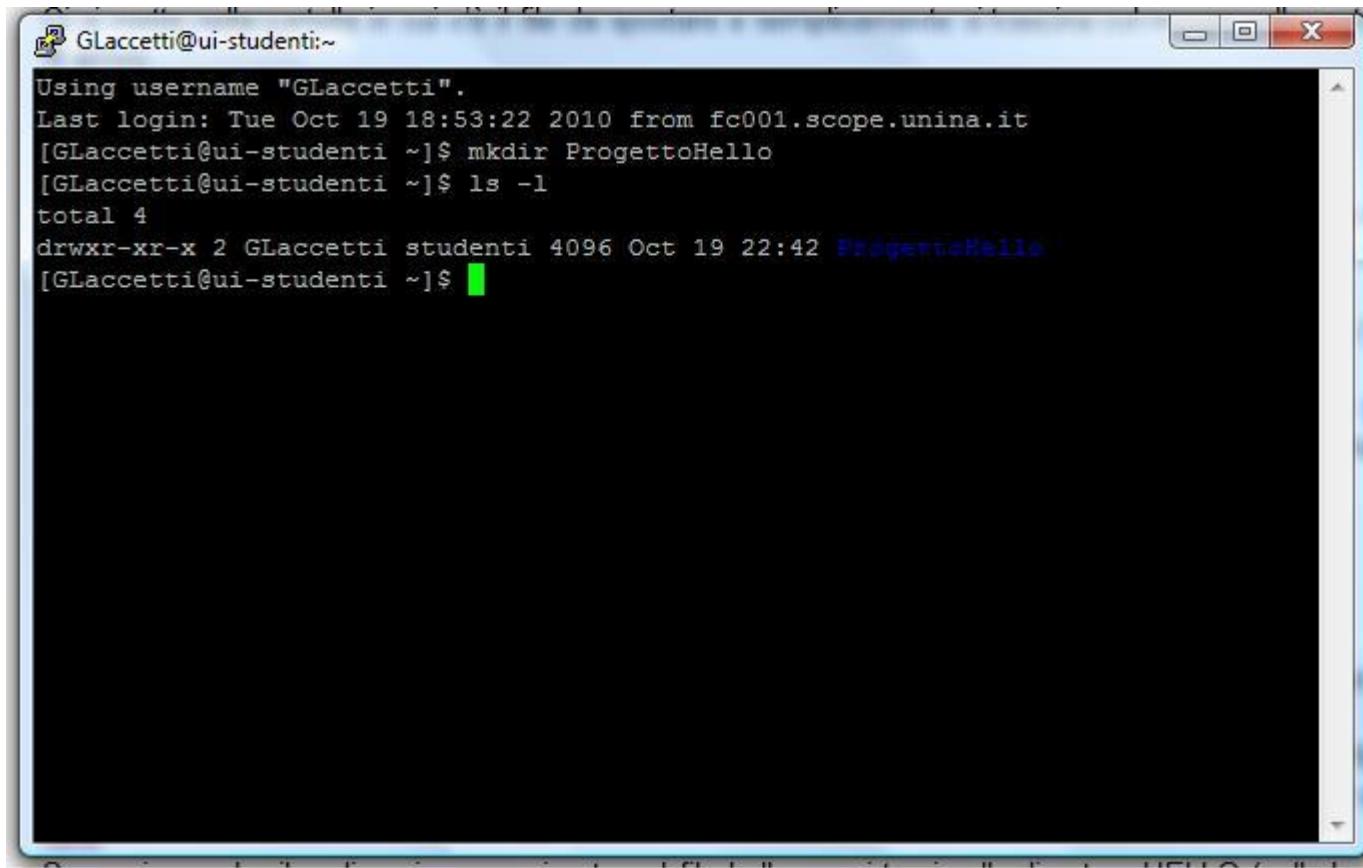


**ATTENZIONE!!!**  
**Al primo accesso vi verrà chiesto di  
cambiare la password!**

# Come lavorare sul cluster

# Scrivere un programma

Nella propria home directory, lo studente crea una cartella per il progetto che sta realizzando, per contenere file sorgente,eseguibile,...



The screenshot shows a terminal window titled "GLaccetti@ui-studenti:~". The window displays the following command-line session:

```
Using username "GLaccetti".
Last login: Tue Oct 19 18:53:22 2010 from fc001.scope.unina.it
[GLaccetti@ui-studenti ~]$ mkdir ProgettoHello
[GLaccetti@ui-studenti ~]$ ls -l
total 4
drwxr-xr-x 2 GLaccetti studenti 4096 Oct 19 22:42 ProgettoHello
[GLaccetti@ui-studenti ~]$
```

The terminal window has a blue header bar and a white body. The text is black on a white background. The cursor is visible at the end of the last command line.

# Scrivere un programma

Quindi, deve scrivere il file sorgente: File C, che si serva della libreria MPI.  
Utilizzeremo il compilatore dell'implementazione OPENMPI.

Il file sorgente deve essere scritto con un editor Linux (vi, nano,...) oppure con un editor in ambiente Windows che non formatti il testo (es. notepad).

Se è stato scritto in Windows, una volta portato il file sul cluster, va eseguito il comando

**dos2unix NomeFileScrittoSuWindows.c**

# Disponibilità del cluster

---

L'accesso al cluster vi sarà possibile solo nei seguenti orari:

- dalle 14:00 alle 20:00 di tutti i lunedì del mese
- dalle 08:00 alle 20:00 di tutti i martedì del mese
- dalle 08:00 alle 20:00 di tutti i giovedì del mese
- dalle 14:00 alle 20:00 di tutti i venerdì del mese
- dalle 08:00 alle 20:00 di tutti i sabato del mese

L'indirizzo a cui inoltrare qualsiasi segnalazione riguardo il cluster, che vi invito a fare REPENTINAMENTE qualora riscontraste qualcosa che non va.

L'indirizzo è: [contactcenter@unina.it](mailto:contactcenter@unina.it) (cc [valeria.mele@unina.it](mailto:valeria.mele@unina.it) )  
scrivete dettagliando bene il problema e i tempi in cui si è verificato .

# Prime credenziali sul cluster

---

Le login per accedere al cluster sono contenute nel materiale del corso nel file  
**PDC2122-ElencoStudenti\_conlogin.pdf**  
nella colonna LOGIN

Attualmente, per ciascuno degli account, la password coincide con il CODICE FISCALE comunicato.

Al primo accesso al sistema verrà richiesto di modificarla.

---

# **FINE LEZIONE**

# **Parallel and Distributed Computing**

**a.a. 2021/2022**

**prof. Giuliano Laccetti**

**Linee guida per la realizzazione della  
documentazione di software parallelo**

**(28/10/2021)**

Le seguenti linee guida derivano da

**Lezioni di Calcolo Numerico e Programmazione e Calcolo Parallelo e Distribuito tenute dal prof Murli**

e, in parte, presenti in

**A. Murli - Lezioni di Laboratorio di Programmazione e Calcolo, 2° ed. - Liguori, 2002**

# Linee guida per la realizzazione della documentazione di software parallelo

## Documentazione esterna

### Definizione ed Analisi del problema

Scopo del software.

Breve descrizione del problema che si vuole risolvere, sottolineando gli aspetti importanti ed evidenziando le eventuali difficoltà affrontate.

### Descrizione dell'algoritmo

Approfondire la scelta di una strategia: motivare la decisione (eventualmente), descrivere la strategia in questione, quanto più precisamente possibile, con l'aiuto anche di schemi se occorre, ed accennare le differenze con le altre strategie.

Descrivere il proprio algoritmo nel dettaglio, riportando possibilmente i passi salienti in pseudo-codice e spiegando le scelte implementative (es. il modo di assegnare dimensioni del sottoproblema diverse per ogni processore, il modo di individuare i processori che comunicano ad ogni passo, i controlli effettuati, ecc ...)

## Input ed Output

Descrivere cosa si dovrà dare in input al software al momento dell'utilizzo, come parametri e/o quando richiesto interattivamente dal software stesso. Spiegare che output aspettarsi, per i diversi tipi di input. Per entrambi spiegare la forma in cui i dati devono essere forniti o in cui le informazioni verranno restituite (specificare il tipo dei dati, descrivere i file eventualmente prodotti, spiegare come interpretare l'output a video).

## Indicatori di errore

Spiegazione delle situazioni di errore previste dalla routine e corrispondenti valori dei parametri di errore.

## Subroutine

Il software potrà essere composto da una o più routine. In entrambi i casi verranno utilizzate anche routine esterne, appartenenti a librerie non standard del C, come MPI.

Per quanto riguarda le routine di libreria utilizzate, riportare la testata e spiegare i parametri di input e di output.

Per quanto riguarda le routine scritte personalmente (es. funzioni per la stampa, per la lettura, per il calcolo,...) riportare la testata, spiegare i parametri di input e di output e dare una breve descrizione del funzionamento; aggiungere eventuali condizioni per il corretto utilizzo.

## Analisi dei tempi

Preliminariamente, calcolare i parametri di valutazione dell'algoritmo implementato:  $T(p)$ ,  $S(p)$ , ed  $E(p)$ .

Dunque, scelti degli esempi test, prendere i tempi delle diverse esecuzioni e costruire tabelle e grafici. Far variare la dimensione del problema.

I risultati che vengono evidenziati devono essere opportunamente commentati.

## Esempi d'uso

Riportare esempi di esecuzione del software, così come appare a video, partendo dal comando “mpirun”, fino all’output. Se si utilizzano o si generano file, riportarne il contenuto.

Fare esempi in cui si utilizzano diversi numeri di processori (da uno solo a 8), e su diverse dimensioni del problema. Se ci sono casi particolari o casi limite, riportare almeno un esempio.

## Riferimenti bibliografici

Se avete utilizzato materiale per studiare, o come riferimento per scrivere descrizioni e commenti, riportate in questa sezione libri, appunti di lezione, lucidi, articoli, siti web da cui questo materiale proviene, dove possibile specificando titolo ed autore.

## Appendice: codice

Riportare il codice scritto, compresa la DOCUMENTAZIONE INTERNA: commentate opportunamente il codice perché sia di facile lettura e comprensione per chi lo analizza, che ne potrà dare così migliore valutazione.

---

Per un eventuale approfondimento, si consiglia di consultare i links:

[https://www.nag.com/numeric/nl/nagdoc\\_latest/clhtml/front\\_matter/manconts.html](https://www.nag.com/numeric/nl/nagdoc_latest/clhtml/front_matter/manconts.html)

documentazione esterna delle routine di una libreria commerciale, la Libreria del NAG (Numerical Algorithms Group), interfaccia per linguaggio C

e

<https://www.nag.com/content/nag-parallel-library-manual-release-3>

documentazione esterna della versione parallela (interfaccia per linguaggio Fortran)

## Documentazione interna

Per quanto riguarda la documentazione interna, che consiste in ogni caso in vere e proprie linee di commento scritte nel corpo stesso del programma (e quindi rispettano le regole per i commenti del linguaggio di programmazione utilizzato), oltre ad usuali linee di commento che, per aiutare la lettura del codice sorgente, descrivono singole istruzioni o blocchi di istruzioni, in genere si riporta, sempre sotto forma di linee di commento, in testa al codice sorgente, una serie di informazioni che fanno parte anche della documentazione esterna (scopo, testata, descrizione parametri, esempio d'uso: in tal caso serve un esempio di input ed il corrispondente output, in genere un caso molto semplice, solo per verificare il funzionamento)

MPI :

# Message Passing Interface

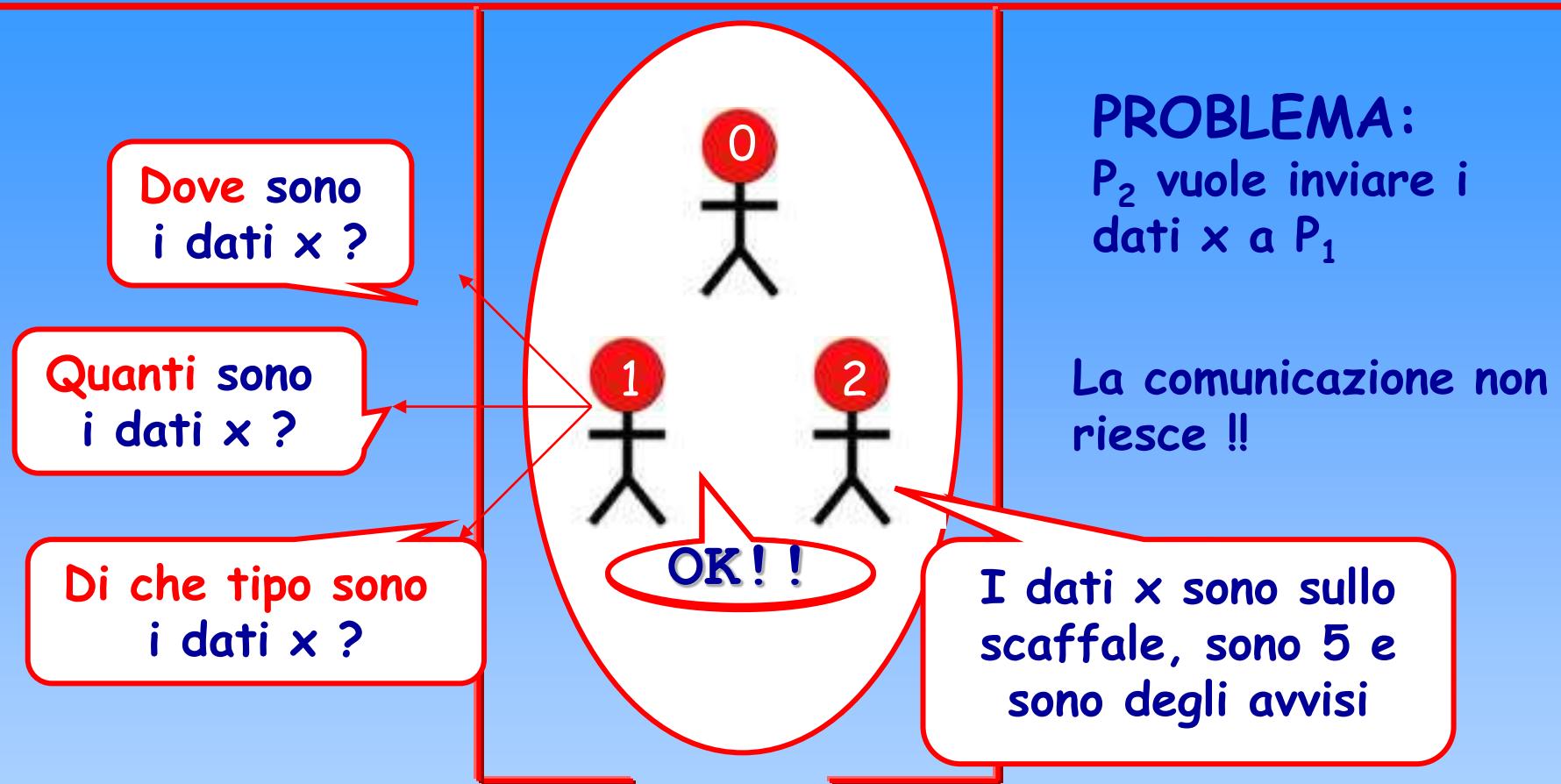
## MPI

MPI :

---

# Comunicazione di un messaggio.

# Il Communicator MPI :



**COMMUNICATOR\_A1**

**Comunicazione Riuscita !**

# Caratteristiche di un messaggio

Un dato che deve essere spedito o ricevuto attraverso un messaggio di MPI è descritto dalla seguente tripla  
**(address, count, datatype)**

Indirizzo  
in memoria del dato

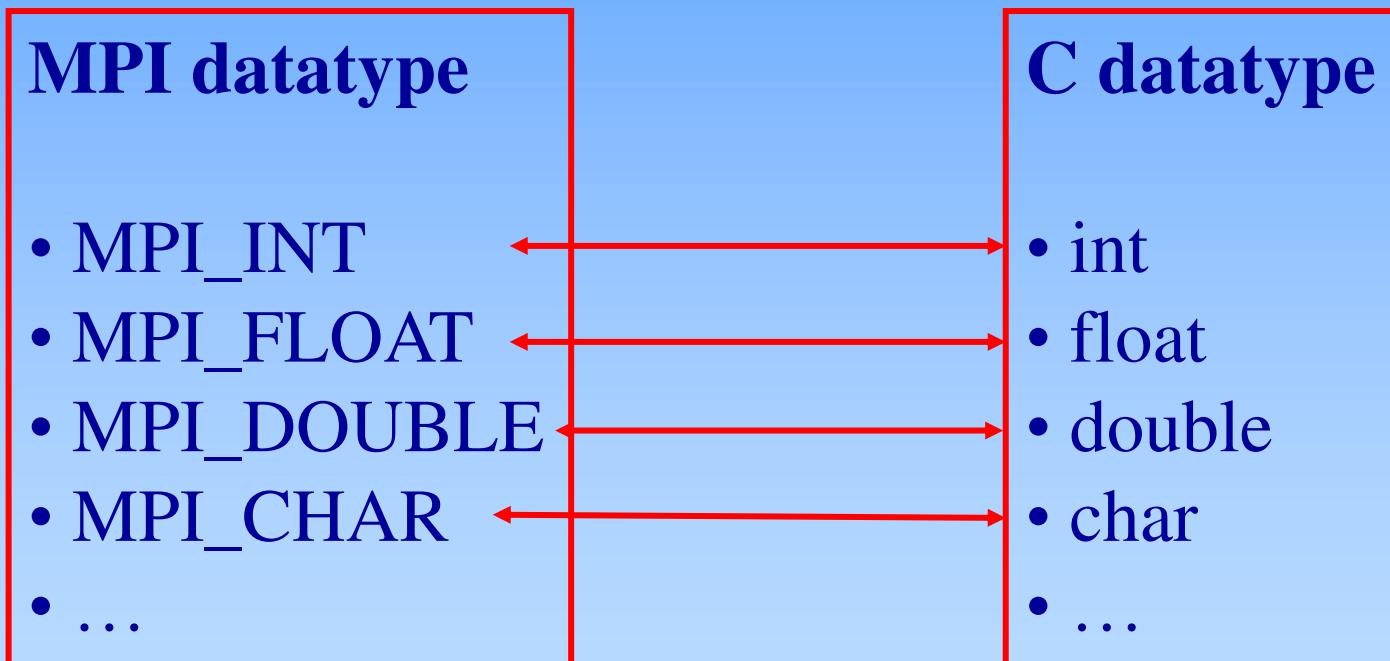
Dimensione del dato

Tipo del dato

Un datatype di MPI è **predefinito** e **corrisponde univocamente** ad un tipo di dato del linguaggio di programmazione utilizzato.

# Esempio 1: linguaggio C

Ogni tipo di dato di MPI corrisponde  
univocamente  
ad un tipo di dato del linguaggio C.



## Esempio 2: linguaggio Fortran

Ogni tipo di dato di MPI corrisponde  
univocamente  
ad un tipo di dato del linguaggio Fortran.

### MPI datatype

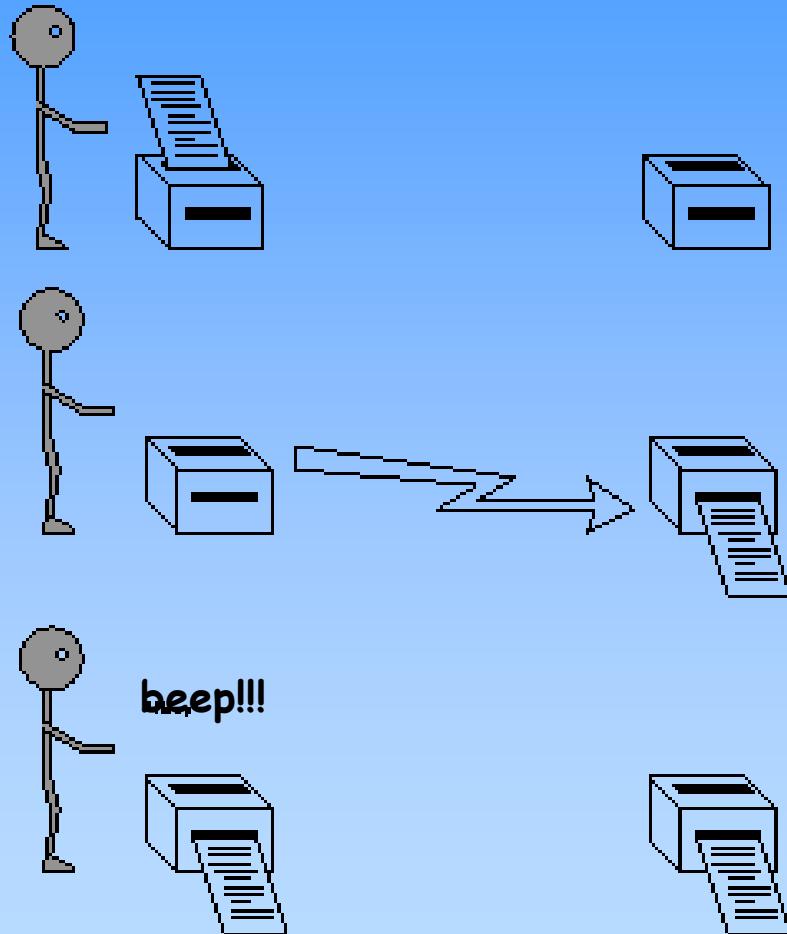
- MPI\_INT
- MPI\_FLOAT
- MPI\_DOUBLE
- MPI\_CHAR
- MPI\_LOGICAL
- ...

### Fortran datatype

- integer
- real
- double precision
- character
- logical
- ...

# Tipi di comunicazioni ( Esempio 1 ) :

## Trasmissione di un fax



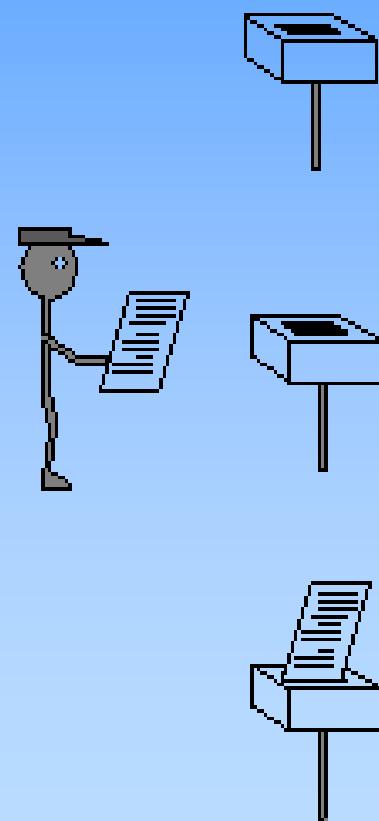
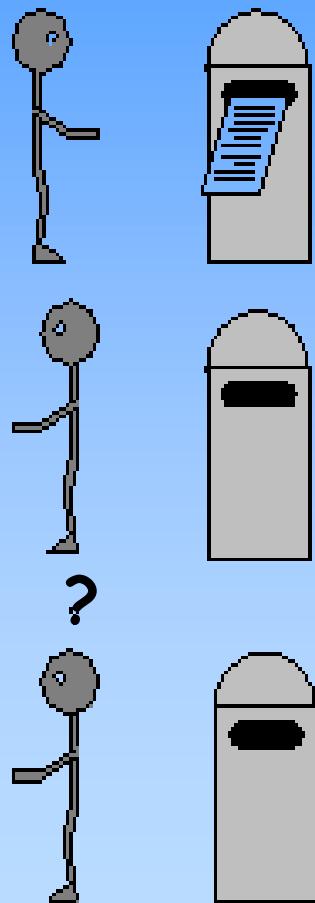
Il fax trasmittente termina l'operazione quando il fax ricevente ha ricevuto completamente il messaggio.



L'operazione di ricezione del messaggio è stata completata .

# Tipi di comunicazioni ( Esempio 2 ) :

## Spedizione di una lettera tramite servizio postale



Il mittente spedisce la lettera, ma non può sapere se è stata ricevuta .

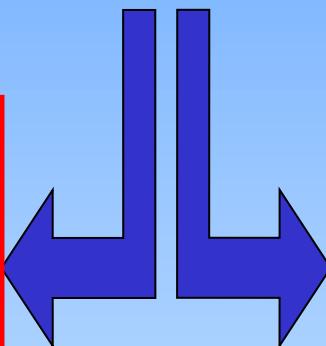


Il mittente non sa se l'operazione di ricezione del messaggio è stata completata.

# Tipi di comunicazioni:

La spedizione o la ricezione  
di un messaggio  
da parte di un processore può essere  
**bloccante** o **non bloccante**

Se un processore  
esegue una  
comunicazione  
**bloccante**  
**si arresta** fino a  
conclusione  
dell'operazione.



Se un processore  
esegue una  
comunicazione  
**non bloccante**  
**prosegue** senza  
preoccuparsi della  
conclusione  
dell'operazione.

# Comunicazioni bloccanti in MPI

Funzione **bloccante** per la *spedizione* di un messaggio:

**MPI\_Send**

Funzione **bloccante** per la *ricezione* di un messaggio:

**MPI\_Recv**

# Un semplice programma con 2 processori:

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    int menum, nproc;
    int n, tag, num;
    MPI_Status info;

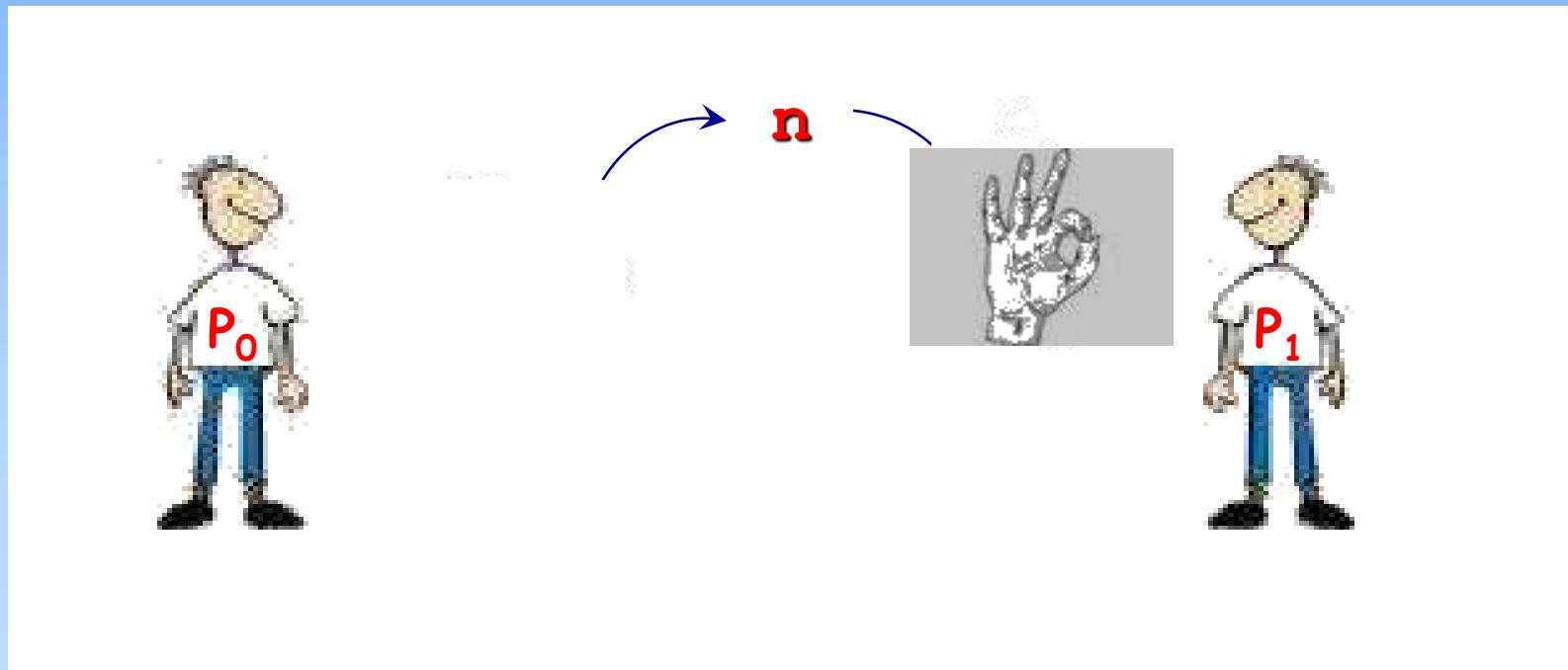
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&menum);
    if(menum==0)
    { scanf("%d",&n);
      tag=10;
      MPI_Send(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD);
    }
    else { tag=10;
      MPI_Recv(&n,1,MPI_INT,0,tag,MPI_COMM_WORLD,&info);
    }
    MPI_Get_count(&info,MPI_INT,&num);
    MPI_Finalize();
    return 0;
}
```



# Nel programma ... :

**`MPI_Send(&n, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);`**

- Con questa routine il processore chiamante  $P_0$  spedisce il parametro **n**, di tipo **`MPI_INT`** e di dimensione **1**, al processore  $P_1$ ; i due processori appartengono entrambi al communicator **`MPI_COMM_WORLD`**. Il parametro **tag** individua univocamente tale spedizione.



# In generale (comunicazione uno ad uno bloccante) :

```
MPI_Send(void *buffer, int count,  
        MPI_Datatype datatype, int dest,  
        int tag, MPI_Comm comm);
```

- Il processore che esegue questa routine spedisce i primi **count** elementi di **buffer**, di tipo **datatype**, al processore con identificativo **dest**.
- L'identificativo **tag** individua univocamente il messaggio nel contesto **comm**.

## In dettaglio...

```
MPI_Send(void *buffer, int count,  
          MPI_Datatype datatype, int dest,  
          int tag, MPI_Comm comm);
```

**\*buffer** indirizzo del dato da spedire

**count** numero dei dati da spedire

**datatype** tipo dei dati da spedire

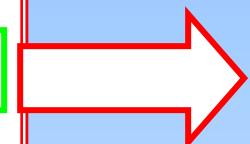
**dest** identificativo del processore destinatario

**comm** identificativo del communicator

**tag** identificativo del messaggio

# Un semplice programma con 2 processori:

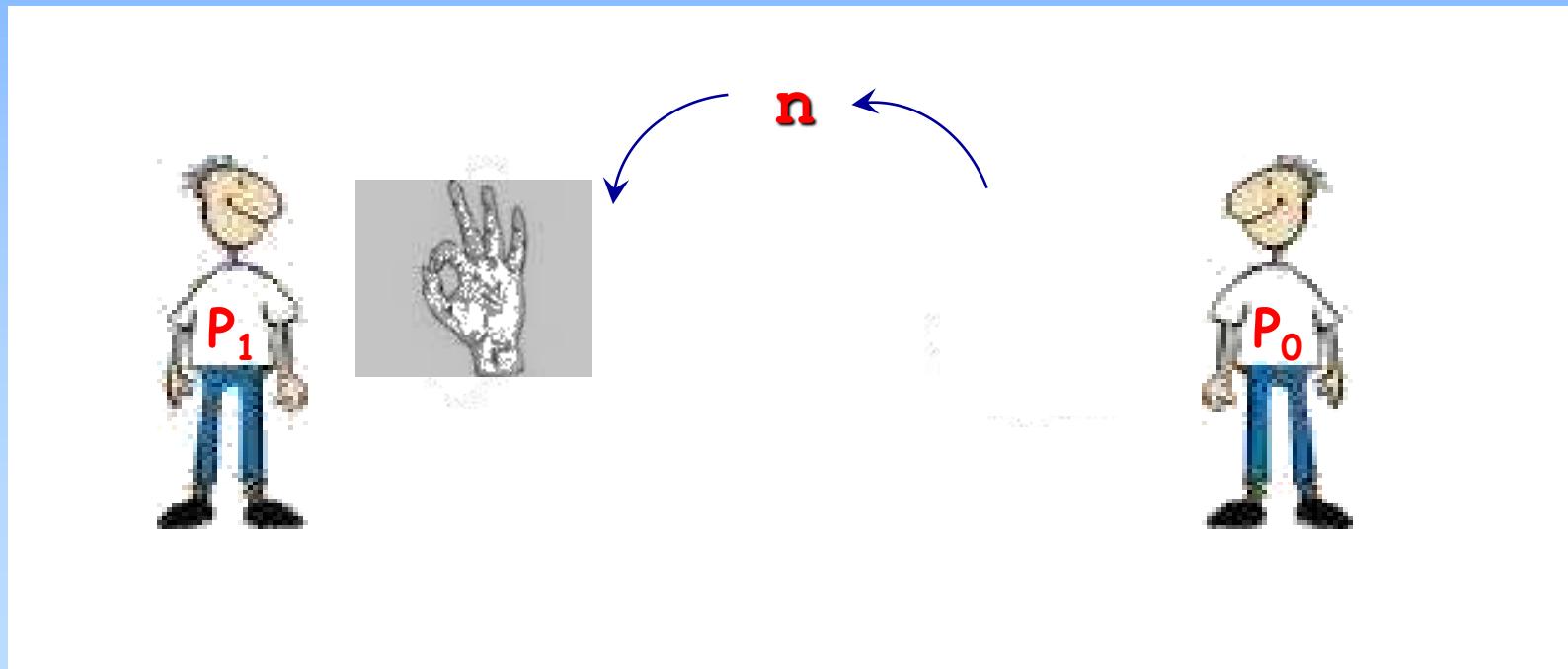
```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{    int menum, nproc;
    int n, tag, num;
    MPI_Status info;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&menum);
    if(menum==0)
    {    scanf("%d",&n);
        tag=10;
        MPI_Send(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD);
    }else { tag=10;
        MPI_Recv(&n,1,MPI_INT,0,tag,MPI_COMM_WORLD,&info);
    }
    MPI_Get_count(&info,MPI_INT,&num);
    MPI_Finalize();
    return 0;
}
```



# Nel programma ... :

**MPI\_Recv(&n, 1, MPI\_INT, 0, tag, MPI\_COMM\_WORLD, &info);**

- Con questa routine il processore chiamante  $P_1$  riceve il parametro **n**, di tipo **MPI\_INT** e di dimensione **1**, dal processore  $P_0$ ; i due processori appartengono entrambi al communicator **MPI\_COMM\_WORLD**. Il parametro **tag** individua univocamente tale spedizione. Il parametro **info**, di tipo **MPI\_Status**, contiene informazioni sulla ricezione del messaggio.



## In generale (comunicazione uno ad uno bloccante) :

```
MPI_Recv(void *buffer, int count,  
        MPI_Datatype datatype, int source,  
        int tag, MPI_Comm comm,  
        MPI_Status *status);
```

- Il processore che esegue questa routine riceve i primi **count** elementi in **buffer**, del tipo **datatype**, dal processore con identificativo **source**.
- L'identificativo **tag** individua univocamente il messaggio in **comm**.
- **status** è un tipo predefinito di MPI che racchiude informazioni sulla ricezione del messaggio.

## In dettaglio...

```
MPI_Recv(void *buffer, int count,  
          MPI_Datatype datatype, int source,  
          int tag, MPI_Comm comm,  
          MPI_Status *status) ;
```

**\*buffer** indirizzo del dato in cui ricevere

**count** numero dei dati da ricevere

**datatype** tipo dei dati da ricevere

**source** identificativo del processore da cui ricevere

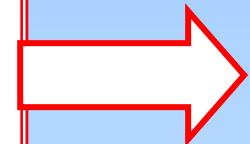
**comm** identificativo del communicator

**tag** identificativo del messaggio

# Un semplice programma con 2 processori:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{    int menum, nproc;
    int n, tag, num;
    MPI_Status info;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&menum);
    if(menum==0)
    {    scanf("%d",&n);
        tag=10;
        MPI_Send(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD);
    }else {    tag=10;
        MPI_Recv(&n,1,MPI_INT,0,tag,MPI_COMM_WORLD,&info);
        MPI_Get_count(&info,MPI_INT,&num);    }
    MPI_Finalize();
    return 0;
}
```



## Nel programma ... :



```
MPI_Get_count(&info,MPI_INT,&num);
```

- num : numero di elementi ricevuti
- Questa routine permette al processore chiamante di conoscere il numero, **num**, di elementi ricevuti, di tipo **MPI\_INT**, nella spedizione individuata da **info**.

## In Generale... :

```
MPI_GET_COUNT(MPI_Status *status  
              MPI_Datatype datatype, int *count);
```

**MPI\_Status** in C è un tipo di dato strutturato, composto da tre campi:

- identificativo del processore da cui ricevere
- identificativo del messaggio
- indicatore di errore

- Il processore che esegue questa routine, memorizza nella variabile **count** il numero di elementi, di tipo **datatype**, che riceve dal messaggio e dal processore indicati nella variabile **status**.

# Comunicazioni non bloccanti in MPI: modalità *Immediate*

Funzione *non bloccante* per la *spedizione* di un messaggio:

**MPI\_Isend**

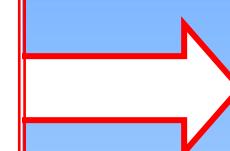
Funzione *non bloccante* per la *ricezione* di un messaggio:

**MPI\_Irecv**

# Un semplice programma con 2 processori:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{ int menum, nproc, n, tag;
  MPI_Request rqst;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&menum);
  if(menum==0)
  { scanf("%d",&n);
    tag=20;
    MPI_Isend(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD,&rqst);
  }else {
    tag=20;
    MPI_Irecv(&n,1,MPI_INT,0,tag,MPI_COMM_WORLD,&rqst);
  }
  MPI_Finalize();
  return 0;
}
```



## Nel programma ... :

 **MPI\_Isend(&n, 1, MPI\_INT, 1, tag, MPI\_COMM\_WORLD, &rqst);**

- Il processore chiamante  $P_0$  spedisce il parametro **n**, di tipo **MPI\_INT** e di dimensione **1**, al processore  $P_1$ ; i due processori appartengono entrambi al communicator **MPI\_COMM\_WORLD**. Il parametro **tag** individua univocamente tale spedizione. Il parametro **rqst** contiene le informazioni dell'intera spedizione. Il processore  $P_0$ , appena inviato il parametro n, è **libero** di procedere nelle successive istruzioni.

## In generale (comunicazione uno ad uno non bloccante) :

```
MPI_ISEND(void *buffer, int count,  
          MPI_Datatype datatype, int dest,  
          int tag, MPI_Comm comm,  
          MPI_Request *request);
```

- Il processore che esegue questa routine spedisce i primi **count** elementi di **buffer**, del tipo **datatype**, al processore con identificativo **dest**.
- L'identificativo **tag** individua univocamente il messaggio in **comm**.
- L'oggetto **request** crea un nesso tra la trasmissione e la ricezione del messaggio

# Un semplice programma con 2 processori:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{   int menum, nproc, n, tag;
    MPI_Request rqst;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD ,&menum) ;
    if(menum==0)
    {   scanf("%d", &n);
        tag=20;
        MPI_Isend(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD ,&rqst) ;
    }else {
        tag=20;
        MPI_Irecv(&n,1,MPI_INT,0,tag,MPI_COMM_WORLD ,&rqst) ;
    }
    MPI_Finalize();
    return 0;
}
```



## Nel programma ... :



```
MPI_Irecv(&n, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &rqst);
```

- Il processore chiamante  $P_1$  riceve il parametro **n**, di tipo **MPI\_INT** e di dimensione **1**, dal processore  $P_0$ ; i due processori appartengono entrambi al communicator **MPI\_COMM\_WORLD**. Il parametro **tag** individua univocamente tale ricezione. Il parametro **rqst** contiene le informazioni dell'intera spedizione. Il processore  $P_1$ , appena ricevuto il parametro n, è **libero** di procedere nelle successive istruzioni.

# Ricezione di un messaggio (comunicazione uno ad uno)

```
MPI_Irecv(void *buffer, int count,  
         MPI_Datatype datatype, int source,  
         int tag, MPI_Comm comm,  
MPI_Request *request);
```

- Il processore che esegue questa routine riceve i primi **count** elementi in **buffer**, del tipo **datatype**, dal processore con identificativo **source**.
- L'identificativo **tag** individua univocamente il messaggio in **comm**.
- L'oggetto **request** crea un nesso tra la trasmissione e la ricezione del messaggio.

# In particolare: request

Le operazioni non bloccanti utilizzano l'oggetto **request** di un tipo predefinito di MPI: **MPI\_Request**.

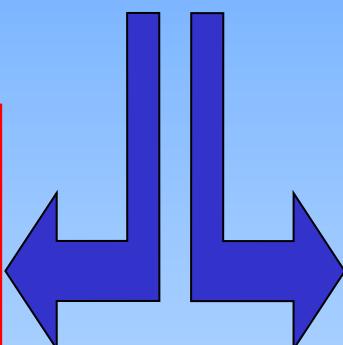


Tale oggetto **collega** l'operazione che **inizia** la comunicazione in esame con l'operazione che la **termina**.

# Osservazione

L'oggetto **request** ha nelle comunicazioni, un ruolo simile a quello di **status**.

**status**  
contiene informazioni  
sulla **ricezione**  
del messaggio.

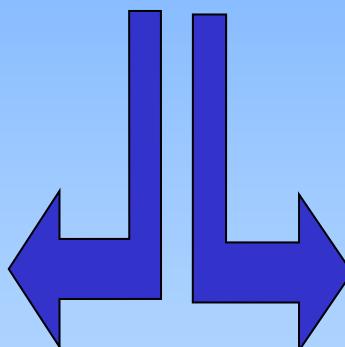


**request**  
contiene informazioni  
su tutta la fase di  
trasmissione o di  
**ricezione**  
del messaggio.

# Osservazione: termine di un'operazione non bloccante

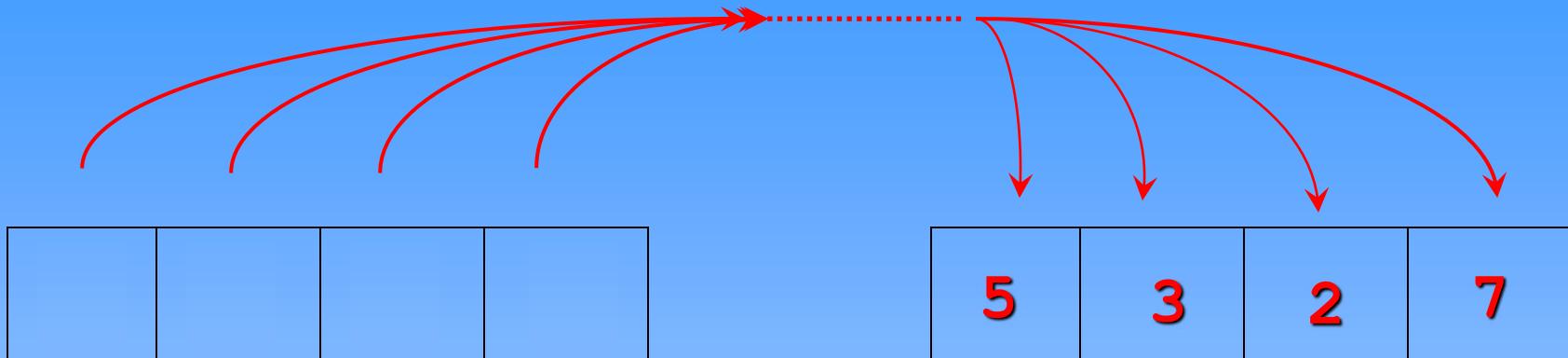
Un'operazione non bloccante  
è terminata  
quando:

Nel caso della  
**spedizione**, quando il  
**buffer** è nuovamente  
riutilizzabile dal  
programma.



Nel caso della  
**ricezione** quando il  
messaggio è stato  
interamente  
ricevuto.

# Osservazione: termine di un'operazione non bloccante



Vettore x da spedire

Operazione di  
spedizione  
non bloccante  
terminata

Vettore x da ricevere

Operazione di  
ricezione  
non bloccante  
terminata

# Domanda

Utilizzando una comunicazione non bloccante  
*come si fa a sapere*  
se l'operazione di spedizione o di ricezione  
**è stata terminata**

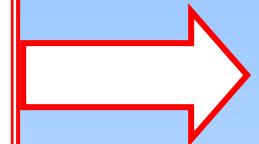
?

# Risposta

MPI mette a disposizione  
delle funzioni per  
*controllare tutta la fase*  
di trasmissione di un messaggio  
mediante comunicazione  
non bloccante.

# Controllo sullo stato di un'operazione non bloccante...

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{ int flag,nloc;
...
nloc=1;
if(menum==0)
{ scanf("%d",&n);
  tag=20;
  MPI_Isend(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD,&rqst) ;
}
if(menum!=0)
{
  tag=20;
  MPI_Irecv(&n,1,MPI_INT,0,tag,MPI_COMM_WORLD,&rqst) ;
MPI_Test(&rqst,&flag,&info);
  if(flag==1){
    nloc+=n;}else{
    MPI_Wait(&rqst,&info);
    nloc+=n;}
  printf("nloc=%d \n",nloc);
}
...
}
```



Nel programma ... :



**MPI\_Test(&rqst,&flag,&info);**

- Il processore chiamante  $P_1$  verifica se la ricezione di **n**, individuata da **rqst**, è stata completata; in questo caso **flag=1** e procede all'incremento di nloc.
- Altrimenti **flag=0**.

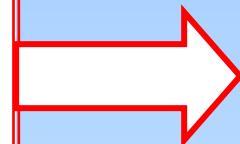
# Controllo sullo stato di un'operazione non bloccante...

```
MPI_Test(MPI_Request *request,  
         int *flag, MPI_Status *status);
```

- Il processore che esegue questa routine testa lo stato della comunicazione non bloccante, identificata da **request**.
- La funzione **MPI\_Test** ritorna l'intero **flag**:  
  
**flag = 1**, l'operazione identificata da **request** è terminata;  
  
**flag = 0**, l'operazione identificata da **request** NON è terminata;

# Controllo sullo stato di un'operazione non bloccante...

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{   int flag,nloc;
...
nloc=1;
if(menum==0)
{ scanf("%d", &n);
  tag=20;
  MPI_Isend(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD,&rqst) ;
}
if(menum!=0)
{
  tag=20;
  MPI_Irecv(&n,1,MPI_INT,0,tag,MPI_COMM_WORLD,&rqst) ;
  MPI_Test(&rqst,&flag,&info) ;
  if(flag==1){
    nloc+=n;}else{
MPI_Wait(&rqst,&info);
  nloc+=n;}
  printf("nloc=%d \n",nloc);
}
...
}
```



Nel programma ... :



**MPI\_Wait(&rqst,&info);**

- Il processore chiamante  $P_1$  procede nelle istruzioni solo quando la ricezione di **n** è stata completata.

## In generale... :

```
MPI_Wait(MPI_Request *request,  
          MPI_Status *status);
```

- Il processore che esegue questa routine controlla lo stato della comunicazione non bloccante, identificata da **request**, e si arresta solo quando l'operazione in esame si è conclusa. In **status** si hanno informazioni sul completamento dell'operazione di Wait.

# Vantaggi delle operazioni non bloccanti

Le comunicazioni di tipo non bloccante hanno  
**DUE** vantaggi:

- 1) Il processore non è obbligato ad **aspettare** in stato di attesa.

```
...
if (menum==0)
{
    scanf("%d", &n);
    tag=20;
    MPI_Isend(&n, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &rqst);
}
/* P0 può procedere nelle operazioni senza dover
   attendere il risultato della spedizione di n
   al processore P1 */
if (menum!=0)
{
    ...
}
...
}
```

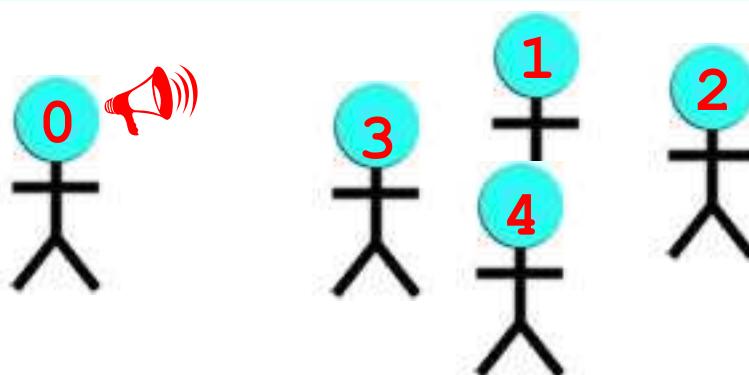
# Vantaggi delle operazioni non bloccanti

2) Più comunicazioni possono avere luogo **contemporaneamente** sovrapponendosi almeno in parte tra loro.

```
...
if (menum==0)
{
    ...
    /* P0 spedisce due elementi a P1 */
    MPI_Isend(&a,1,MPI_INT,1,0,MPI_COMM_WORLID,&rqst1);
    MPI_Isend(&b,1,MPI_INT,1,0,MPI_COMM_WORLD,&rqst2);
} elseif (menum==1) {
    /* P1 riceve due elementi da P0
       secondo l'ordine di spedizione*/
    MPI_Irecv(&a,1,MPI_INT,0,0,MPI_COMM_WORLD,&rqst1);
    MPI_Irecv(&b,1,MPI_INT,0,0,MPI_COMM_WORLD,&rqst2);
}
...
...
```

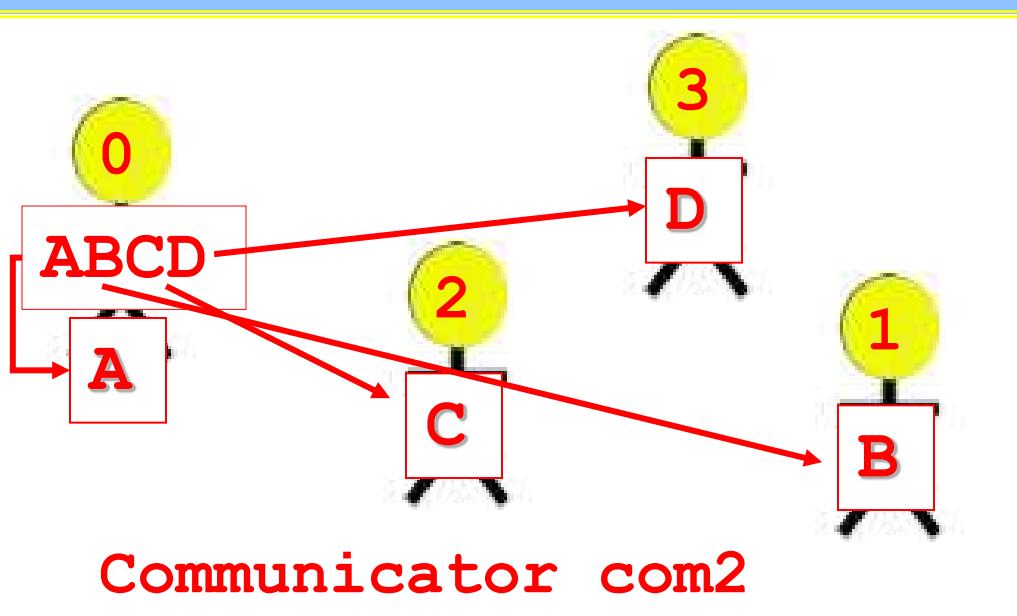
# Le operazioni collettive.

# Esempi di operazioni collettive :



Communicator **com1**

Nel connunicator  
**com1**  $P_0$  comunica  
con tutti gli altri  
processori



Communicator **com2**

$P_0$  distribuisce  
a tutti i  
processori di  
**com2** un  
elemento del  
proprio vettore

# Caratteristiche delle operazioni collettive

Le operazioni collettive sono eseguite da **tutti** i processori appartenenti ad un communicator.

Inoltre...

- Tutti i processori che eseguono l'operazione collettiva eseguono almeno **una** comunicazione.
- L'operazione collettiva può richiedere una **sincronizzazione**.
- Tutte le operazioni collettive sono **bloccanti**.

# Scopo delle operazioni collettive

Le operazioni collettive permettono:

- La **Sincronizzazione** dei processori.
- L'esecuzione di **operazioni globali** (es. ricerca del massimo in un vettore distribuito fra i processori).
- Gestione **ottimizzata** degli input/output seguendo uno schema ad albero.

# Sincronizzazione dei processori ... :

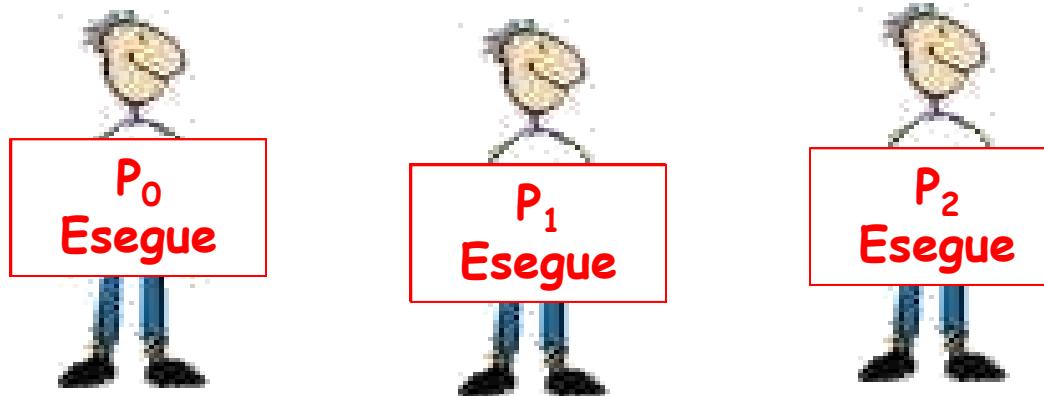
```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{
    ...
    MPI_Init(&argc, &argv);
    ...
    MPI_Barrier(MPI_COMM_WORLD);
    ...
    MPI_Finalize();
}
```



# Nel programma ... :

**MPI\_Barrier(MPI\_COMM\_WORLD);**

- Ogni processore dell'ambiente ***MPI\_COMM\_WORLD*** può procedere solo quando tutti gli altri avranno richiamato questa routine.



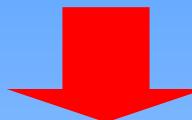
## In generale ... :

**MPI\_Barrier(MPI\_Comm comm) ;**

- Questa routine fornisce un meccanismo sincronizzante per **tutti** i processori del communicator **comm**.
- Ogni processore si ferma fin quando tutti i processori di **comm** non eseguono **MPI\_Barrier**.

# La comunicazione collettiva di un messaggio

La comunicazione  
di un messaggio può coinvolgere  
due o più processori.



Per comunicazioni che  
coinvolgono  
solo **due** processori



Si considerano  
funzioni MPI per  
**comunicazioni uno a uno**

Per comunicazioni che  
coinvolgono  
**più** processori



Si considerano  
funzioni MPI per  
**comunicazioni collettive**

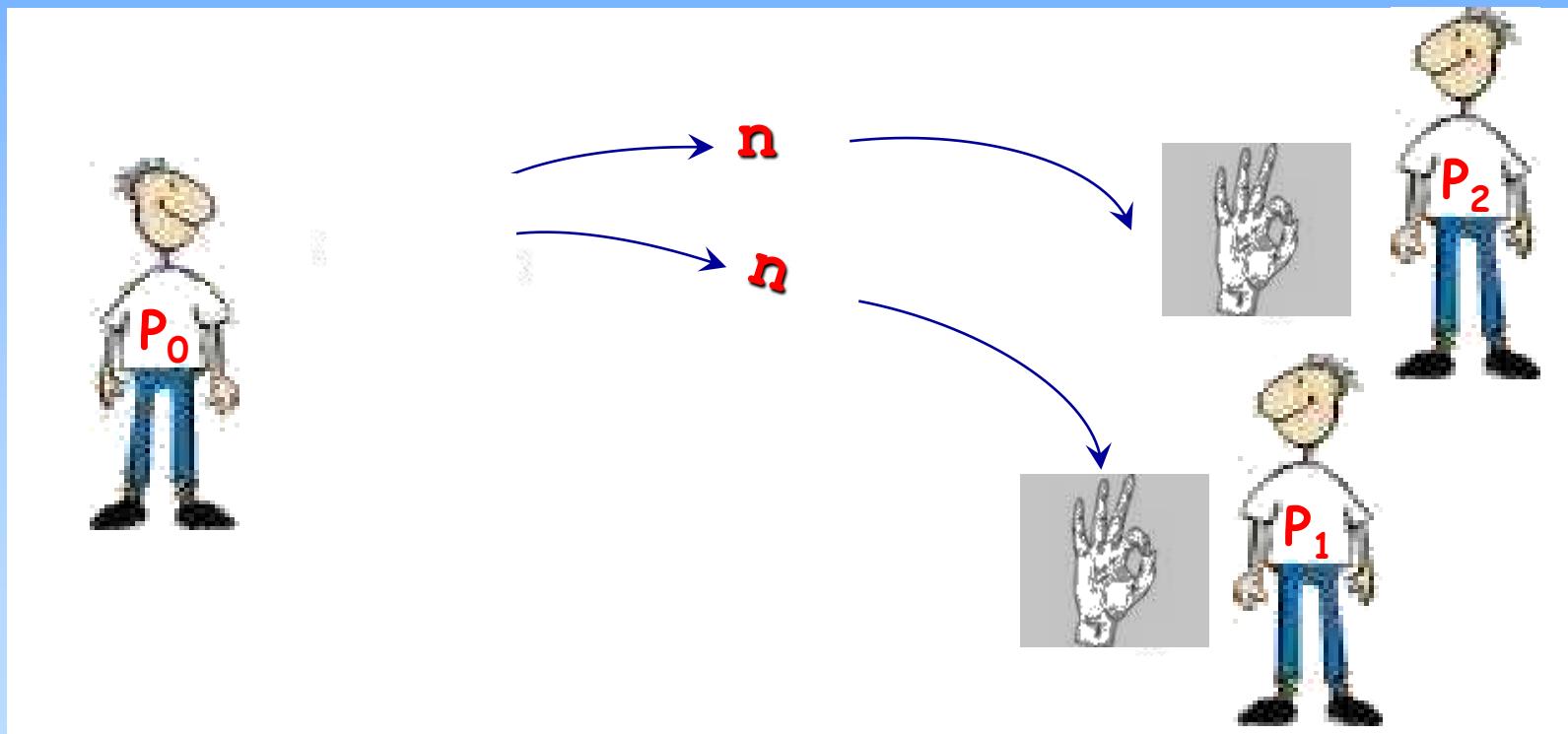
```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{    int menum, nproc;
    int n, tag, num;
    MPI_Status info;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&menum) ;
    if(menum==0)
    {
        scanf ("%d", &n) ;
    }
    MPI_Bcast (&n,1,MPI_INT,0,MPI_COMM_WORLD) ;
    MPI_Finalize();
    return 0;
}
```



Nel programma ... :

**→ MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);**

- Il processore  $P_0$  spedisce  $n$ , di tipo  $\text{MPI\_INT}$  e di dimensione 1, a tutti i processori dell'ambiente  $\text{MPI\_COMM\_WORLD}$ .



```
MPI_Bcast(void *buffer, int count,  
          MPI_Datatype datatype,  
          int root, MPI_Comm comm);
```

- Il processore con identificativo **root** spedisce a tutti i processori del comunicator **comm** lo stesso dato memorizzato in **\*buffer**.
- **Count, datatype, comm** devono essere uguali per ogni processore di **comm**.

## In dettaglio...:

```
MPI_Bcast(void *buffer, int count,  
           MPI_Datatype datatype,  
           int root, MPI_Comm comm);
```

**\*buffer** indirizzo del dato da spedire

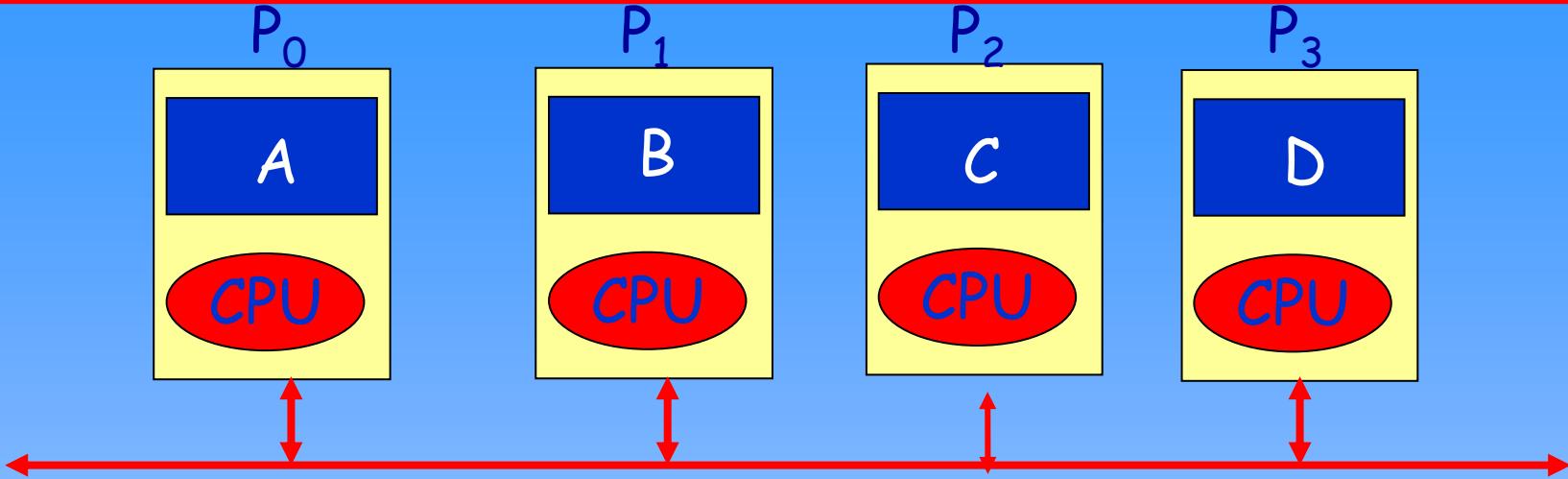
**count** numero dei dati da spedire

**datatype** tipo dei dati da spedire

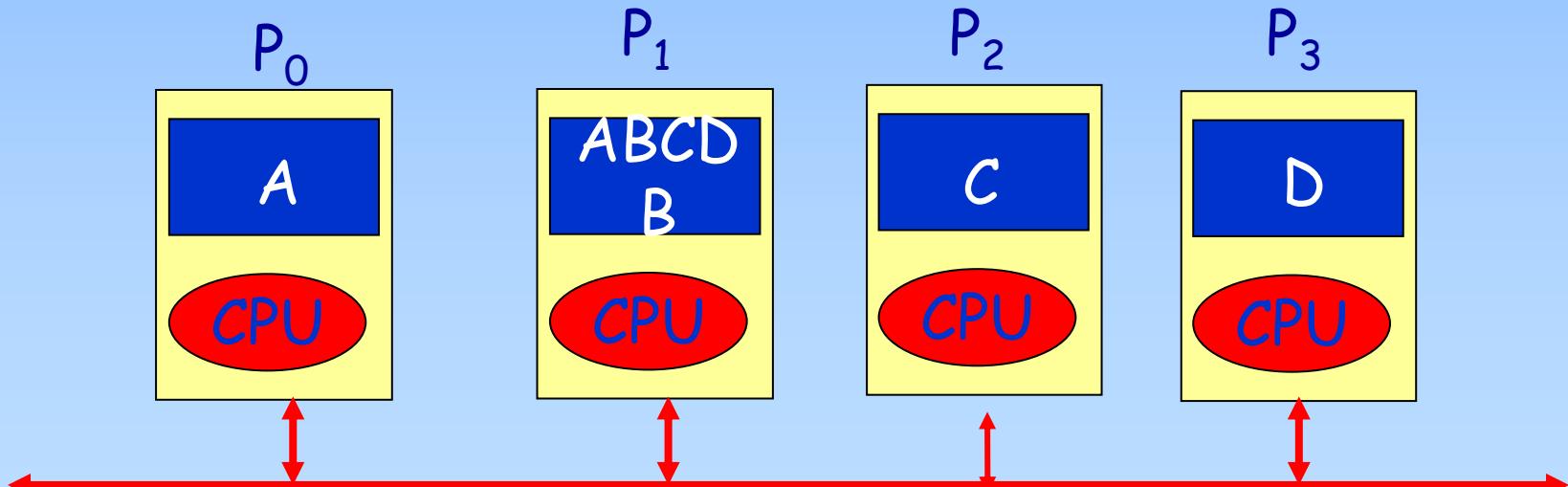
**root** identificativo del processore che spedisce a tutti

**comm** identificativo del communicator

# Operazione collettiva di tipo: *data gather*



Tutti i processori spediscono i propri dati ad un processore assegnato (es al processore P<sub>1</sub>)



```
MPI_Gather(void *send_buff, int send_count,  
         MPI_Datatype datatype,  
         void *recv_buff, int recv_count,  
         MPI_Datatype recv_type,  
         int root, MPI_Comm comm) ;
```

- Ogni processore di **comm** spedisce il contenuto di **\*send\_buff** al processore con identificativo **root**
- Il processore con identificativo **root** concatena i dati ricevuti in **recv\_buff**, memorizzando prima i dati ricevuti dal processore 0, poi i dati ricevuti dal processore 1, poi quelli ricevuti dal processore 2, etc...

```
MPI_Gather(void *send_buff, int send_count,  
          MPI_Datatype datatype,  
          void *recv_buff, int recv_count,  
          MPI_Datatype recv_type,  
          int root, MPI_Comm comm) ;
```

- Gli argomenti **recv\_** sono significativi solo per il processore **root**
- L'argomento **recv\_count** è il numero di dati da ricevere da ogni processore, non è il numero totale dei dati da ricevere.

```
MPI_Gather(void *send_buff, int send_count,  
         MPI_Datatype sendtype,  
         void *recv_buff, int recv_count,  
         MPI_Datatype recv_type,  
         int root, MPI_Comm comm) ;
```

**\*send\_buff** indirizzo del dato da spedire

**send\_count** numero dei dati da spedire

**sendtype** tipo dei dati da spedire

**\*recv\_buff** indirizzo del dato in cui **root** riceve

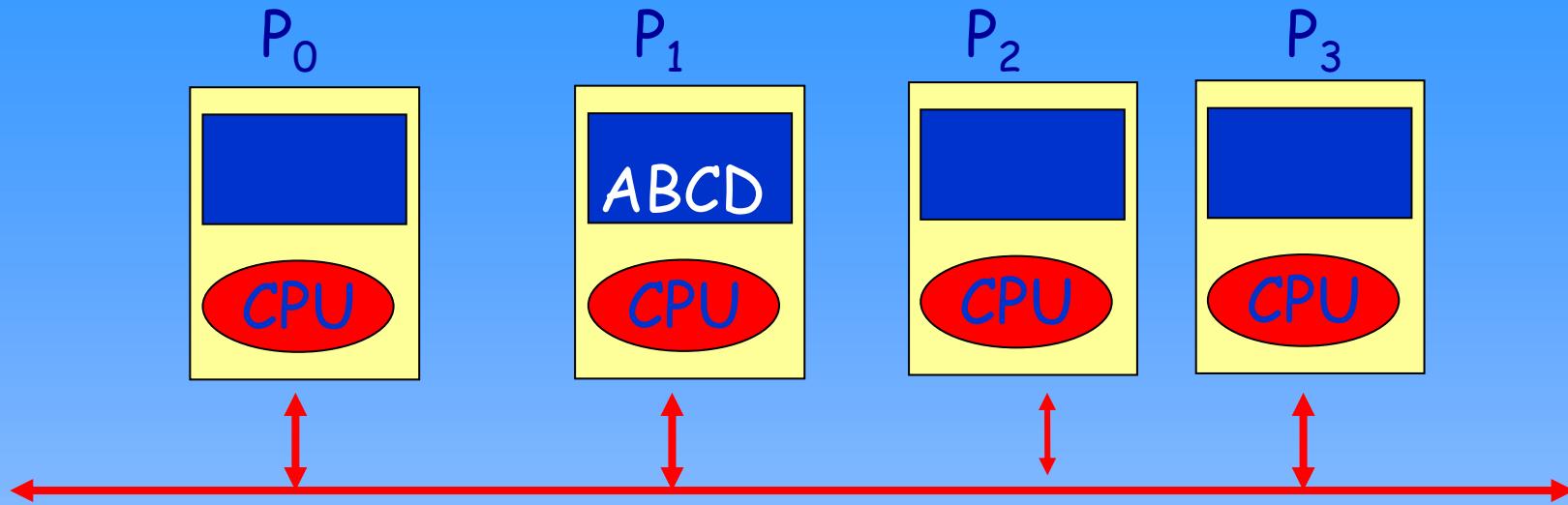
**recv\_count** numero dei dati che root riceve

**recv\_type** tipo dei dati che root riceve

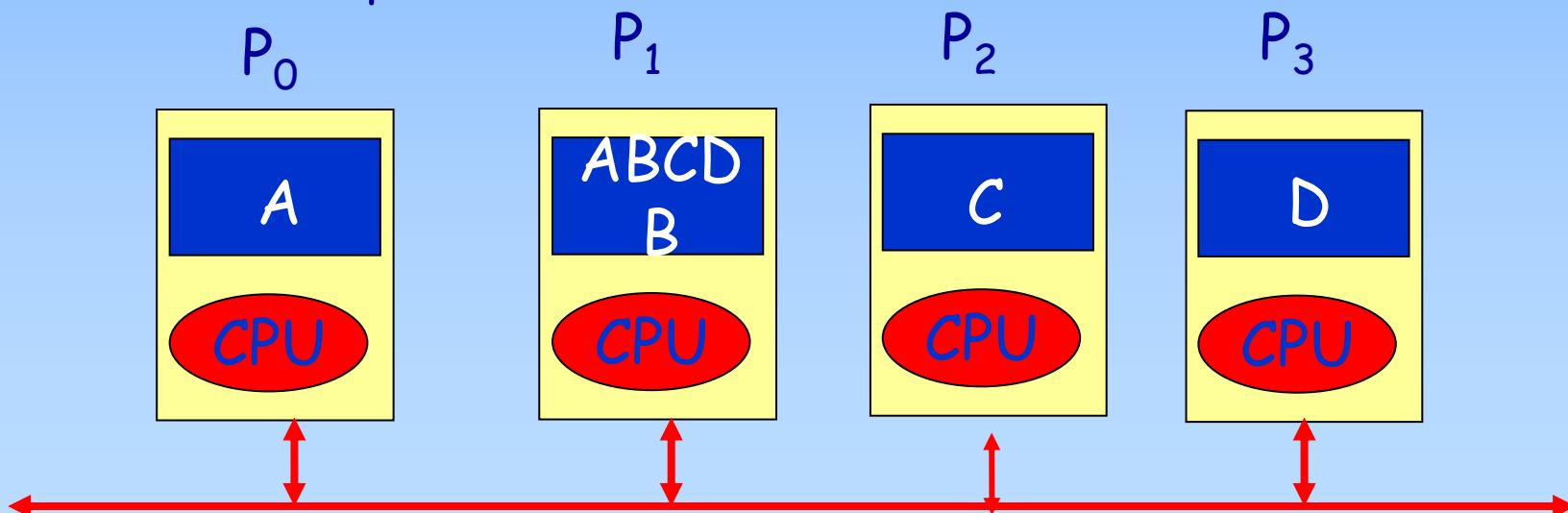
**root** identificativo del processore che riceve da tutti

**comm** identificativo del communicator

# Operazione collettiva di tipo: *data scatter*



Un solo processore distribuisce i propri dati agli altri processori, se stesso compreso.



```
MPI_Scatter(void *send_buff, int send_count,  
MPI_Datatype send_type,  
void *recv_buff, int recv_count,  
MPI_Datatype recv_type,  
int root, MPI_Comm comm);
```

- Il processore con identificativo **root** distribuisce i dati contenuti in **send\_buff**.
- Il contenuto di **send\_buff** viene suddiviso in **nproc** segmenti ciascuno di lunghezza **send\_count**
- Il primo segmento viene affidato al processore con identificativo 0, il secondo al processore con identificativo 1, il terzo al processore con identificativo 2, etc.

```
MPI_Scatter(void *send_buff, int send_count,  
         MPI_Datatype send_type,  
         void *recv_buff, int recv_count,  
         MPI_Datatype recv_type,  
         int root, MPI_Comm comm) ;
```

**\*send\_buff** indirizzo del dato da spedire

**send\_count** numero dei dati da spedire

**send\_type** tipo dei dati da spedire

**\*recv\_buff** indirizzo del dato in cui ricevere

**recv\_count** numero dei dati da ricevere

**recv\_type** tipo dei dati da ricevere

**root** identificativo del processore che spedisce a tutti

**comm** identificativo del communicator

---

# **FINE LEZIONE**

# OpenMp

Introduzione agli strumenti

Prof. G. Laccetti

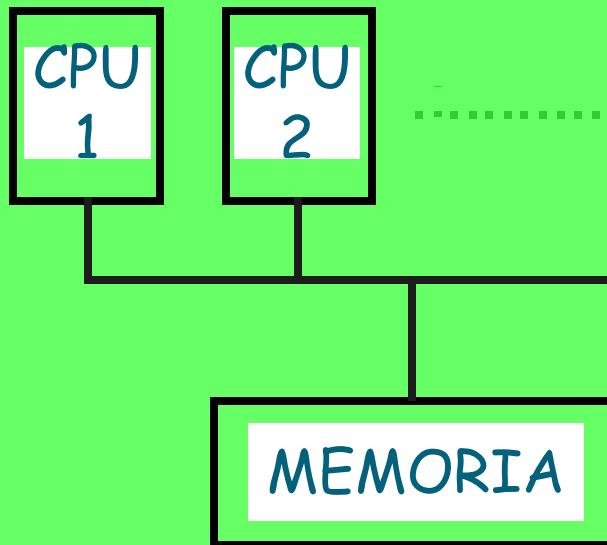
# OpenMP – Introduzione

- Materiale tratto da lezioni di *Calcolo Parallelo e Distribuito* tenute da A. Murli e dal testo  
A. Murli – Lezioni di Calcolo Parallelo, Liguori

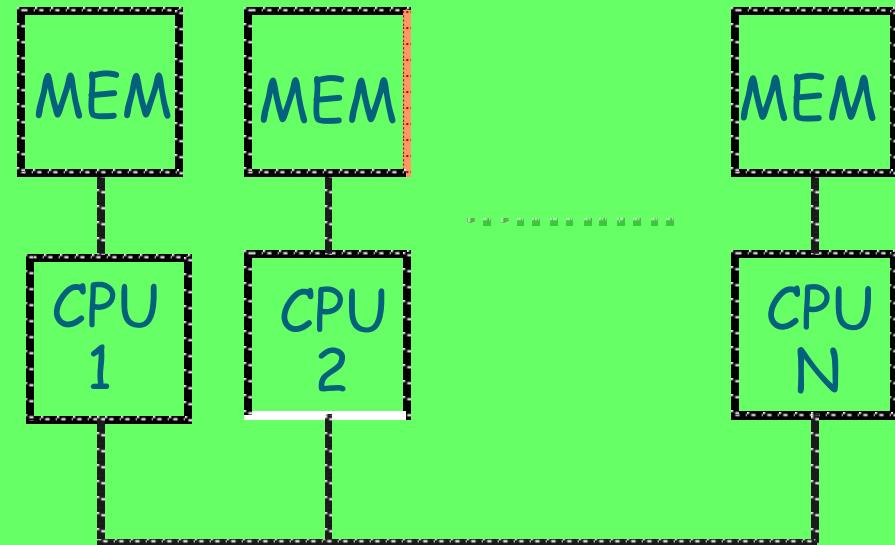


# Shared vs Distributed

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



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



# Shared vs Distributed

Calcolatori MIMD a memoria **condivisa**

Possibilità di utilizzare i dati senza distribuirli

MEMORIA

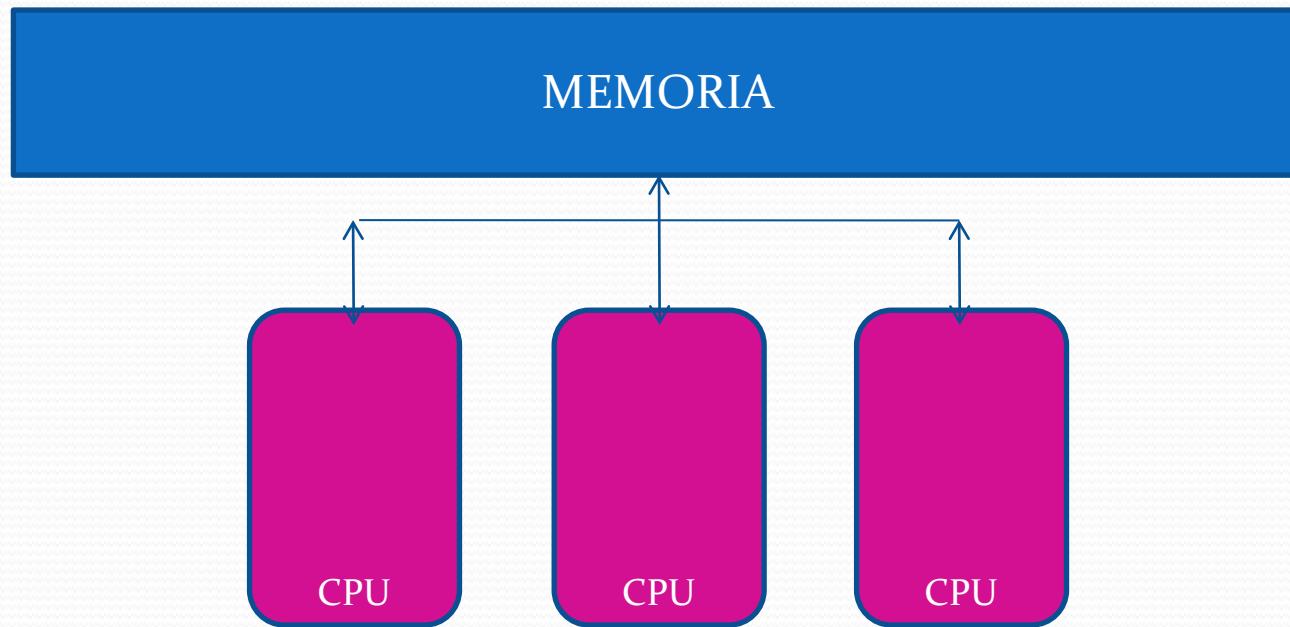
Calcolatori MIMD a memoria **distribuita**

Necessità di organizzare le comunicazioni tra i processi



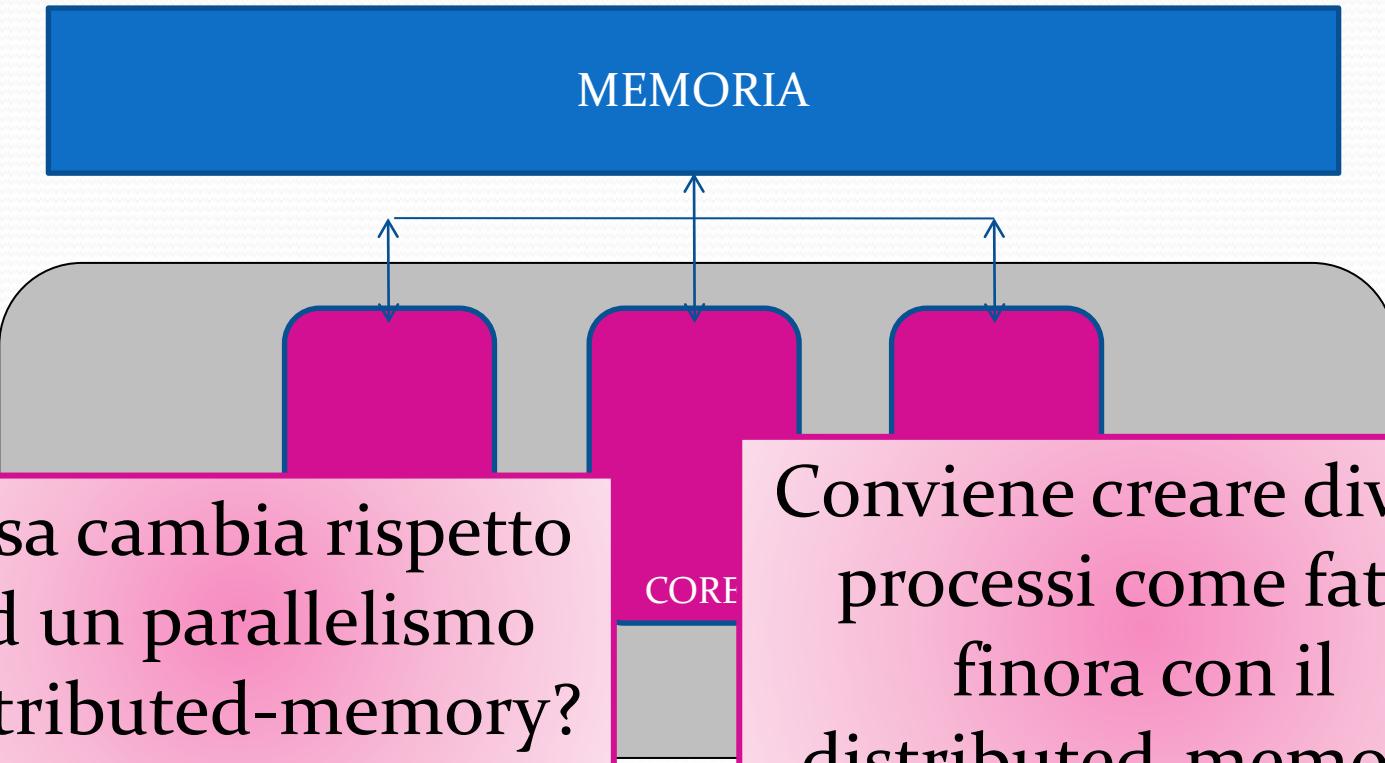
# Shared Memory:

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

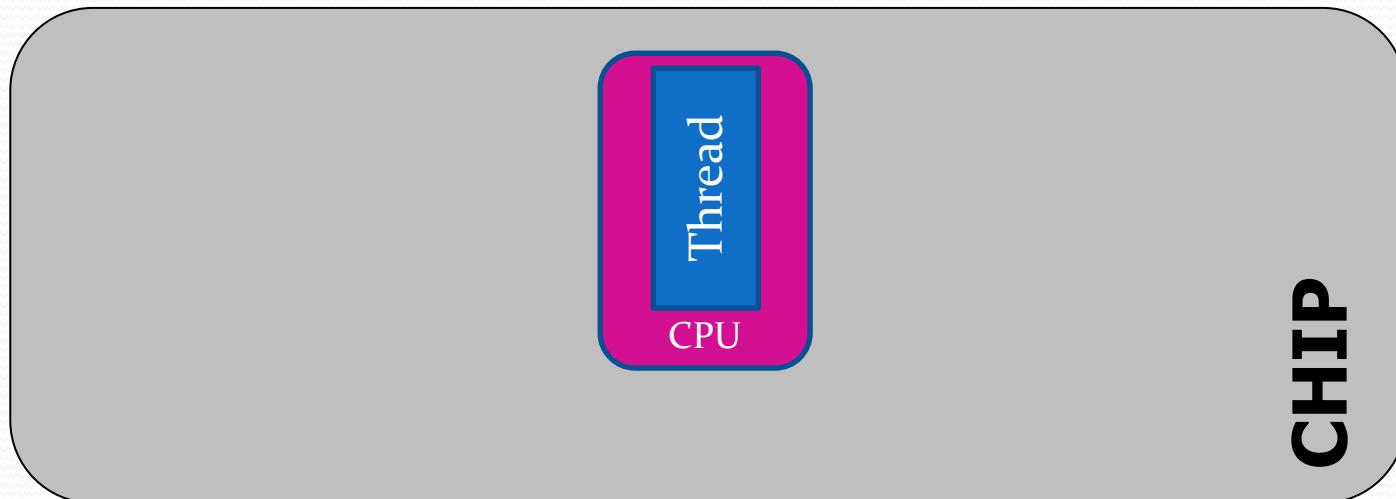
Se si tratta di un sistema *multicore* la situazione non è diversa



# Threads

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

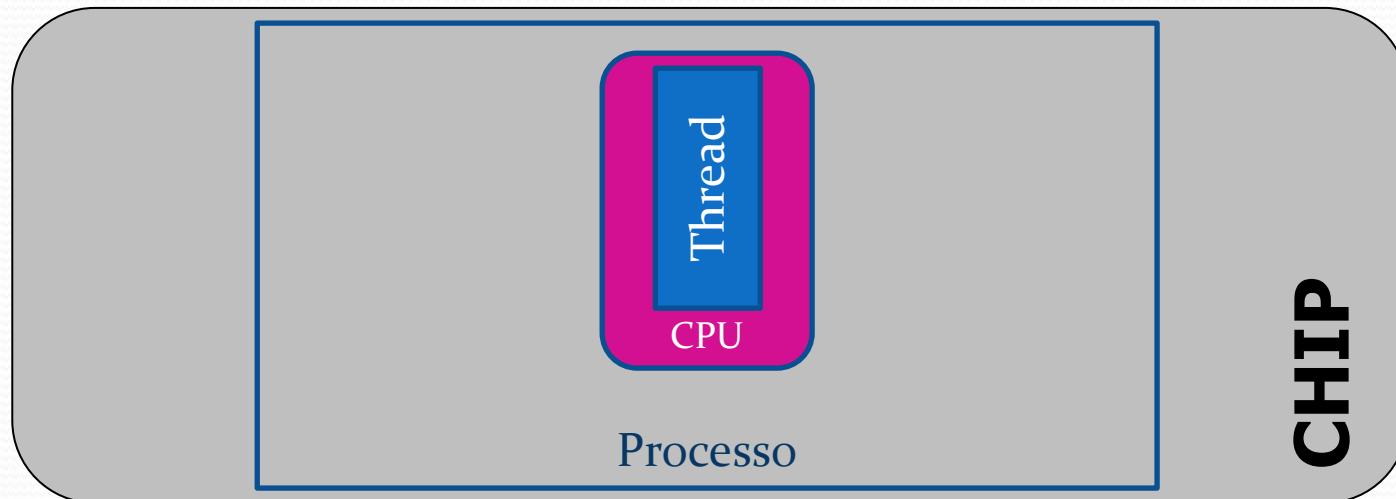
Il thread è quindi un flusso di istruzioni indipendente da altri che deve essere eseguito sequenzialmente su una CPU



# Threads vs Processi

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

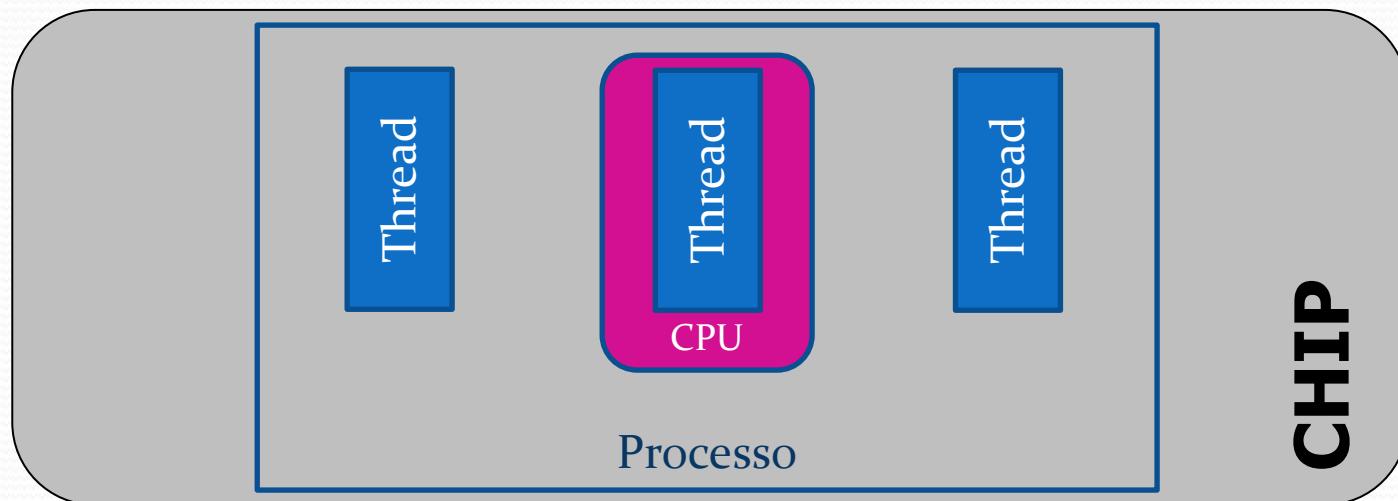
Un **processo** è costituito da almeno un **thread** ...



# Threads vs Processi

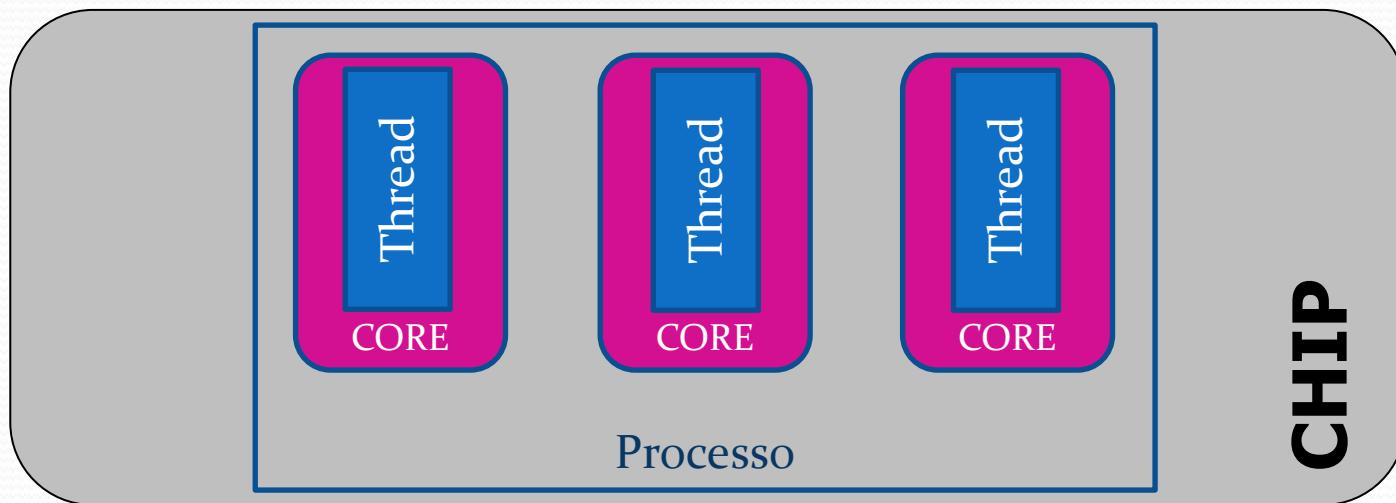
Un **processo** si definisce banalmente come  
“un programma in esecuzione”.

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



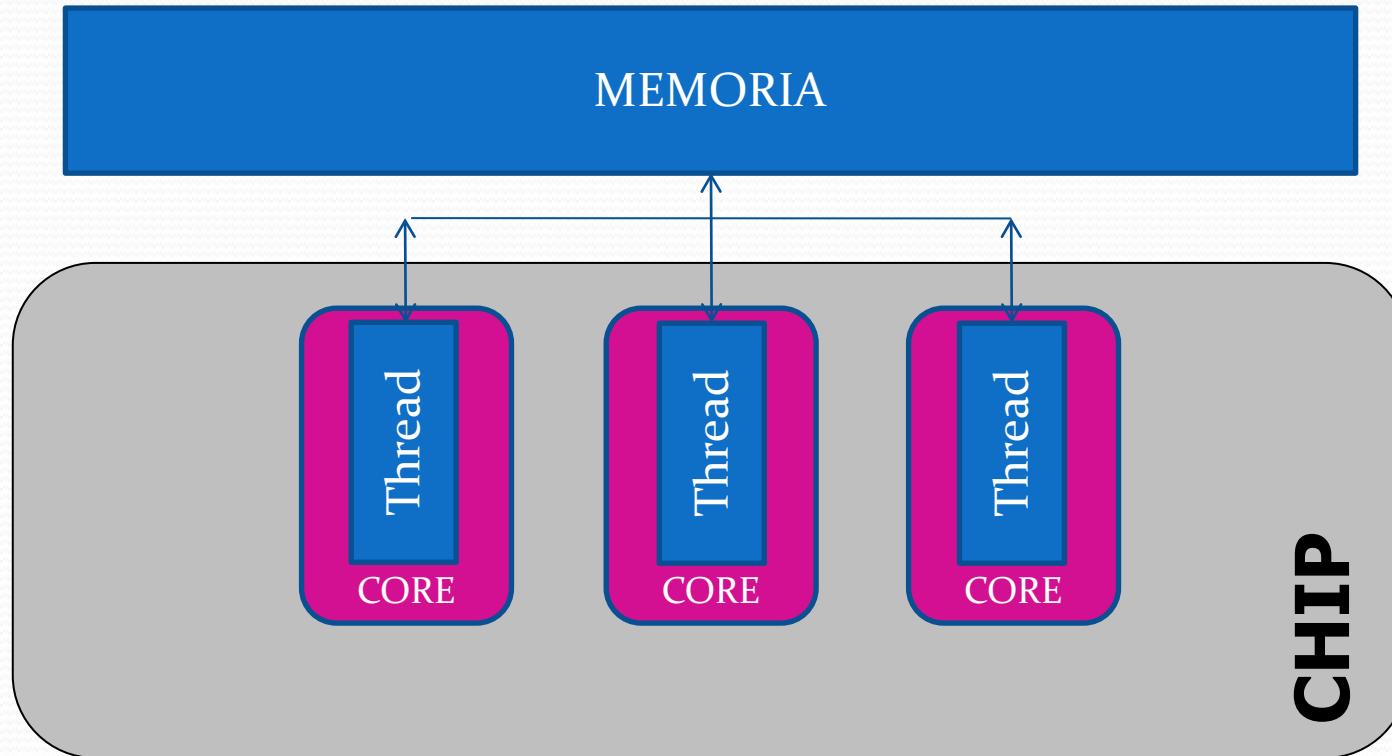
# Threads vs Processi

Thread diversi possono essere eseguiti indipendentemente su cpu/core diversi



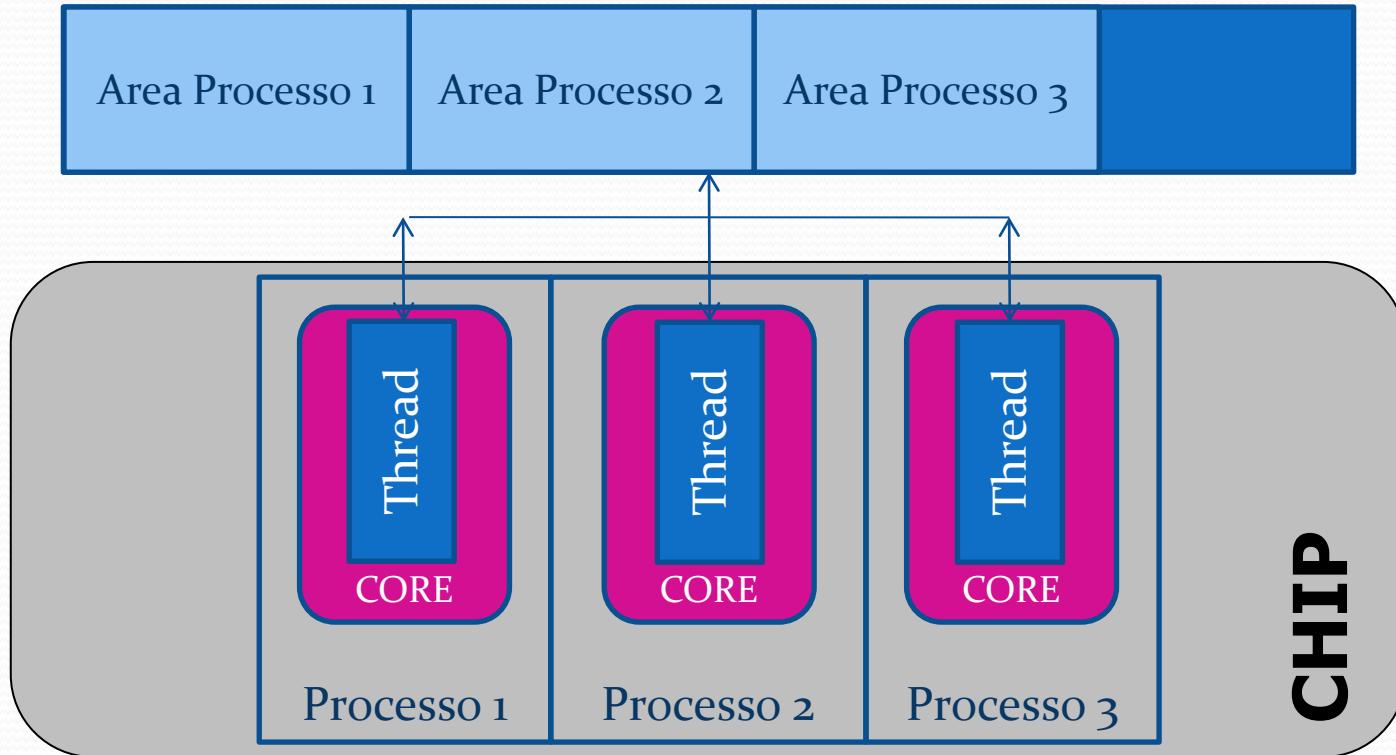
# Threads vs Processi

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



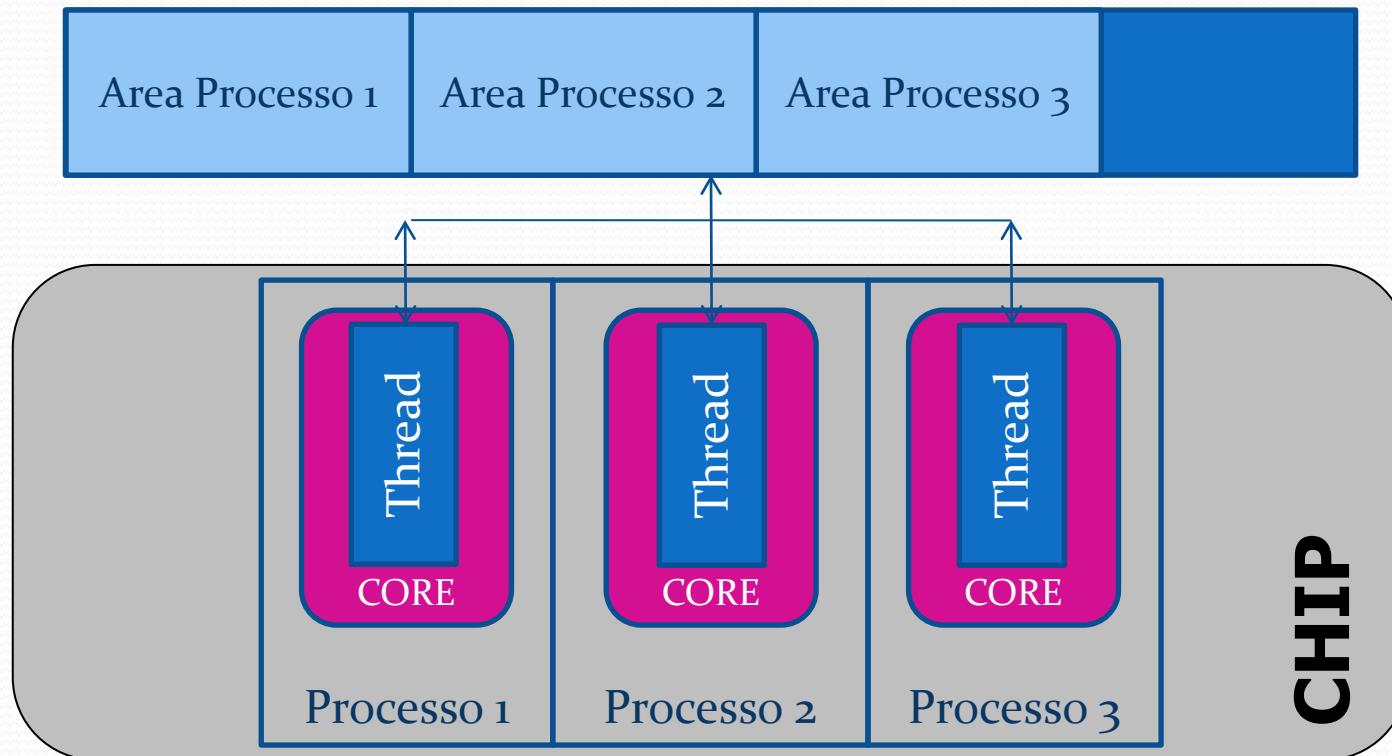
# Threads vs Processi

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



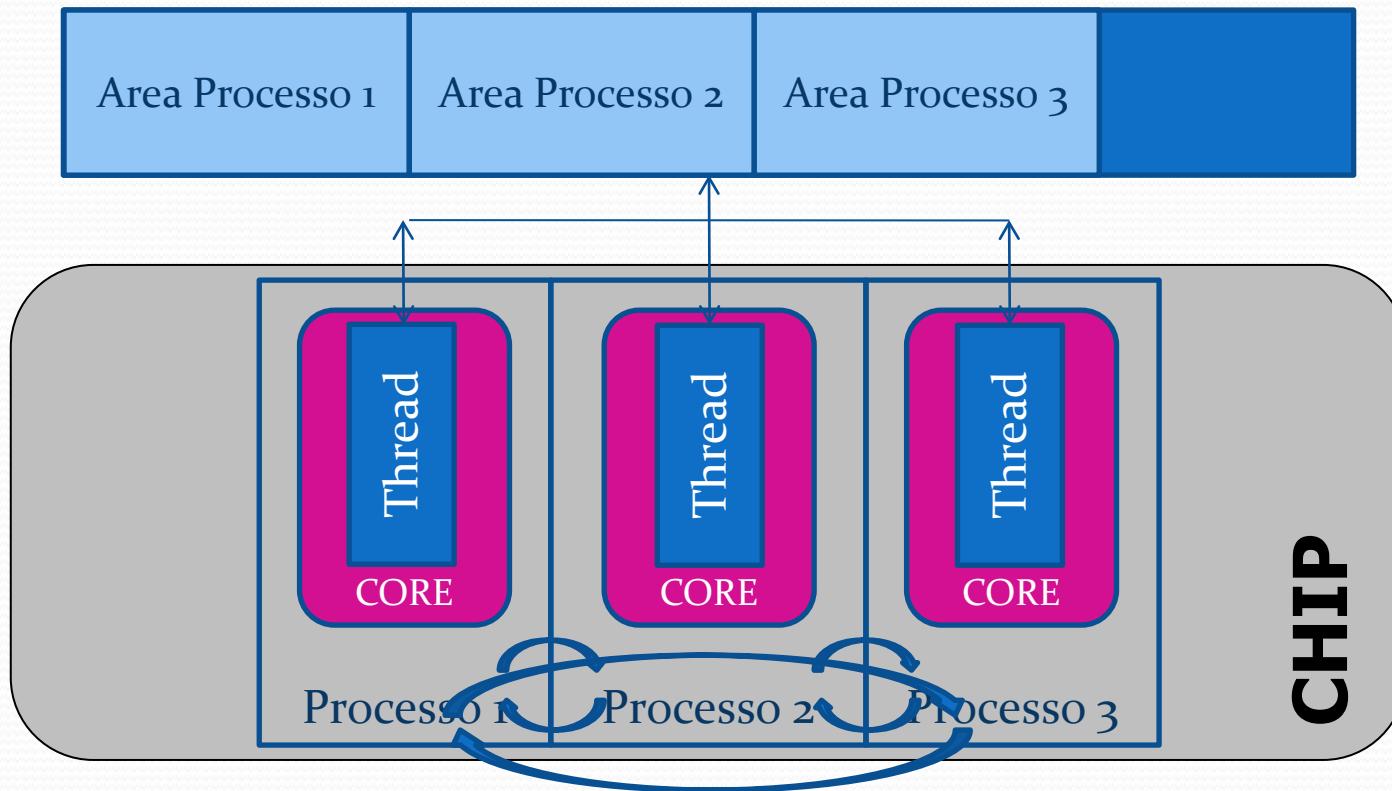
# Threads vs Processi

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



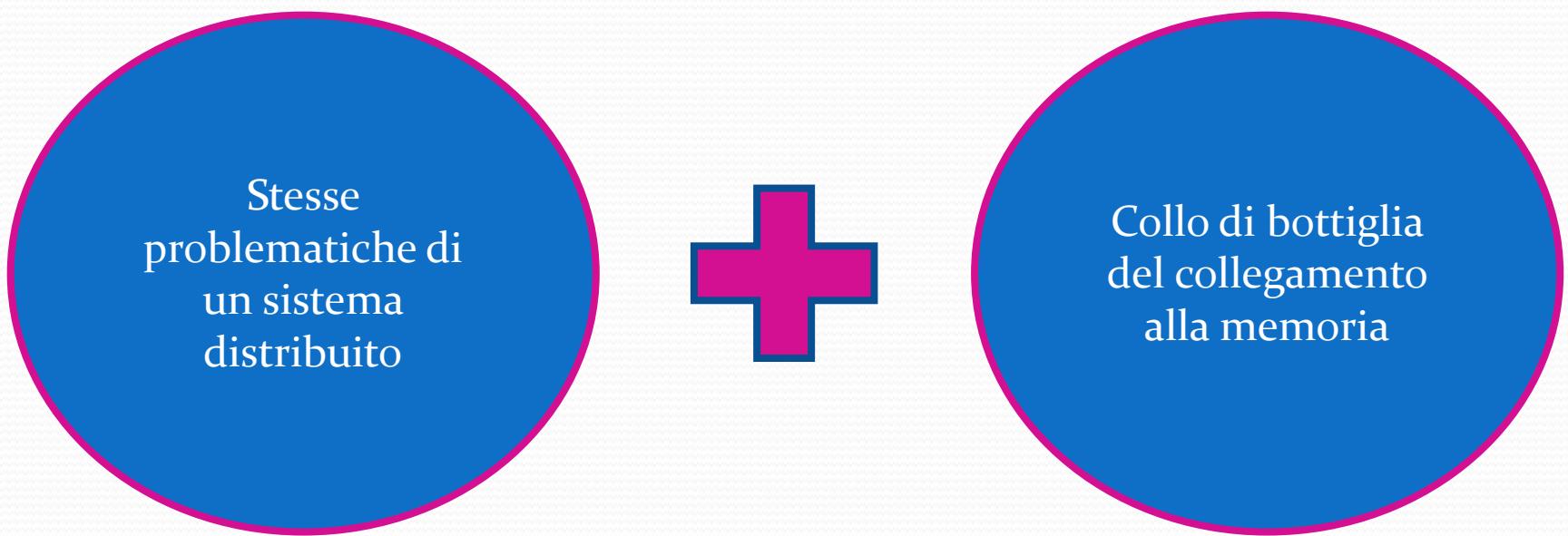
# Threads vs Processi

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**



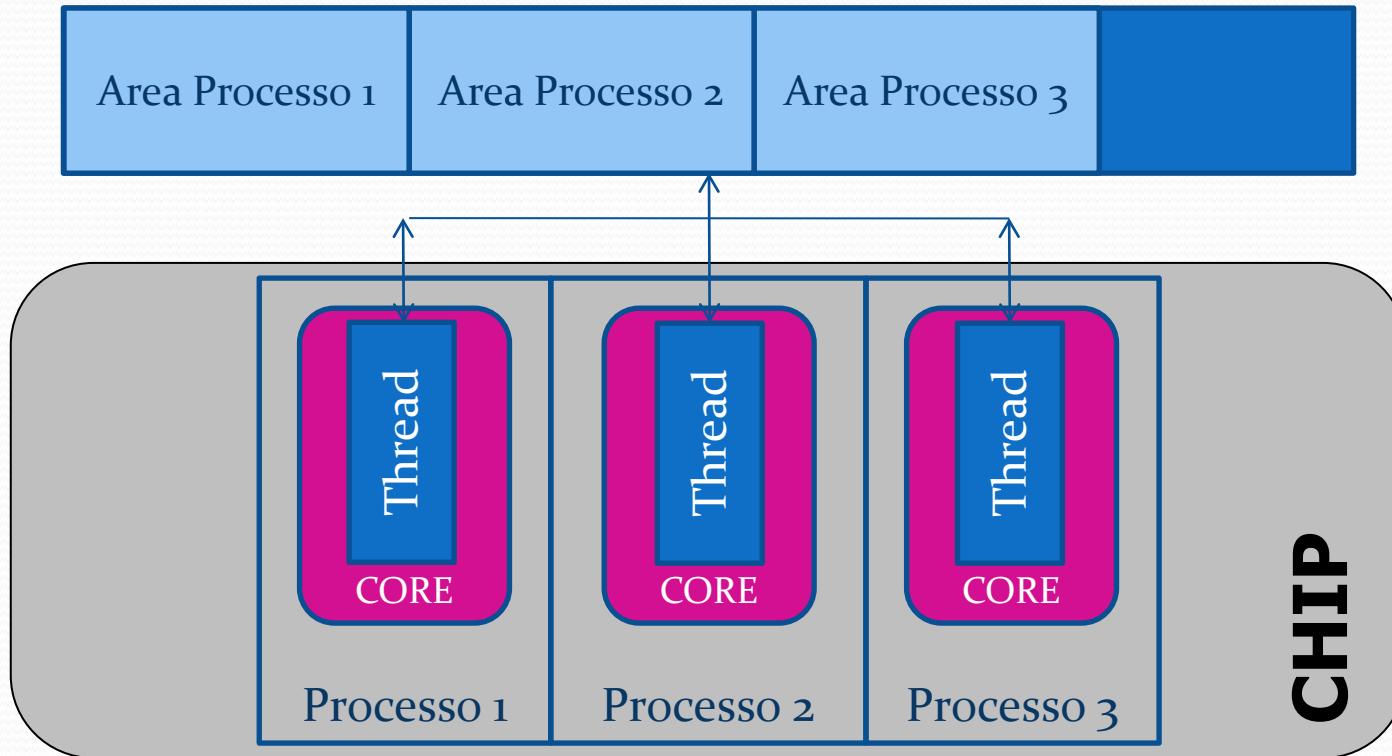
Stesse problematiche di un sistema distribuito

Collo di bottiglia del collegamento alla memoria

# Threads vs Processi

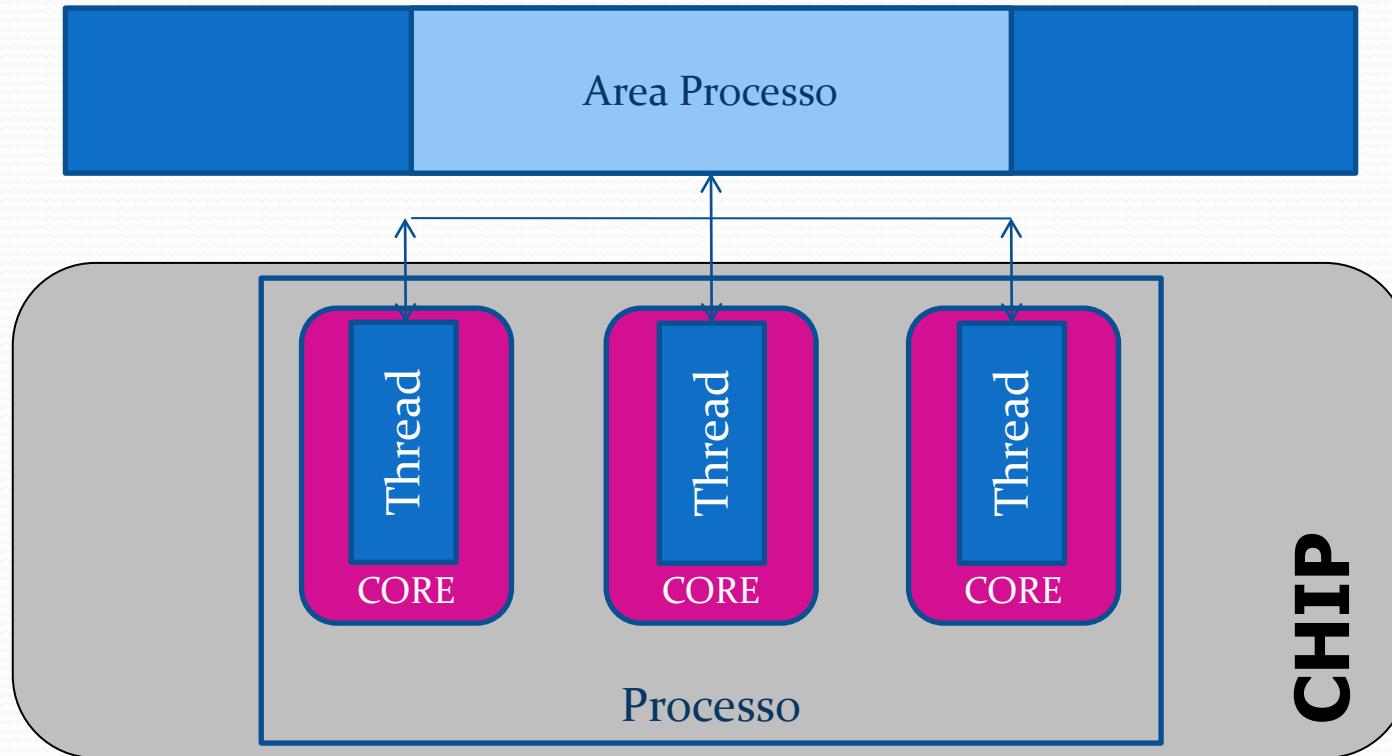
Vantaggio: protezione dei dati

Svantaggio: gestione pesante e poco efficiente



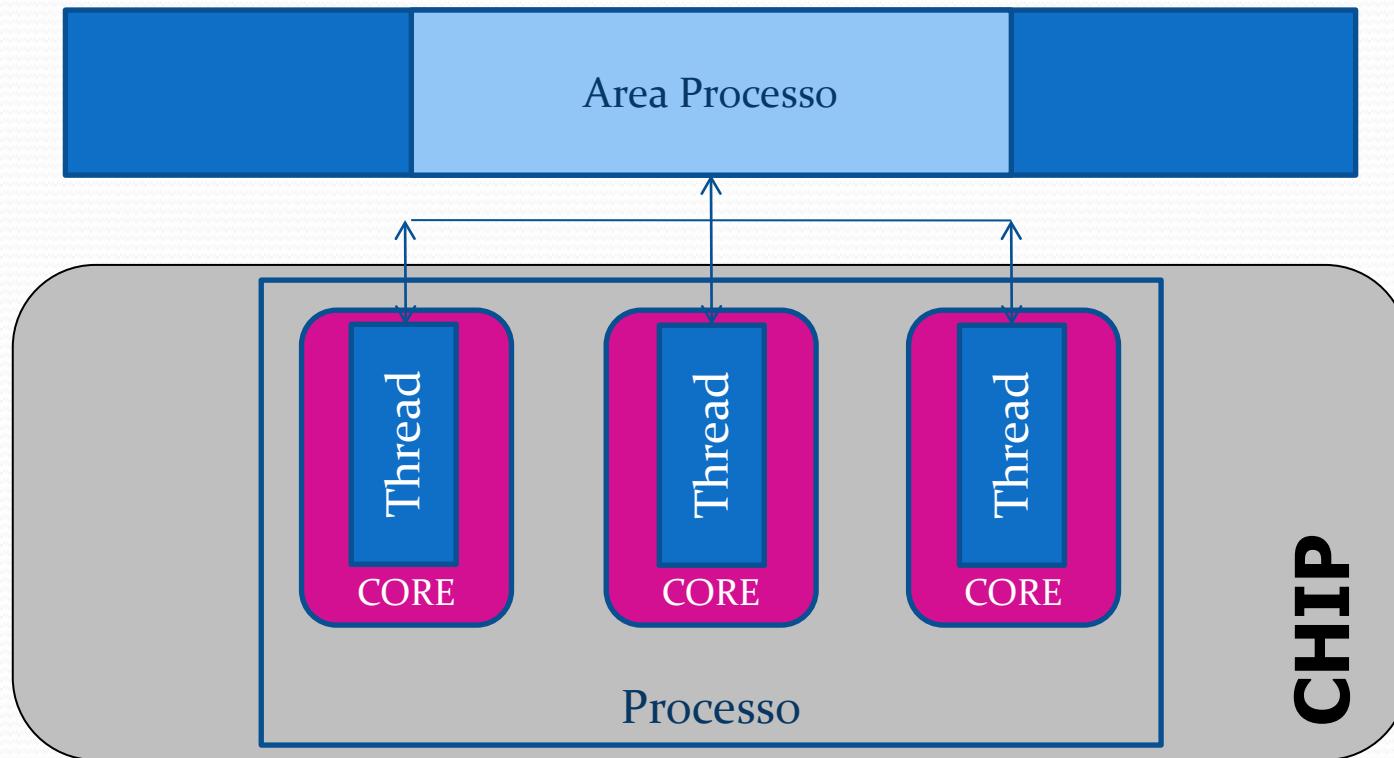
# Threads vs Processi

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



# Threads vs Processi

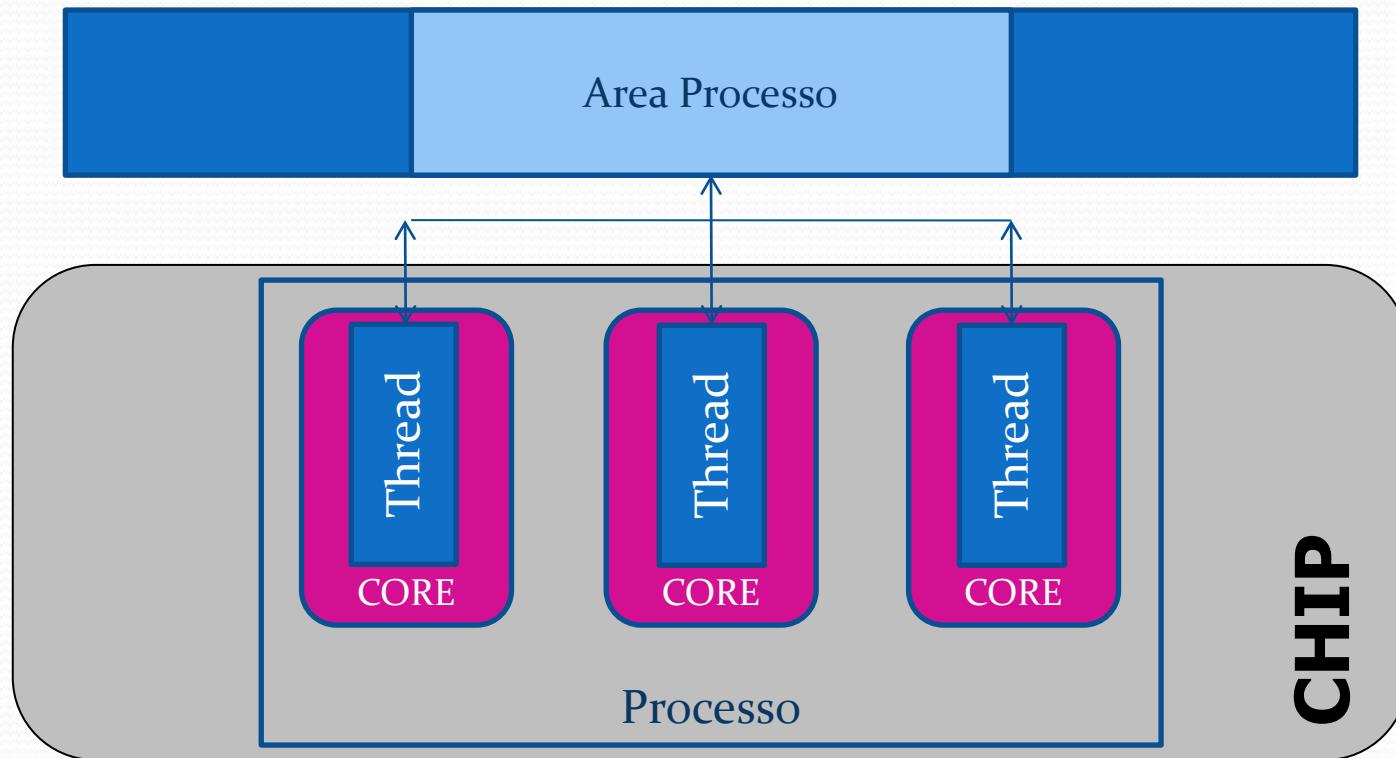
Lavorano insieme in maniera naturale. Si può pensare di dividere il lavoro tra thread di uno stesso processo, invece che tra processi diversi.



# Threads vs Processi

Vantaggio: leggerezza ed efficienza

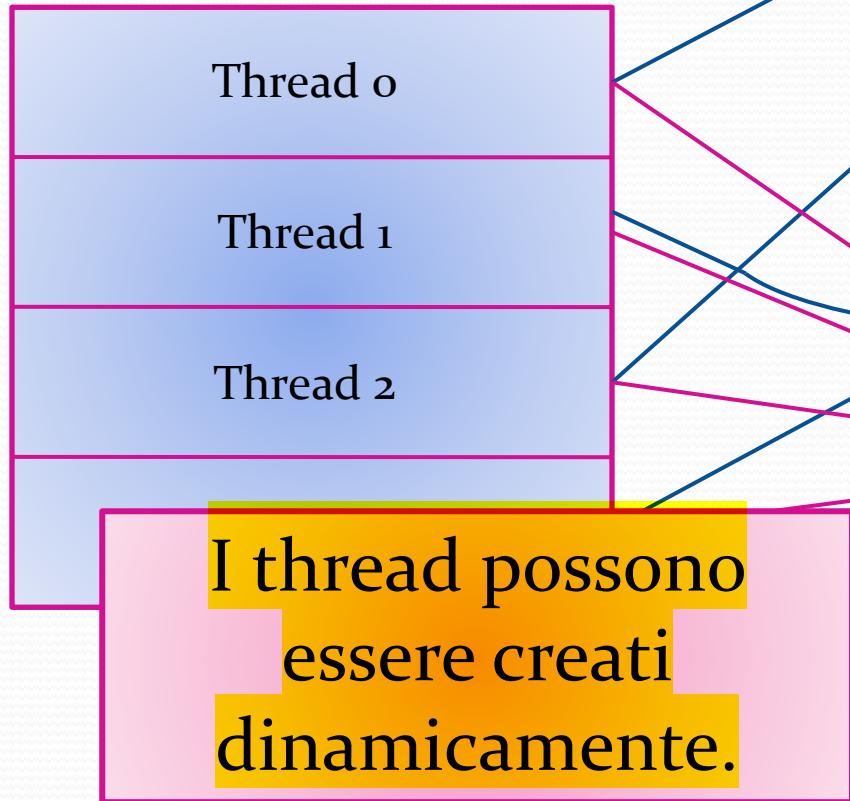
Svantaggio: i dati non sono protetti, è necessaria **sincronizzazione**



# Threads

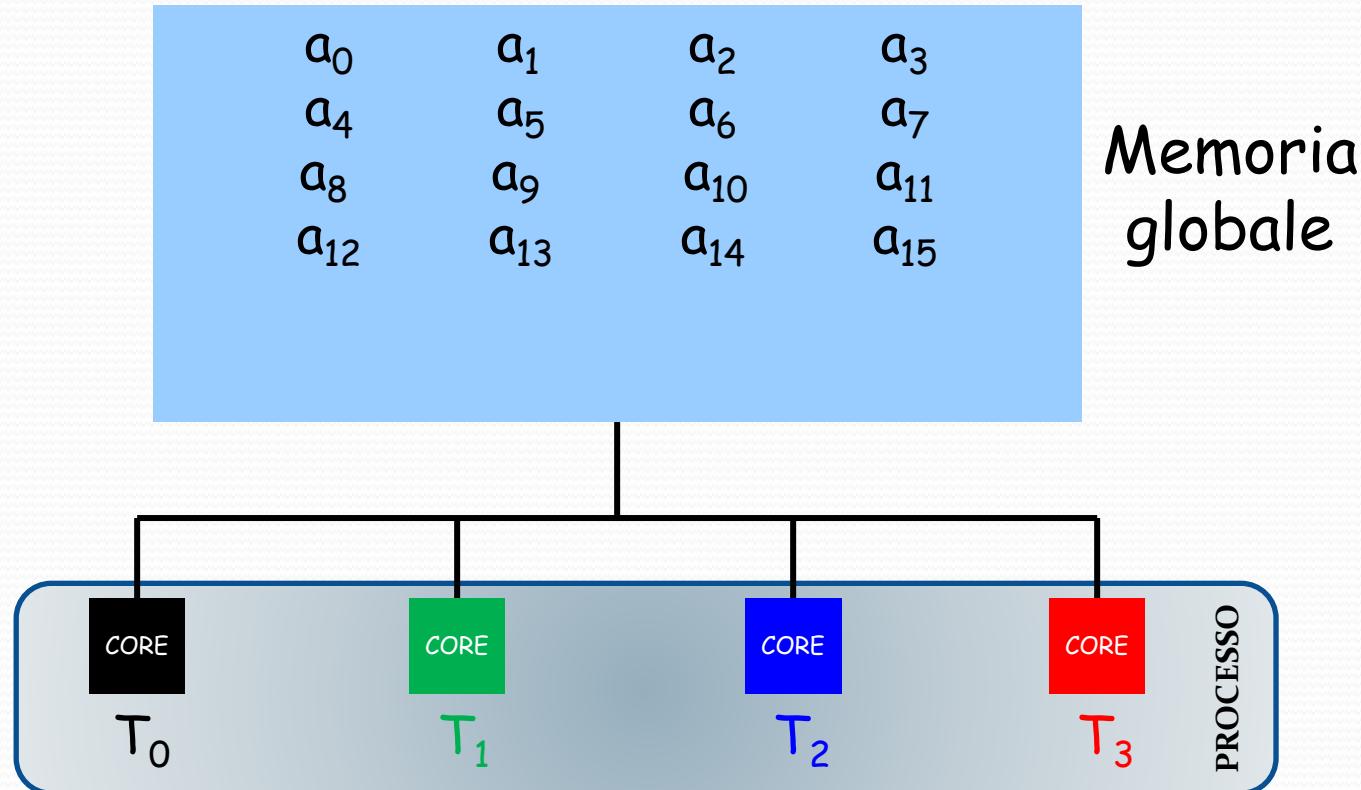
I thread vengono coordinati attraverso la **sincronizzazione** degli accessi alle variabili condivise.

Processo multi-thread



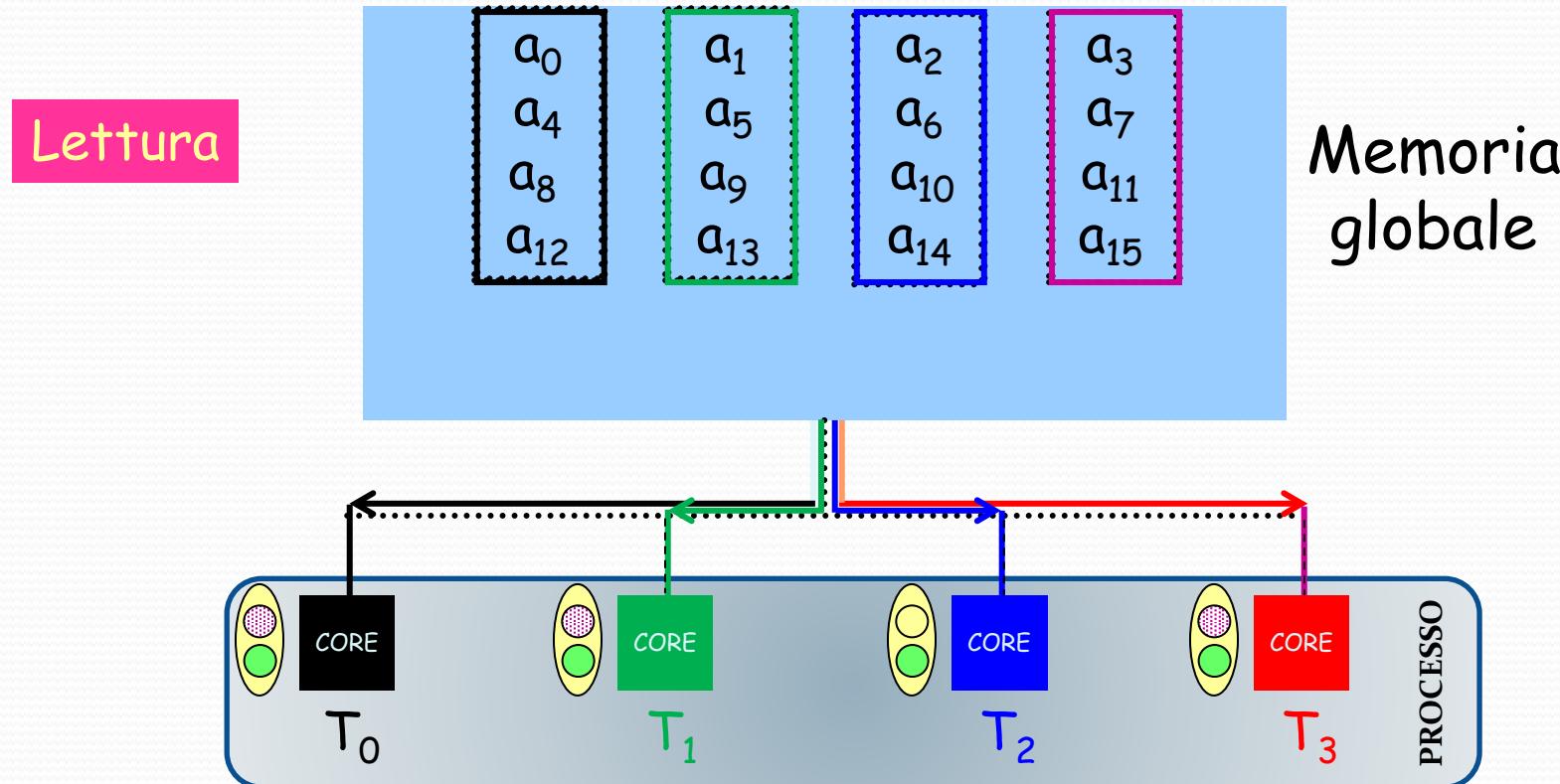
# Esempio: Somma

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



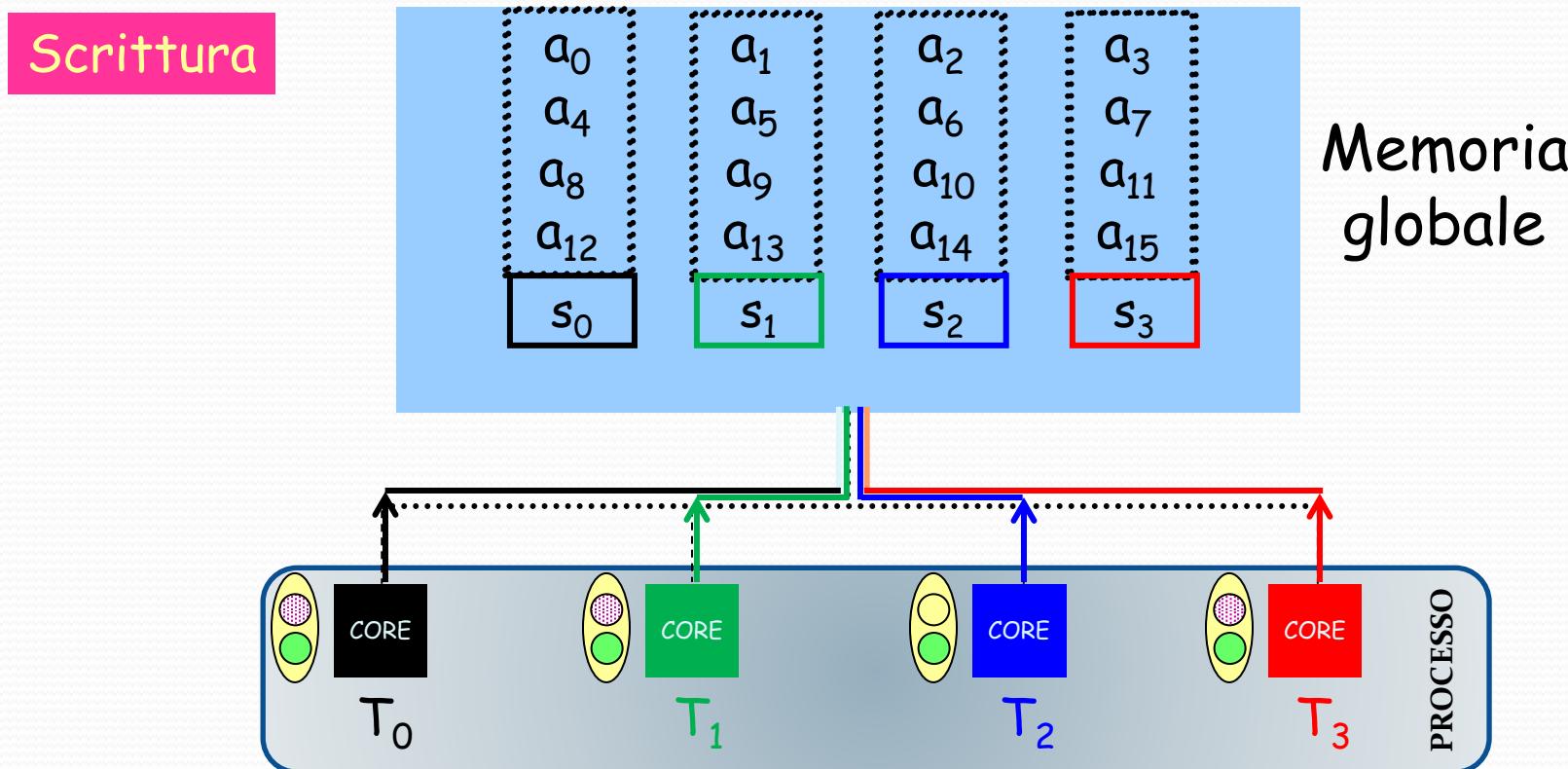
# Esempio: Somma

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



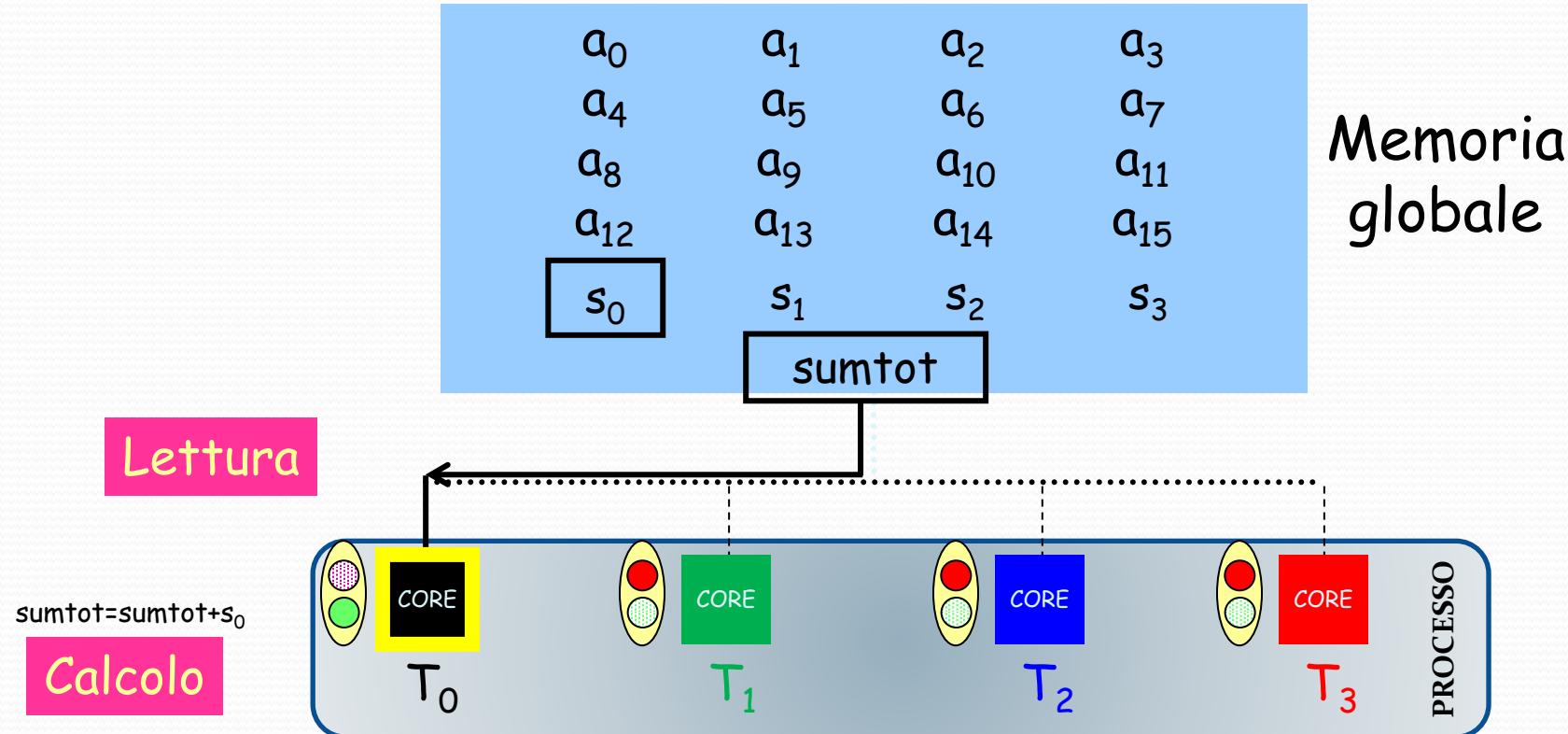
# Esempio: Somma

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



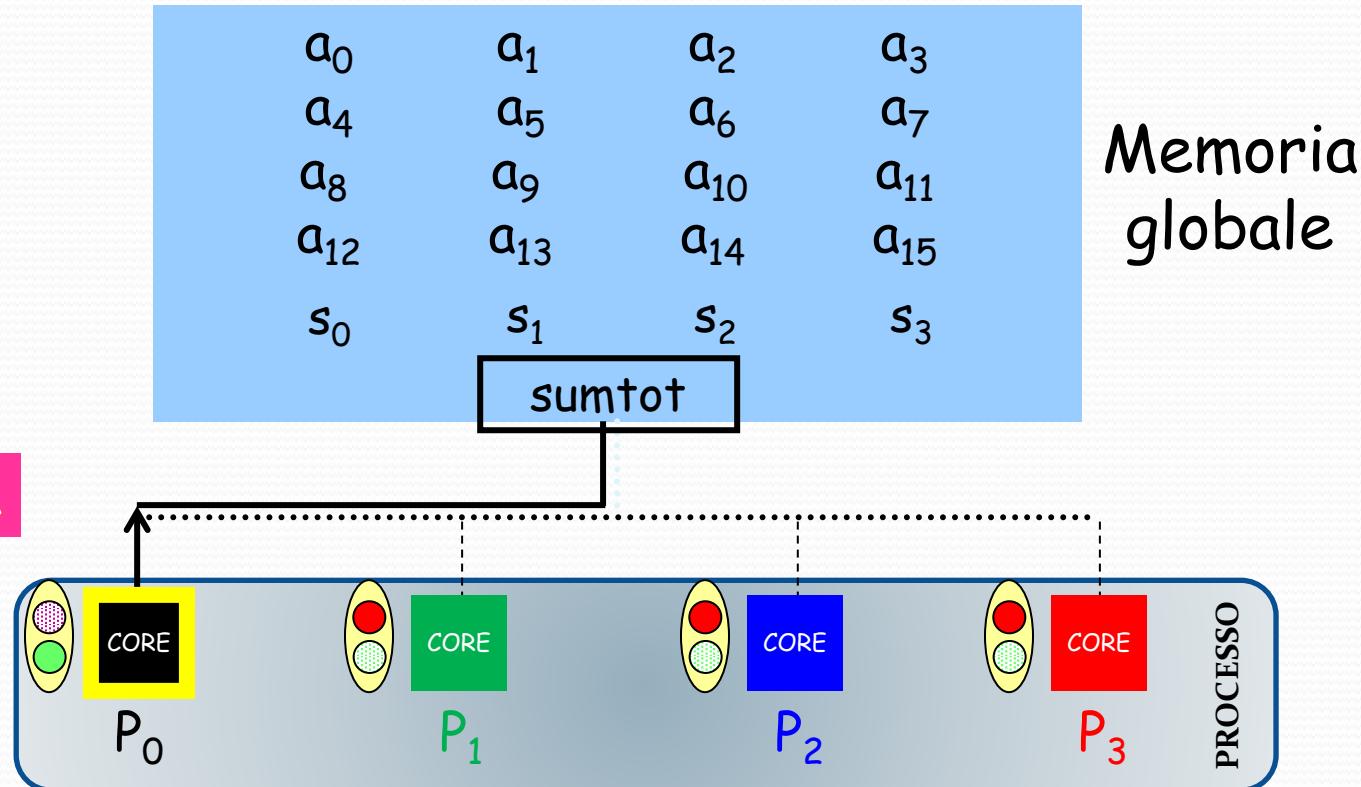
# Esempio: Somma

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



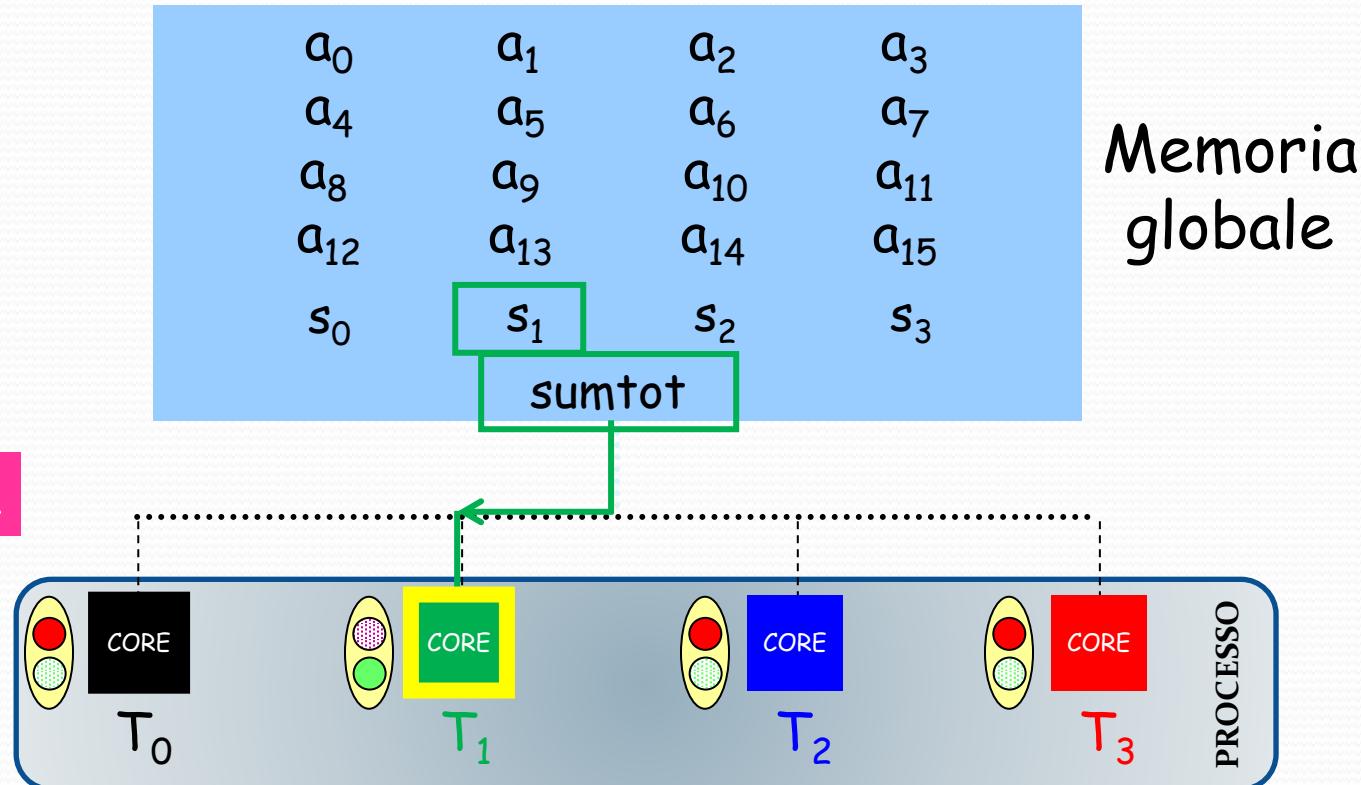
# Esempio: Somma

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



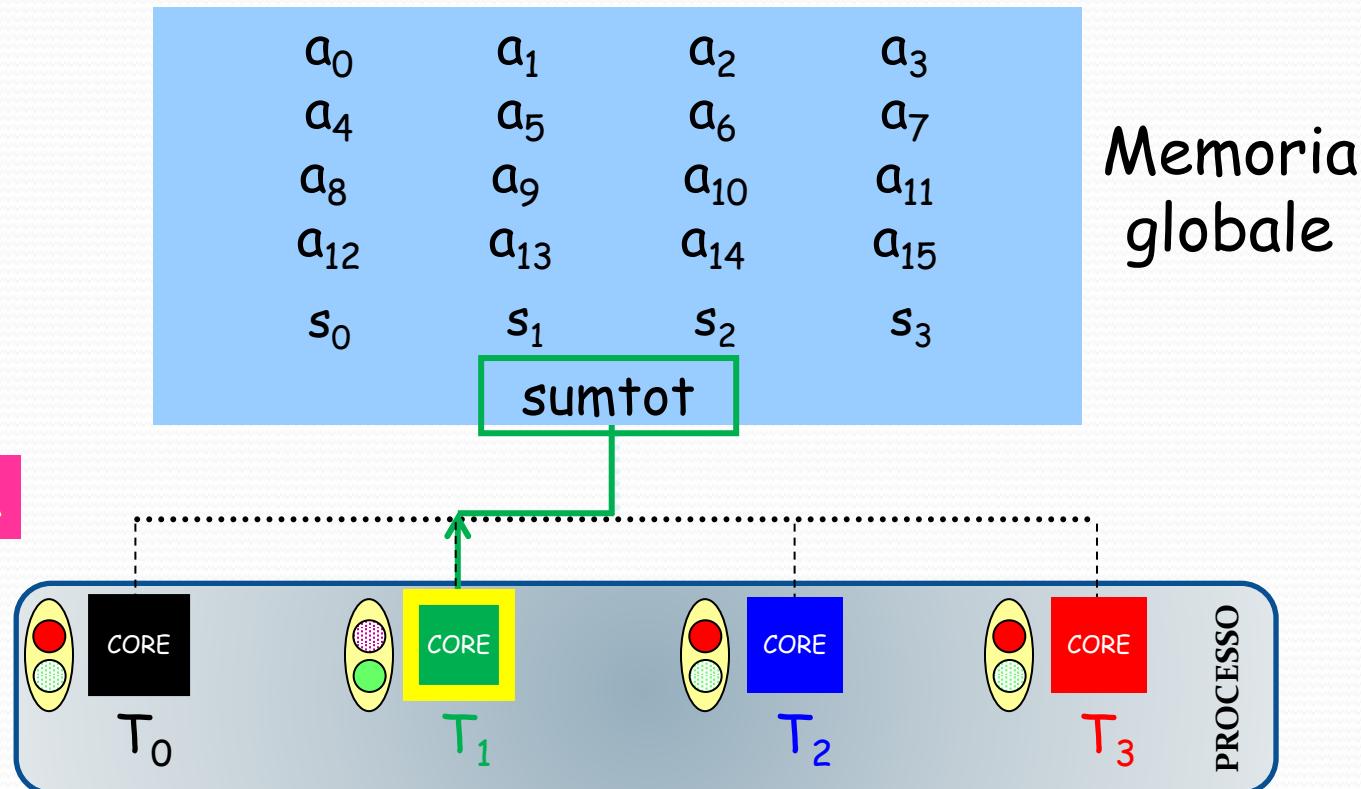
# Esempio: Somma

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



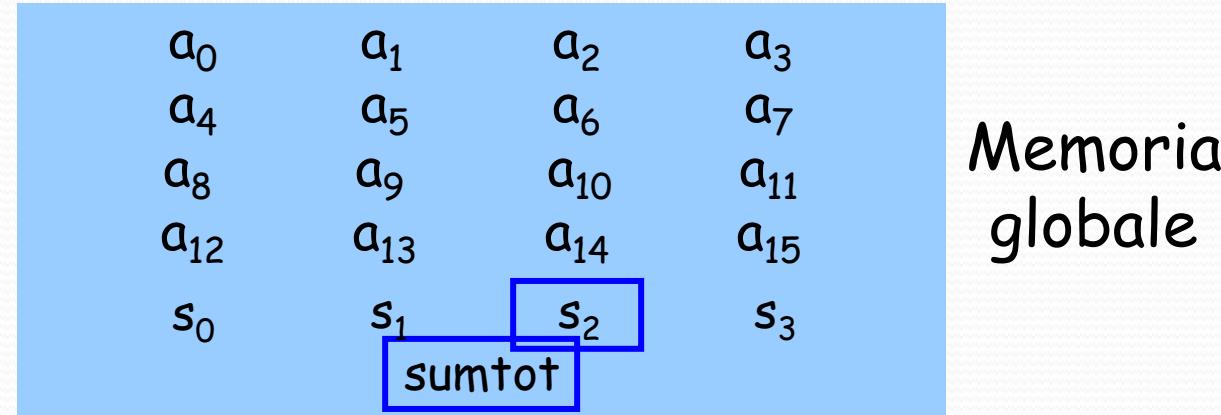
# Esempio: Somma

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



# Esempio: Somma

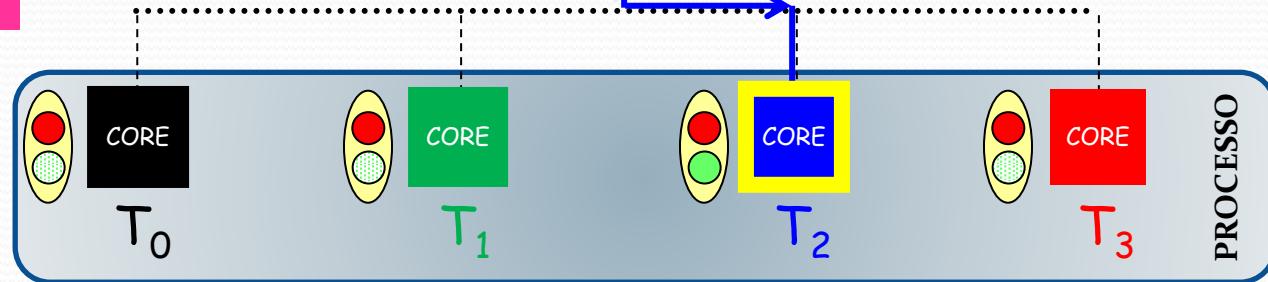
- Esempio:  $N=16$ ,  $p=4$



Lettura

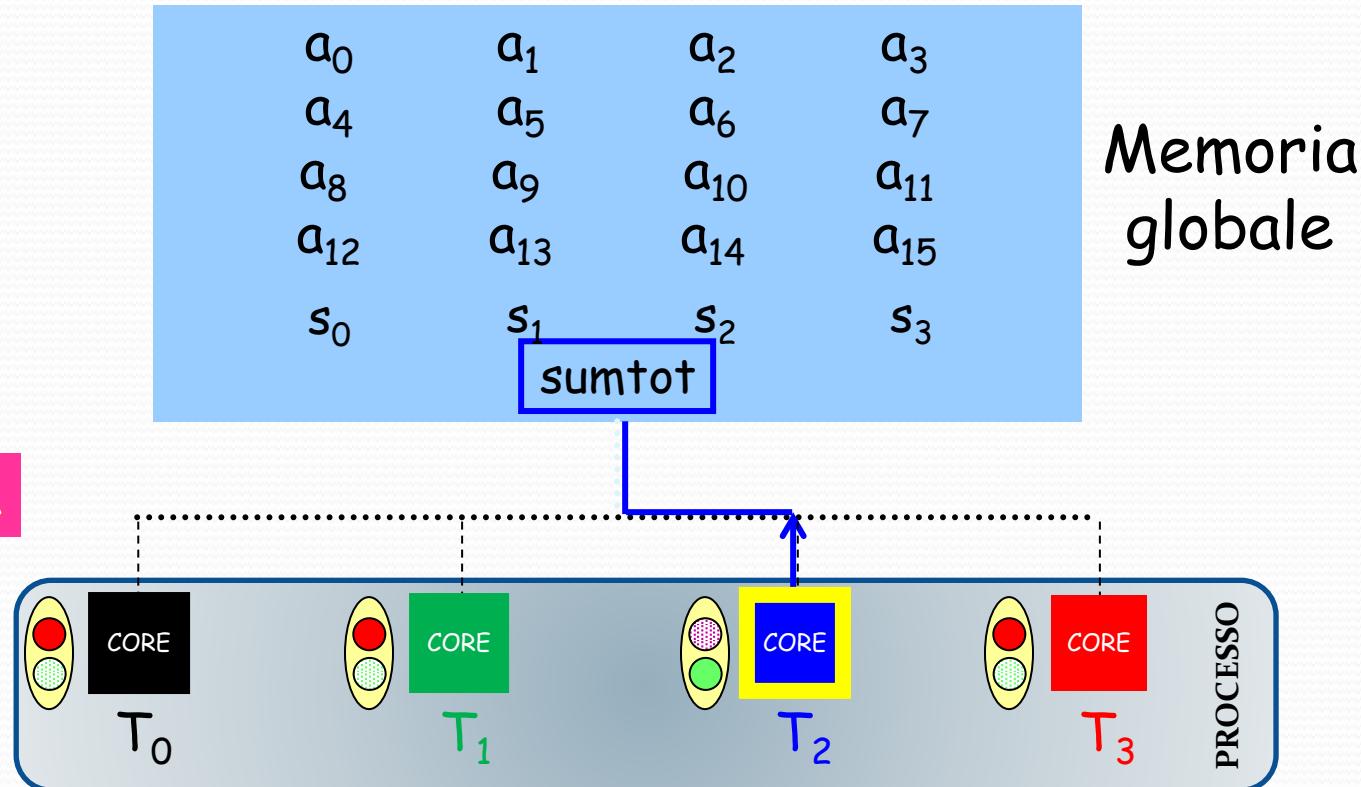
sumtot=sumtot+s<sub>2</sub>

Calcolo



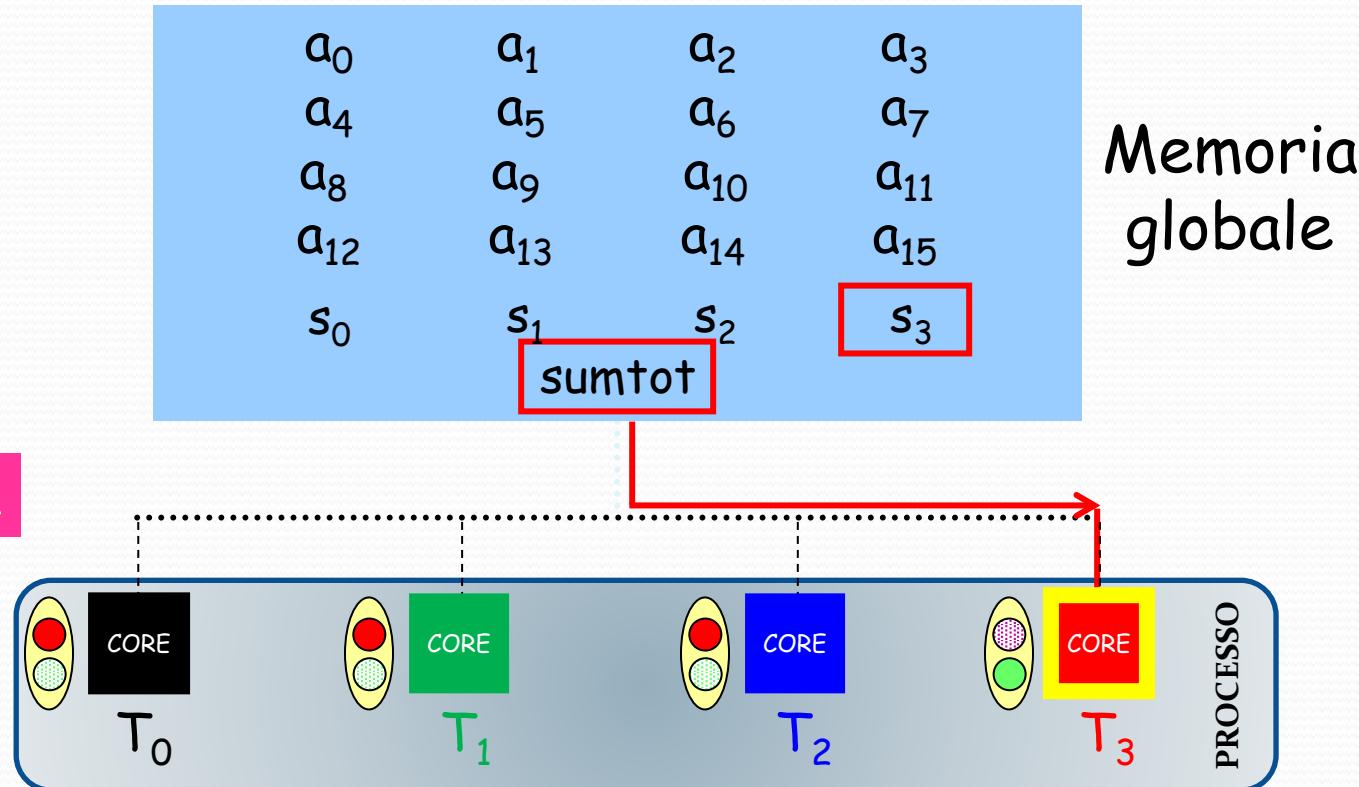
# Esempio: Somma

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



# Esempio: Somma

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

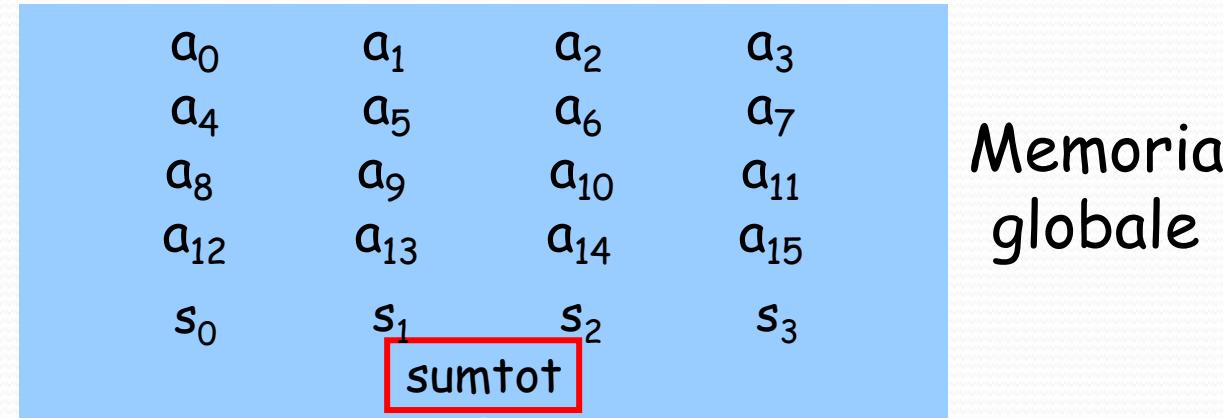


$\text{sumtot} = \text{sumtot} + s_3$

**Calcolo**

# Esempio: Somma

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

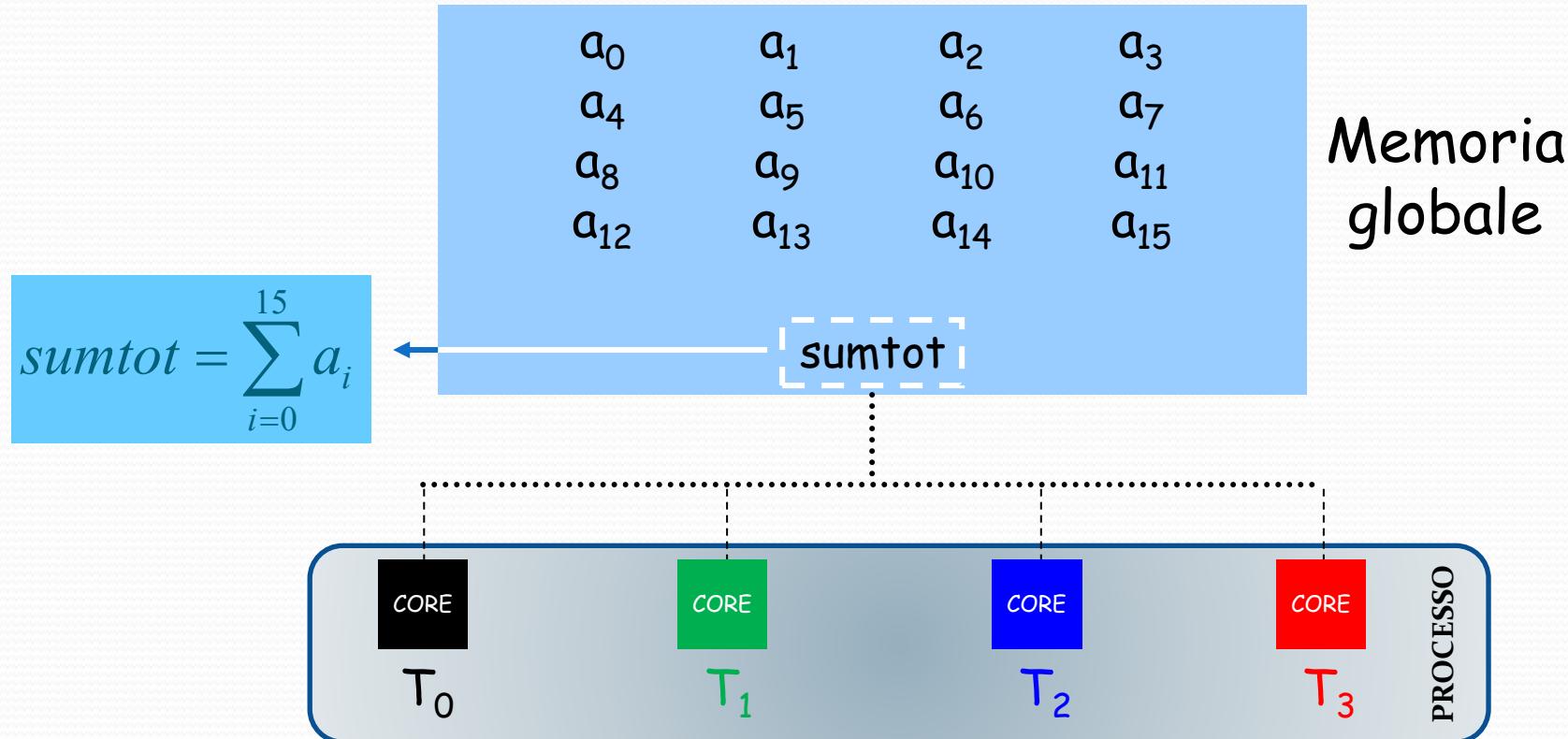


Scrittura



# 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+ ai
    endfor
    ...
end
```

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

$p = \# \text{ thread}$

Porzione  
di codice  
eseguita  
da ogni  
thread  $T_i$

begin

```
sumtot := 0
forall Ti, 0 ≤ i ≤ p-1 do
    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)
```

end forall

end

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

$p = \# \text{ thread}$

Porzione di codice eseguita da ogni thread  $T_i$

private

shared

```
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)
```

Uso di semafori per sincronizzare gli accessi

Regione Critica

# 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 **Algoritmo 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 Multi Processing
- 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

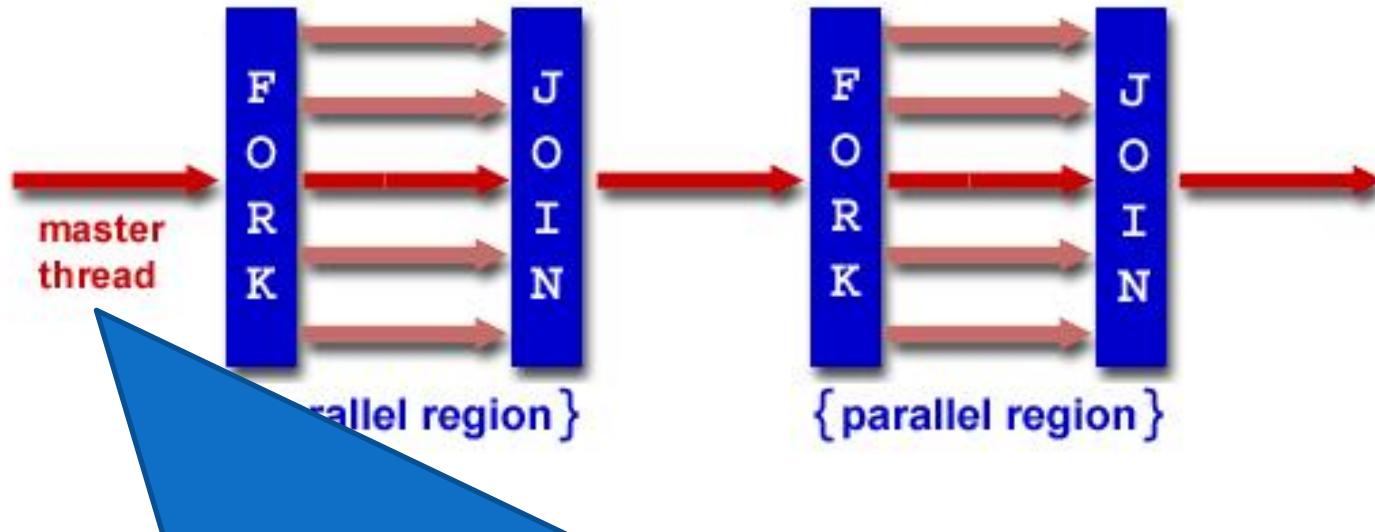


Algoritmo parallelo con OpenMP

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

# Introduzione ad OpenMp

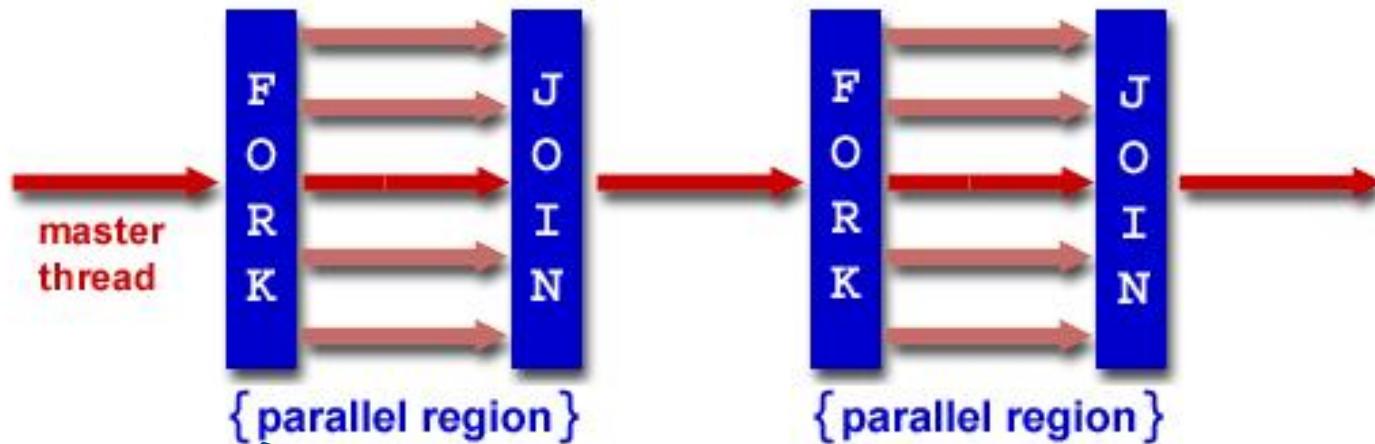
- Il modello d'esecuzione parallela è quello *fork-join*



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

# Introduzione ad OpenMp

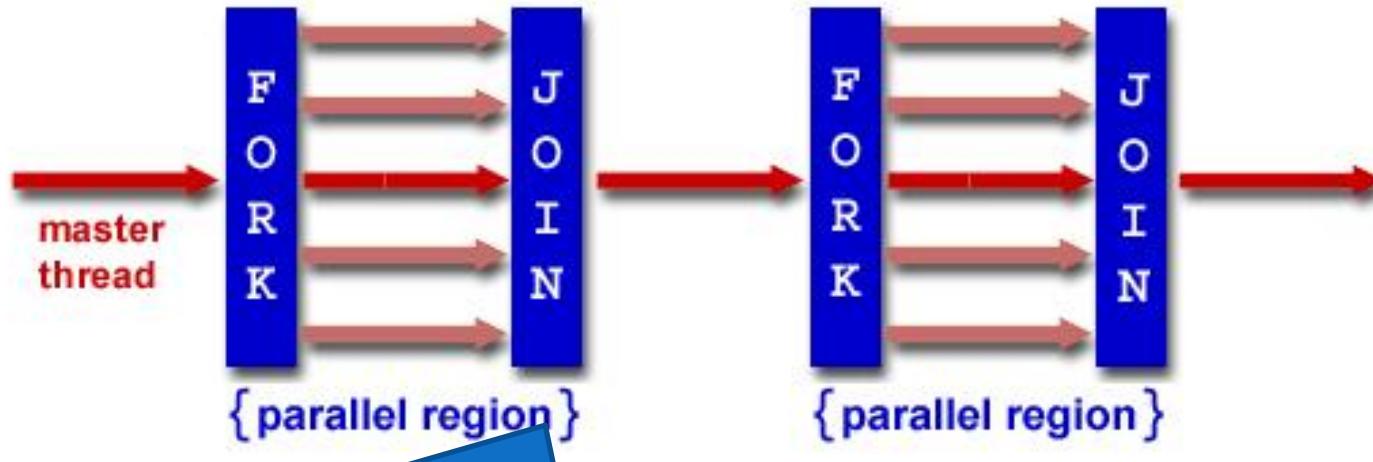
- Il modello d'esecuzione parallela è quello *fork-join*



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

# Introduzione ad OpenMp

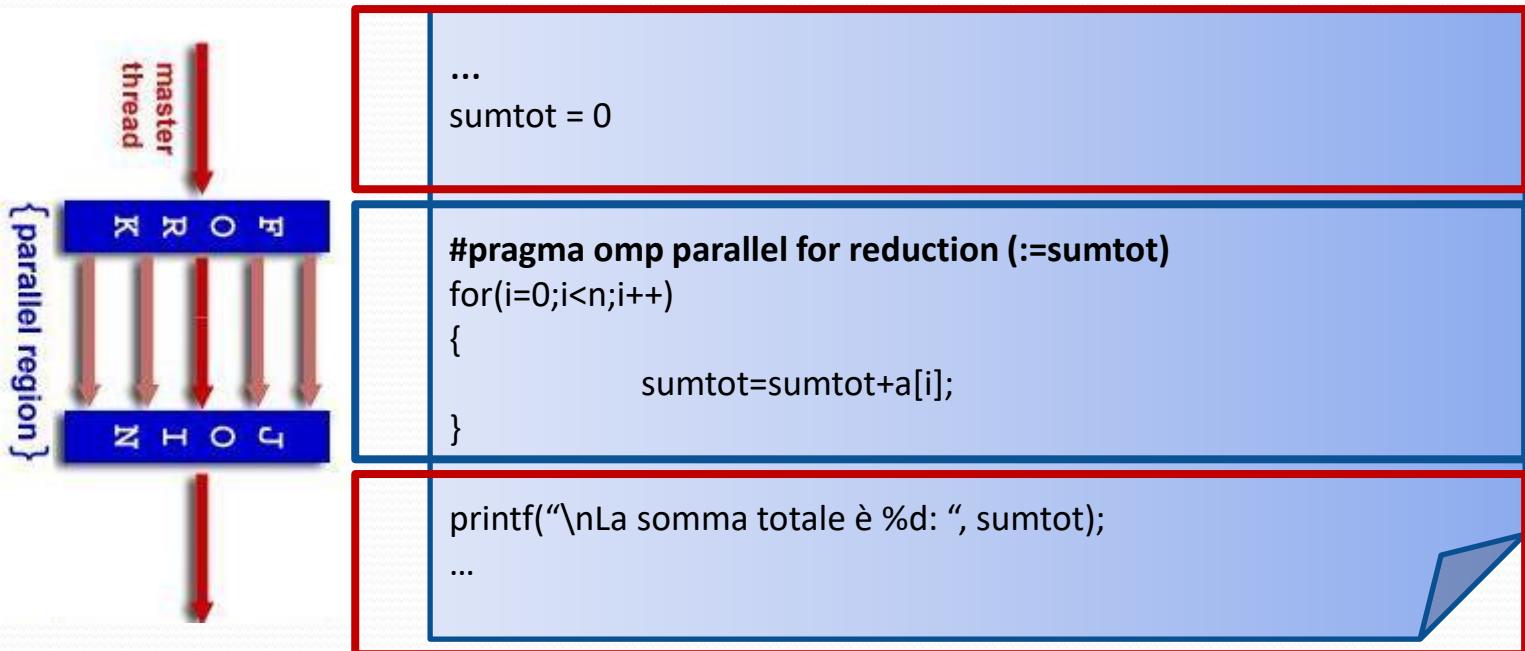
- Il modello d'esecuzione parallela è quello *fork-join*



**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**



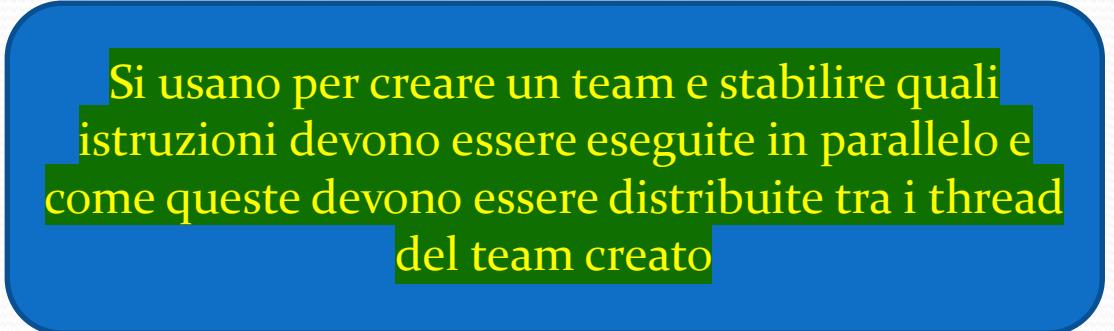
Non è garantita automaticamente e costantemente la **consistenza** della memoria



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

# Introduzione ad OpenMp

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



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



Istruzioni in un formato speciale, comprensibili solo ad un compilatore dotato di supporto OpenMp



Possono essere innestate per ottenere un parallelismo multilivello

# 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 runtime (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

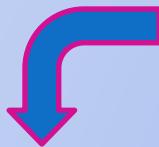
# Struttura generica del codice

```
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  
    ...  
    ...  
    ...  
    ...  
}
```



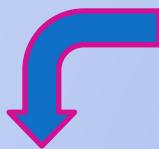
Necessità di studiare una strategia di parallelizzazione come abbiamo imparato a fare nel caso della memoria distribuita

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  
  
    ...  
  
    ...  
  
    ...  
  
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

Perché non  
“while”



Esclusione  
della barriera  
in uscita

# 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

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

## MODELLO S-M

In genere tutti i dati sono condivisi da tutti i thread

# Clausole

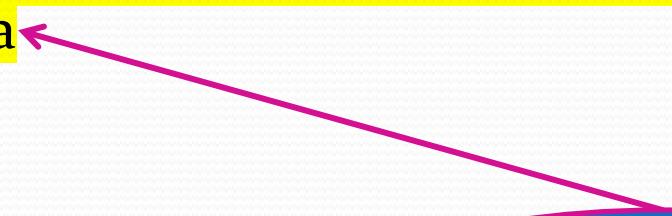
- *default(shared|none)*: controlla gli attributi di data-sharing delle variabili in un costrutto

**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

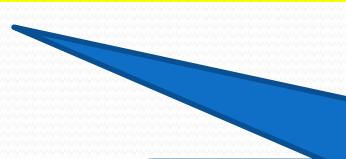
- *default(shared|none)*: controlla gli attributi di data-sharing delle variabili in un costrutto
- *shared(list)*: gli argomenti contenuti in *list* sono condivisi tra i thread del team
- *private(list)*: gli argomenti contenuti in *list* sono privati per ogni thread che li utilizza

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 del team  
All'uscita le variabili manterranno come valore l'"ultimo" della sezione parallela (se fosse stata eseguita in sequenziale)
- *firstprivate(list)*: gli argomenti contenuti in *list* sono privati per i thread del team e quelli originali verranno aggiornati al termine della sezione parallela
- *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.
- *lastprivate(list)*: gli argomenti in *list* sono privati per i thread originali e per quelli originati da questi thread. I valori inizializzati con il valore che avevano gli argomenti in *list* saranno conservati al termine della costruzione.
- *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 (0) 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()
{
    #pragma omp parallel
    printf("Hello from thread %d, nthreads %d\n",
        omp_get_thread_num(), omp_get_num_threads());
    return 0;
}
```

Creazione di un team di thread

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 diversi provider di compilatori, tra questi il gcc
  - Per compilare un programma OpenMp:
    - aggiungere la flag -fopenmp
    - fare make
- 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, si esegue lo stesso come di consueto
  - Si può eseguire direttamente dal terminale dello 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/>

# OpenMp

Introduzione agli strumenti  
Esempio: Prodotto Matrice-Vettore

Prof. G. Laccetti

# Esempio: Hello World

Libreria

```
#include <omp.h>
#include <stdio.h>
int main()
{
    #pragma omp parallel
    printf("Hello from thread %d, nthreads %d\n", omp_get_thread_num(), omp_get_num_threads());
    return 0;
}
```

Creazione di un team di thread

Library function per conoscere l'id del thread chiamante

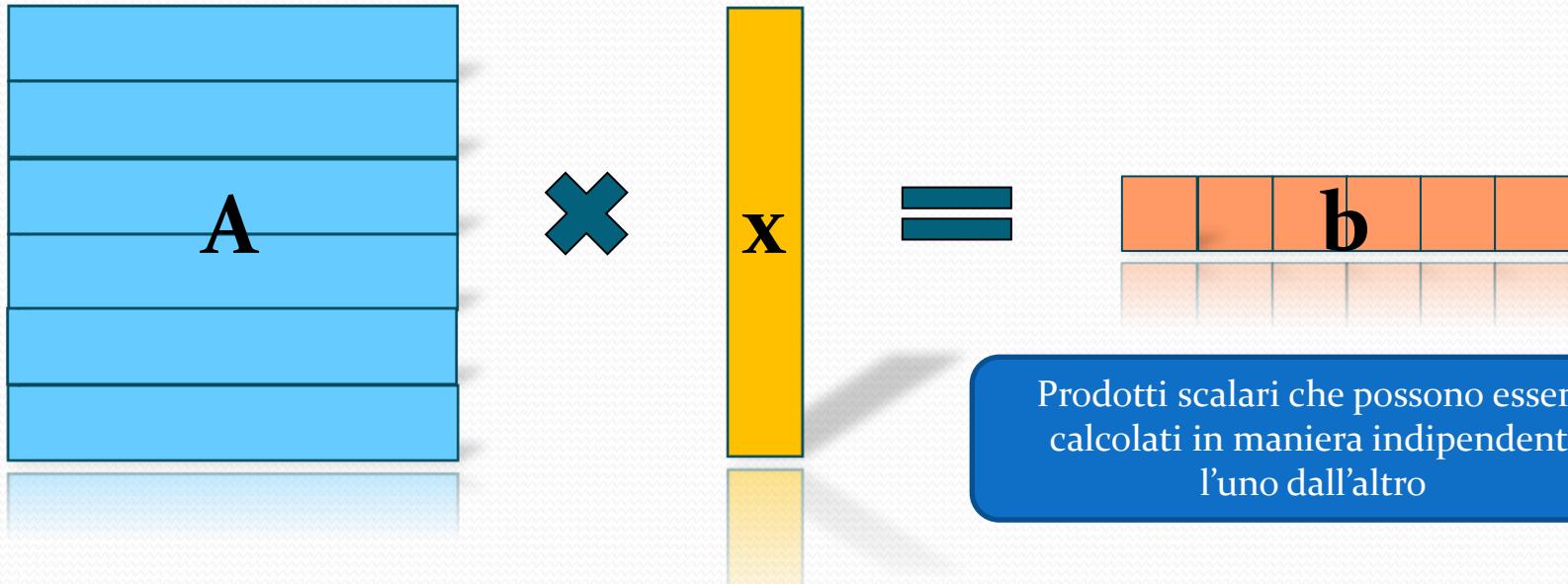
Library function per conoscere il numero di thread attivi

# Esempio: Prodotto Mat-Vet

## Problema

$$\text{Prodotto } Ax=b$$

$$A \in \mathbb{R}^{n \times m} \quad x \in \mathbb{R}^m \quad b \in \mathbb{R}^n$$



# Esempio: Prodotto Mat-Vet

Programma chiamante

```
main()  
begin
```

*dichiarazione delle variabili  
allocazione della memoria  
assegnazione di valori alle dimensioni della matrice (**n,m**)  
assegnazione dei valori alla matrice **A** ed al vettore **x**  
inizializzazione del vettore risultato **b***

```
    b:=matxvet(m,n,x,A)  
  
    for i:=0 to n-1  
    begin  
        stampa b[i]  
    end  
end
```

Funzione che realizza il  
prodotto della matrice **A** per il  
vettore **x**

# Esempio: Prodotto Mat-Vet

```
#include <omp.h>
#include <math.h>
#include <stdlib.h>
...
double * matxvet(int m, int n, double *x, double **A){
    int i,j;
    double *b;
    ...
    /*allocazione memoria per b*/
    ...
}
```

Parte  
sequenziale  
eseguita dal solo  
master thread

# Esempio: Prodotto Mat-Vet

```
#include <omp.h>
#include <math.h>
#include ...
...
double prodottoMatVet(int m, int n, int n, double *x, double **A){
    double *b, ...
    /*allocazione memoria per b*/
    ...
#pragma omp parallel for default(none) shared(m,n,A,x,b) private(i,j)
    for (i=0; i<n; i++){
        for (j=0; j<m; j++)
            b[i] += A[i][j]*x[j];
    }
}
```

Creazione di un team di thread

i-mo prodotto scalare

# Esempio: Prodotto Mat-Vet

```
#include <omp.h>
#include <math.h>
#include <stdlib.h>
...
double * matxvet(int m, int n, double *x, double *A, double *b) {
    int i,j;
    double *b;
    ...
    /*allocazione memoria per b*/
    ...
#pragma omp parallel for default(None) shared(m,n,A,x,b) private(i,j)
    for (i=0; i<n; i++){
        for (j=0; j<m; j++)
            b[i] += A[i][j]*x[j];
    }
}
```

Distribuzione delle iterazioni del for tra i thread

Con la clausola *schedule* si può scegliere la distribuzione

i-mo prodotto scalare

# Esempio: Prodotto Mat-Vet

```
#include <omp.h>
#include <math.h>
#include <stdlib.h>
...
double * matxvet()
{
    int i,j;
    double *b;
    ...
    /*allocazione memoria per b*/
    ...
#pragma omp parallel for default(None) shared(m,n,A,x,b) private(i,j)
    for (i=0; i<n; i++){
        for (j=0; j<m; j++)
            b[i] += A[i][j]*x[j];
    }
}
```

Sarà il programmatore a stabilire cosa è condiviso e cosa non lo è

i-mo prodotto scalare

# Esempio: Prodotto Mat-Vet

```
#include <omp.h>
#include <math.h>
#include <stdlib.h>
...
double * matxvet(int m, int n, double *x, double **A){
    int i,j;
    double *b;
    ...
    /*allocazione memoria per b*/
    ...
#pragma omp parallel for default(None) shared(m,n,A,x,b) private(i,j)
    for (i=0; i<n; i++){
        for (j=0; j<m; j++)
            b[i] += A[i][j]*x[j];
    }
}
```

Variabili condivise tra i thread

Tutti devono poterle leggere

Se fossero private  
m, n, A e x  
verrebbero **de-inizializzate**

i-mo prodotto scalare

# Esempio: Prodotto Mat-Vet

```
#include <omp.h>
#include <math.h>
#include <stdlib.h>
...
double * matxvet(int m, int n, double *x, double **A){
    int i,j;
    double *b;
    ...
    /*allocazione memoria per b*/
    ...
#pragma omp parallel for default(None) shared(m,n,A,x,b) private(i,j)
    for (i=0; i<n; i++){
        for (j=0; j<m; j++)
            b[i] += A[i][j]*x[j];
    }
}
```

Variabili condivise tra i thread

Tutti devono poterle leggere

Se fosse privata b, non sarebbe accessibile fuori dalla regione parallela

i-mo prodotto scalare

# Esempio: Prodotto Mat-Vet

```
#include <omp.h>
#include <math.h>
#include <stdlib.h>
...
double *  
...  
    /*allocazione memoria per b */  
...
#pragma omp parallel for default(None) shared(m,n,A,x,b) private(i,j)  
for (i=0; i<n; i++){  
    for (j=0; j<m; j++)  
        b[i] += A[i][j]*x[j]; }  
}  
}
```

Comportamento imprevedibile

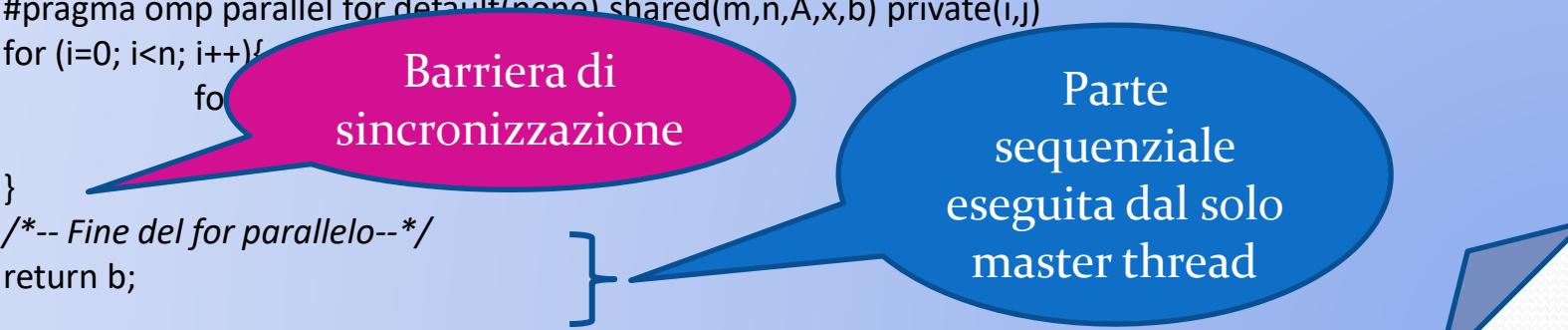
Se i e j fossero condivise, un thread potrebbe modificare il contatore per un altro

Variabili private per ogni thread

i-mo prodotto scalare

# Esempio: Prodotto Mat-Vet

```
#include <omp.h>
#include <math.h>
#include <stdlib.h>
...
double * matxvet(int m, int n, double *x, double **A){
    int i,j;
    double *b;
    ...
    /*allocazione memoria per b*/
    ...
#pragma omp parallel for default(none) shared(m,n,A,x,b) private(i,j)
    for (i=0; i<n; i++){
        fo
    }
    /*-- Fine del for parallelo--*/
    return b;
}
```



# Esempio: Prodotto Mat-Vet

```
#include <omp.h>
#include <math.h>
#include <stdlib.h>
...
double * matxvet(int m, int n, double *x, double **A){
    int i,j;
    double *b;
    ...
    /*allocazione memoria per b*/
    ...
#pragma omp parallel for default(None) shared(m,n,A,x,b) private(i,j)
    for (i=0; i<n; i++){
        for (j=0; j<m; j++)
            b[i] += A[i][j]*x[j];
    }
    /*-- Fine del for parallelo--*/
    return b;
}
```

A come array  
lineare

Ottimizzabile con  
la cura di alcuni  
dettagli

vettori *restricted*  
Es: *double \* restrict x*

Forzare i vettori ad  
occupare regioni  
disgiunte di memoria

# Riferimenti bibliografici

**Using OpenMp**

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

The MIT Press

<http://openmp.org/>

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

“Prendere” il tempo di esecuzione in  
ambiente a memoria distribuita

# Prendere i tempi di esecuzione

- Rispondiamo con queste note a due domande:
  - Come facciamo a sapere che i processi stanno lavorando effettivamente con memoria distribuita?
  - Quali istruzioni si devono usare nel codice per avere il tempo giusto in output?

# Eseguire in un ambiente a memoria distribuita

- In questa fase, vogliamo che i nostri processi vengano eseguiti su nodi diversi, perché vogliamo lavorare in un ambiente omogeneo a memoria distribuita.
- Ricordando come è fatto il nostro cluster (ogni nodo ha 8 processori, i quali in parte condividono la memoria), dobbiamo quindi assicurarci che tali processi vengano eseguiti su nodi diversi
- Vediamo un “trucco” per realizzare questa condizione, con una rapida **modifica allo script pbs** già visto a lezione
- Approfondiremo successivamente gli altri modi di fare la stessa cosa

# Modifiche al job-script

## per usare processori su macchine diverse

```
#!/bin/bash
#####
## The PBS directives ##
#####
#PBS -q studenti
#PBS -N Hello
#PBS -o Hello.out
#PBS -e Hello.err
#PBS -l nodes=8:ppn=8
#####
# -q coda su cui va eseguito il job #
# -l numero di nodi richiesti #
# -N nome job(stesso del file pbs) #
# -o, -e nome files contenente l'output e gli errori #
#####
#           #
#   stampa di qualche informazione sul job   #
#           #
#####
```

#PBS -j oe  
per far confluire lo standard error nello standard output

**8 nodi, 8 processori (8 processori per nodo)**

# Modifiche al job-script

```
echo PBS: qsub is running on $PBS_O_HOST  
echo PBS: originating queue is $PBS_O_QUEUE  
echo PBS: executing queue is $PBS_QUEUE  
echo PBS: working directory is $PBS_O_WORKDIR  
echo PBS: execution mode is $PBS_ENVIRONMENT  
echo PBS: job identifier is $PBS_JOBID  
echo PBS: job name is $PBS_JOBNAME  
echo PBS: node file is $PBS_NODEFILE  
echo PBS: number of nodes is $NNODES  
echo PBS: current home directory is $PBS_O_HOME  
echo PBS: PATH = $PBS_O_PATH
```

```
echo -----  
echo 'Job reserved node(s): '  
cat $PBS_NODEFILE ←  
echo -----
```

Con questo **cat** vedete tutti processori  
che vi siete riservati

In questo file ci sono tutti i nodi riservati  
(ogni nodo compare tante volte quanti sono i suoi  
processori, cioè ci sono  $8 \times 8 = 64$  macchine)

# Modifiche al job-script

```
sort -u $PBS_NODEFILE > hostlist
```

```
NCPU=`wc -l < hostlist`  
echo -----  
echo ' This job is allocated on '${NCPU}' cpu(s)' on hosts:'  
cat hostlist  
echo -----
```

```
PBS_O_WORKDIR=$PBS_O_HOME/ProgettoHello
```

```
echo -----  
echo "Eseguo: /usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/Hello $PBS_O_WORKDIR/Hello.c"
```

```
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/Hello $PBS_O_WORKDIR/Hello.c
```

```
echo "Eseguo: /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU $PBS_O_WORKDIR/Hello"  
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU $PBS_O_WORKDIR/Hello
```

## Sort

ordina le righe del file

-u

prende solo una volta quelle uguali

L'output viene scritto in **hostlist**, in cui ci saranno le diverse macchine ma una sola volta (8 macchine)

# Modifiche al job-script

```
sort -u $PBS_NODEFILE > hostlist
```

```
NCPU=`wc -l < hostlist`
```

```
echo -----
```

```
echo ' This job is allocated on '${NCPU}' cpu(s)' on hosts:'
```

```
cat hostlist
```

```
echo -----
```

```
PBS_O_WORKDIR=$PBS_O_HOME/ProgettoHello
```

```
echo -----
```

```
echo "Eseguo: /usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_
```

```
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/H
```

```
echo "Eseguo: /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machine
```

```
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU $PBS_O_WORKDIR>Hello
```

**wc -l**

conta le righe del file **hostlist** (8)

Nel file ogni riga è una macchina che vogliamo usare, quindi, l'output del comando è il nostro numero di cpu.

**NOTA:**

Nell'altro file PBS visto, si faceva la stessa operazione direttamente sul file \$PBS\_NODEFILE. (che ora però ha 64 righe)

# Modifiche al job-script

```
sort -u $PBS_NODEFILE > hostlist
```

```
NCPU=`wc -l < hostlist`  
echo -----  
echo ' This job is allocated on '${NCPU}' cpu(s)' ' on hosts:'  
cat hostlist ←  
echo -----
```

Con questo **cat** vedete tutti processori  
che **userete**

```
PBS_O_WORKDIR=$PBS_O_HOME/ProgettoHello
```

```
echo -----  
echo "Eseguo: /usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/Hello $PBS_O_WORKDIR/Hello.c"  
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/Hello $PBS_O_WORKDIR/Hello.c  
  
echo "Eseguo: /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU $PBS_O_WORKDIR/Hello"  
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU $PBS_O_WORKDIR/Hello
```

# Modifiche al job-script

```
sort -u $PBS_NODEFILE > hostlist
```

```
NCPU=`wc -l < hostlist`  
echo -----  
echo ' This job is allocated on '${NCPU}' cpu(s)' ' on hosts:'  
cat hostlist  
echo -----
```

```
PBS_O_WORKDIR=$PBS_O_HOME/ProgettoHello
```

```
echo -----  
echo "Eseguo: /usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/Hello $PBS_O_WORKDIR/Hello.c"  
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/Hello $PBS_O_WORKDIR/Hello.c
```

```
echo "Eseguo: /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU $PBS_O_WORKDIR/Hello"  
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU $PBS_O_WORKDIR/Hello
```

RICORDATE di dare il giusto file a mpiexec!!!!

# **ATTENZIONE!!!!**

Utilizzate questo secondo tipo di script

**SOLO**

quando sarete certi che il vostro codice  
funziona e realizza sempre la somma  
che volete



# Istruzioni da aggiungere al programma per ottenere il tempo di esecuzione in output

MPI fornisce una funzione semplice:

**double MPI\_Wtime()**

che restituisce un tempo in secondi (double)

- L'utilizzo di questa funzione all'interno del vostro codice è piuttosto semplice...
- Vediamo un esempio attraverso l'algoritmo per la somma di n numeri, già visto a lezione

# Dichiarazioni

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char *argv[]){
    int menum, nproc,... ;
    int n, nloc, tag, i,...;
    int *x, *xloc;
    double t0, t1, time;      /*servono a tutti i processi*/
    double timetot;          /*serve solo a P0*/
    MPI_Status status;
    ...
}
```

# Lettura e distribuzione dei dati (1)

```
MPI_Init(&argv, &argc);
MPI_Comm_rank(MPI_COMM_WORLD, &menu);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
if (menu==0){
    “Lettura dei dati di input: n e x”
}
MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
nloc=n/nproc
rest=n%nproc
if (menu<rest) nloc=nloc+1
“allocazione di xloc”
if (menu==0){
    xloc=x
```

# Lettura e distribuzione dei dati (2)

```
tmp=nloc
start=0
for (i=1;i<nproc;i++){
    start=start+tmp
    tag=22+i;
    if(i==rest) tmp=tmp-1
    MPI_Send(&x[start],tmp,MPI_INT,i,tag,MPI_COMM_WORLD);
}
}/*endif*/
else{
    tag=22+menu
    MPI_Recv(xloc,nloc,MPI_INT,0,tag, MPI_COMM_WORLD,&status);
}
```

# Calcolo Locale

...

...

...

...

```
MPI_Barrier(MPI_COMM_WORLD);
t0=MPI_Wtime();
/*tutti i processori*/
sum=0
for(i=0;i<nloc;i++)
    sum=sum+xloc[i]
```

...

...

...

# Calcolo della somma totale

## (I strategia di comunicazione)

...

...

```
if (menum==0){  
    for(i=1;i<nproc;i++){  
        tag=80+i  
        MPI_Recv(&sum_parz,1,MPI_int,i,tag,MPI_COMM_WORLD,&status);  
        sum=sum+sum_parz  
    }  
}  
else{  
    tag=menum+80  
    MPI_Send(&sum,1,MPI_INT,0,tag, MPI_COMM_WORLD);  
}  
t1=MPI_Wtime();
```

# Stampa del risultato (I versione)

```
...
time=t1-t0;      /*ora ogni processore SA il proprio tempo*/
printf("Sono %d: Tempo impiegato: %e secondi\n",menum,time);
...

MPI_Reduce(&time,&timetot,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
/*ora P0 sa anche il MASSIMO tra tutti i tempi dei processi*/

...
/*se ci basta che la stampi solo un processore (P0)*/
if (menum==0){
    printf("\nSomma totale=%d\n", sum);
    printf("Tempo totale impiegato: %e secondi\n",timetot);
}

}
```

# Fine Note

**Primo utilizzo  
dell'architettura parallela  
messa a disposizione dal  
*Progetto SCoPE***

# Disponibilità del cluster

---

L'accesso al cluster vi sarà possibile solo nei seguenti orari:

- dalle 14:00 alle 20:00 di tutti i lunedì del mese
- dalle 08:00 alle 20:00 di tutti i martedì del mese
- dalle 08:00 alle 20:00 di tutti i giovedì del mese
- dalle 14:00 alle 20:00 di tutti i venerdì del mese
- dalle 08:00 alle 20:00 di tutti i sabato del mese

L'indirizzo a cui inoltrare qualsiasi segnalazione riguardo il cluster, che vi invito a fare REPENTINAMENTE qualora riscontraste qualcosa che non va.

L'indirizzo è: [contactcenter@unina.it](mailto:contactcenter@unina.it) (cc [valeria.mele@unina.it](mailto:valeria.mele@unina.it) )  
scrivete dettagliando bene il problema e i tempi in cui si è verificato .

# Prime credenziali sul cluster

---

Le login per accedere al cluster sono contenute nel materiale del corso nel file  
**PDC2122-ElencoStudenti\_conlogin.pdf**  
nella colonna LOGIN

Attualmente, per ciascuno degli account, la password coincide con il CODICE FISCALE comunicato.

Al primo accesso al sistema verrà richiesto di modificarla.



**ATTENZIONE!!!**  
**Al primo accesso vi verrà chiesto di  
cambiare la password!**

MPI :

# Message Passing Interface

## MPI

# Message Passing Interface

## MPI

### Implementazioni di MPI

- MPICH
  - Argonne National Laboratory (ANL) and
  - Mississippi State University (MSU)
- OpenMPI
- ...

Linguaggi ospiti: Fortran, C, C++, Matlab

# MPI : dove trovarlo

The screenshot shows the homepage of the MPICH website at [www.mpich.org](http://www.mpich.org). The page has a green header bar with the MPICH logo and navigation links for Home, About, Downloads, Documentation, Support, and ABI Compatibility Initiative. Below the header, there's a large green box containing text about MPICH and a 'Download MPICH' button. To the right of this box is the R&D 100 award logo. At the bottom, there are sections for News & Events, Learn About MPICH, and Support.

**MPICH**

High-Performance Portable MPI

Home About Downloads Documentation Support ABI Compatibility Initiative

**MPICH** is a high performance and widely portable implementation of the **Message Passing Interface (MPI)** standard.

MPICH and its derivatives form the most widely used implementations of MPI in the world. They are used exclusively on nine of the top 10 supercomputers (June 2016 ranking), including the world's fastest supercomputer: Taihu Light.

Download MPICH

**R&D 100**

NEWS & EVENTS

**MPICH 3.3a2 released**  
A new preview release of MPICH, 3.3a2, is now available for download.

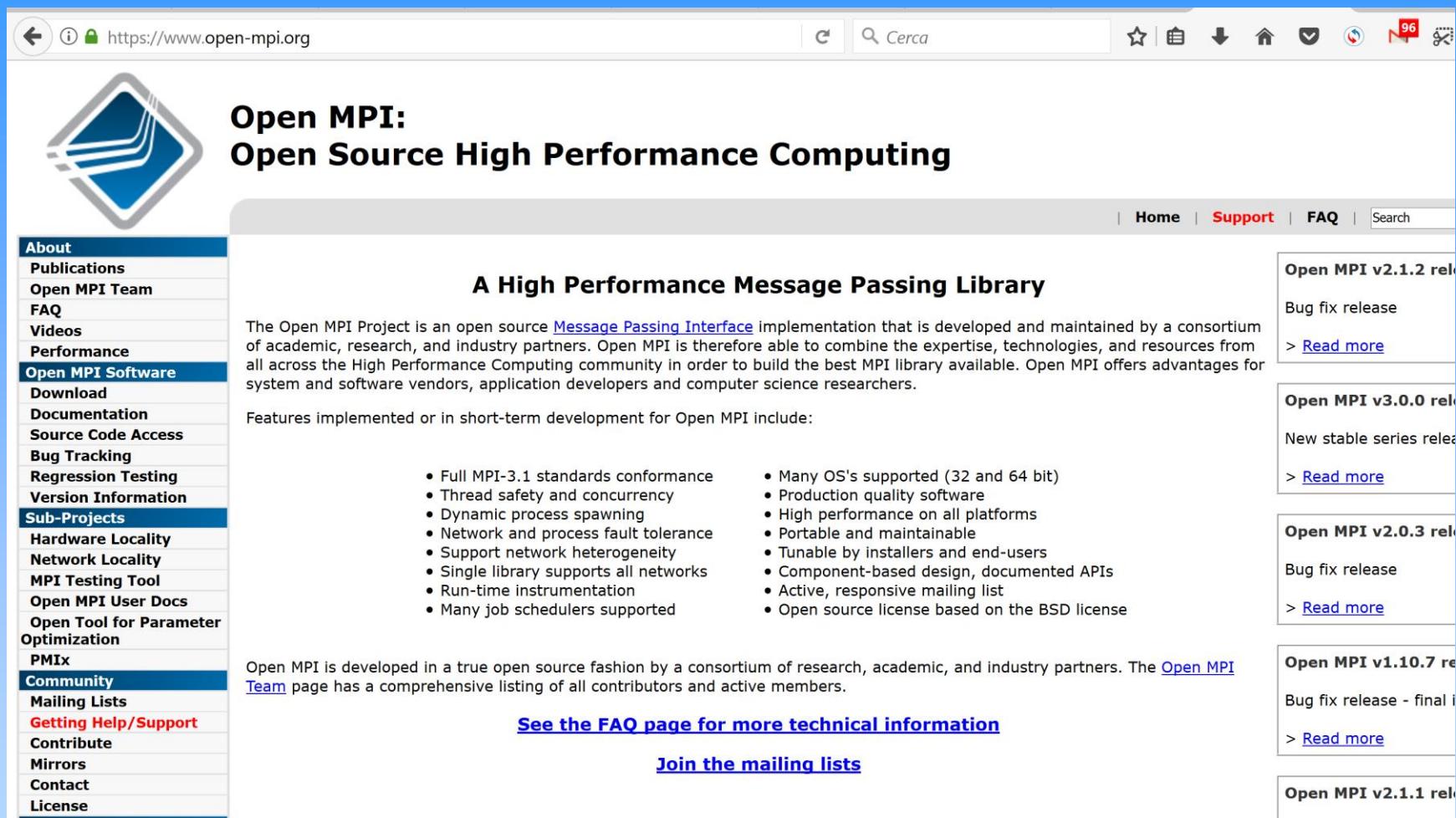
LEARN ABOUT MPICH

[The documentation page](#) provides documents for installing MPICH, how to get started with MPI, and how to

SUPPORT

[The support page](#) provides help for MPICH users and developers. There are links to frequently asked

<http://www.mpich.org/>



The screenshot shows the homepage of the Open MPI website (<https://www.open-mpi.org>). The page features a large logo on the left and a navigation menu on the right. The main content area includes a section about MPI features and a sidebar with release notes for various versions of Open MPI.

**Open MPI:**  
**Open Source High Performance Computing**

**A High Performance Message Passing Library**

The Open MPI Project is an open source [Message Passing Interface](#) implementation that is developed and maintained by a consortium of academic, research, and industry partners. Open MPI is therefore able to combine the expertise, technologies, and resources from all across the High Performance Computing community in order to build the best MPI library available. Open MPI offers advantages for system and software vendors, application developers and computer science researchers.

Features implemented or in short-term development for Open MPI include:

- Full MPI-3.1 standards conformance
- Thread safety and concurrency
- Dynamic process spawning
- Network and process fault tolerance
- Support network heterogeneity
- Single library supports all networks
- Run-time instrumentation
- Many job schedulers supported
- Many OS's supported (32 and 64 bit)
- Production quality software
- High performance on all platforms
- Portable and maintainable
- Tunable by installers and end-users
- Component-based design, documented APIs
- Active, responsive mailing list
- Open source license based on the BSD license

Open MPI is developed in a true open source fashion by a consortium of research, academic, and industry partners. The [Open MPI Team](#) page has a comprehensive listing of all contributors and active members.

[See the FAQ page for more technical information](#)

[Join the mailing lists](#)

**About**  
Publications  
Open MPI Team  
FAQ  
Videos  
Performance  
**Open MPI Software**  
Download  
Documentation  
Source Code Access  
Bug Tracking  
Regression Testing  
Version Information  
**Sub-Projects**  
Hardware Locality  
Network Locality  
MPI Testing Tool  
Open MPI User Docs  
Open Tool for Parameter Optimization  
PMIx  
**Community**  
Mailing Lists  
Getting Help/Support  
Contribute  
Mirrors  
Contact  
License

| Home | **Support** | **FAQ** | Search

**Open MPI v2.1.2 release**  
Bug fix release  
> [Read more](#)

**Open MPI v3.0.0 release**  
New stable series released  
> [Read more](#)

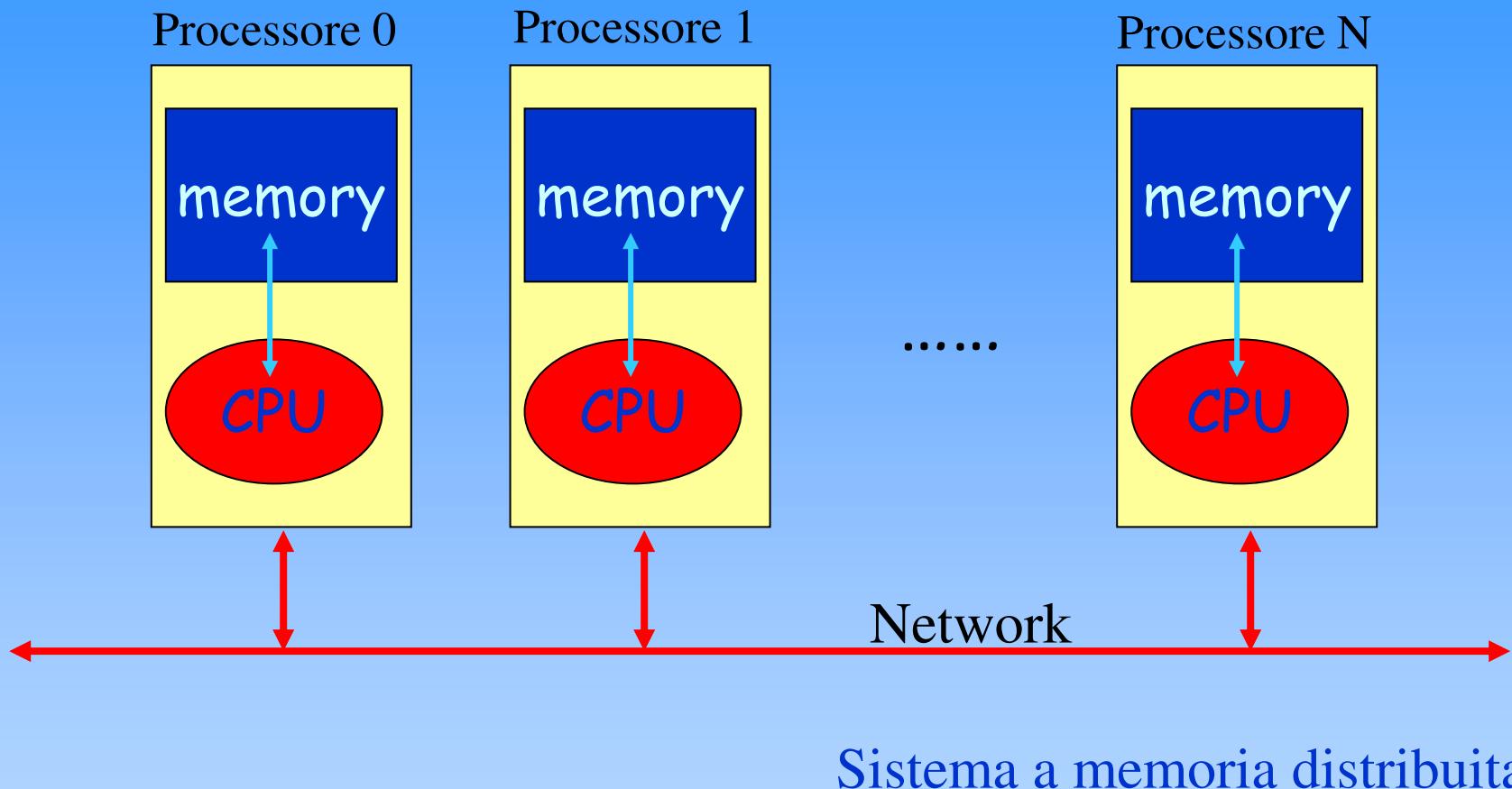
**Open MPI v2.0.3 release**  
Bug fix release  
> [Read more](#)

**Open MPI v1.10.7 release**  
Bug fix release - final i  
> [Read more](#)

**Open MPI v2.1.1 release**

<https://www.open-mpi.org/>

# Paradigma del message passing

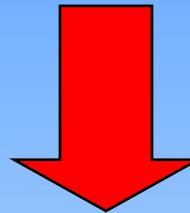


Ogni processore ha una propria memoria locale alla quale accede direttamente.

Ogni processore può conoscere i dati in memoria di un altro processore o far conoscere i propri, attraverso il trasferimento di dati.

# Definizione: message passing

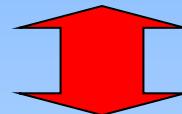
... ogni processore può conoscere i dati nella memoria di un altro processore o far conoscere i propri, attraverso il trasferimento di dati.



*Message Passing :*  
modello per la progettazione  
di software in  
ambiente di Calcolo Parallelo.

# Lo standard

La necessità di **rendere portabile**  
il modello *Message Passing*  
ha condotto alla definizione  
e all'implementazione  
di un **ambiente standard**.



**Message Passing Interface**  
**MPI**

# Per cominciare...

```
...  
int main()  
{  
  
    ...  
    sum=0;  
    for i= 0 to 14 do  
        sum=sum+a[i];  
    endfor  
  
    ...  
    return 0;  
}
```

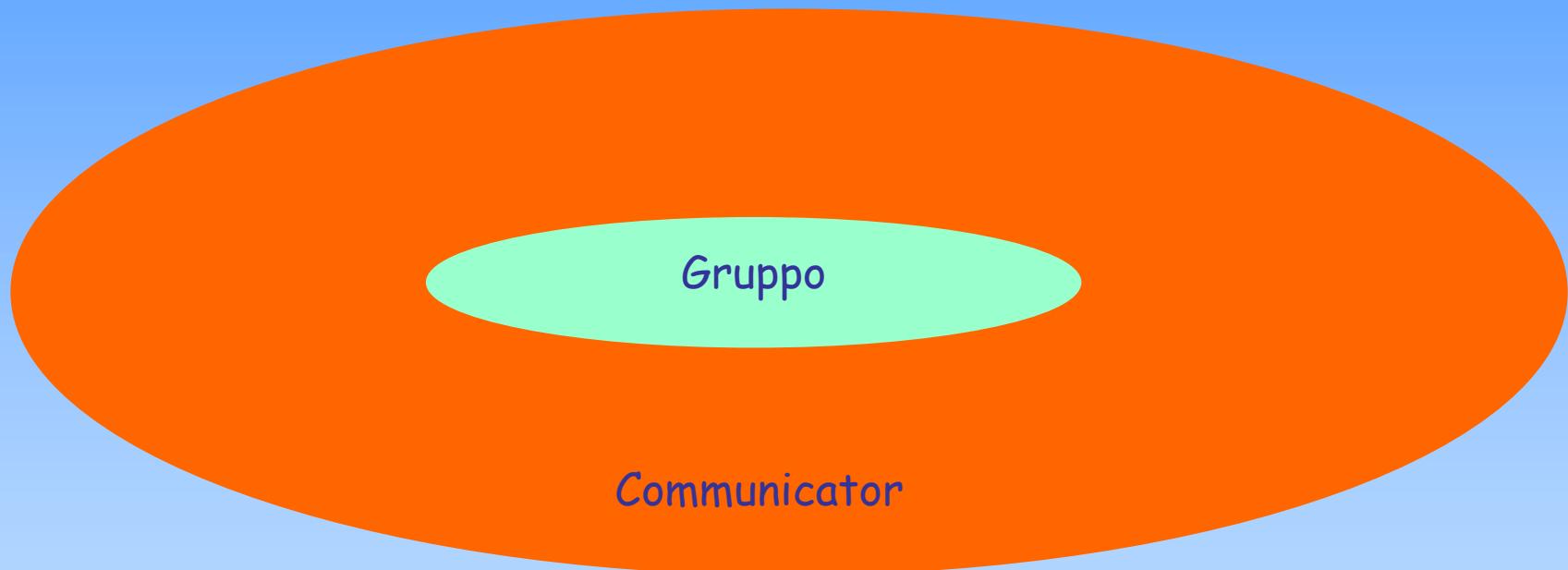
In ambiente MPI un programma  
è visto come un insieme di  
**componenti (o processi) concorrenti**

```
...  
main()  
{  
  
    ...  
    sum=0;  
    for i=0 to 4 do  
        sum=sum+a[i];  
    endfor  
  
    ...  
    return 0;  
}
```

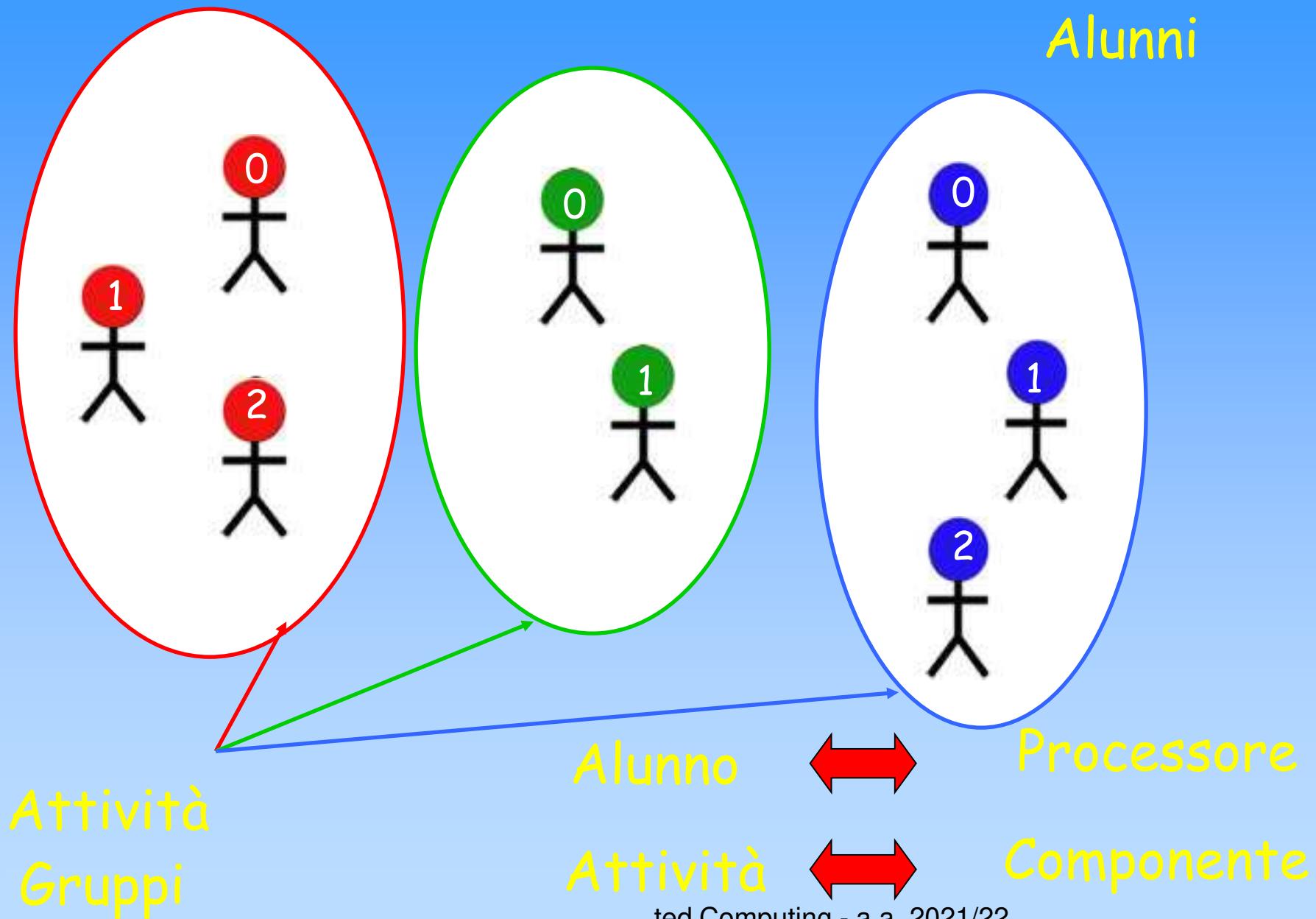
```
...  
main()  
{  
  
    ...  
    sum=0;  
    for i=5 to 9 do  
        sum=sum+a[i];  
    endfor  
  
    ...  
    return 0;  
}
```

```
...  
main()  
{  
  
    ...  
    sum=0;  
    for i=10 to 14do  
        sum=sum+a[i];  
    endfor  
  
    ...  
    return 0;  
}
```

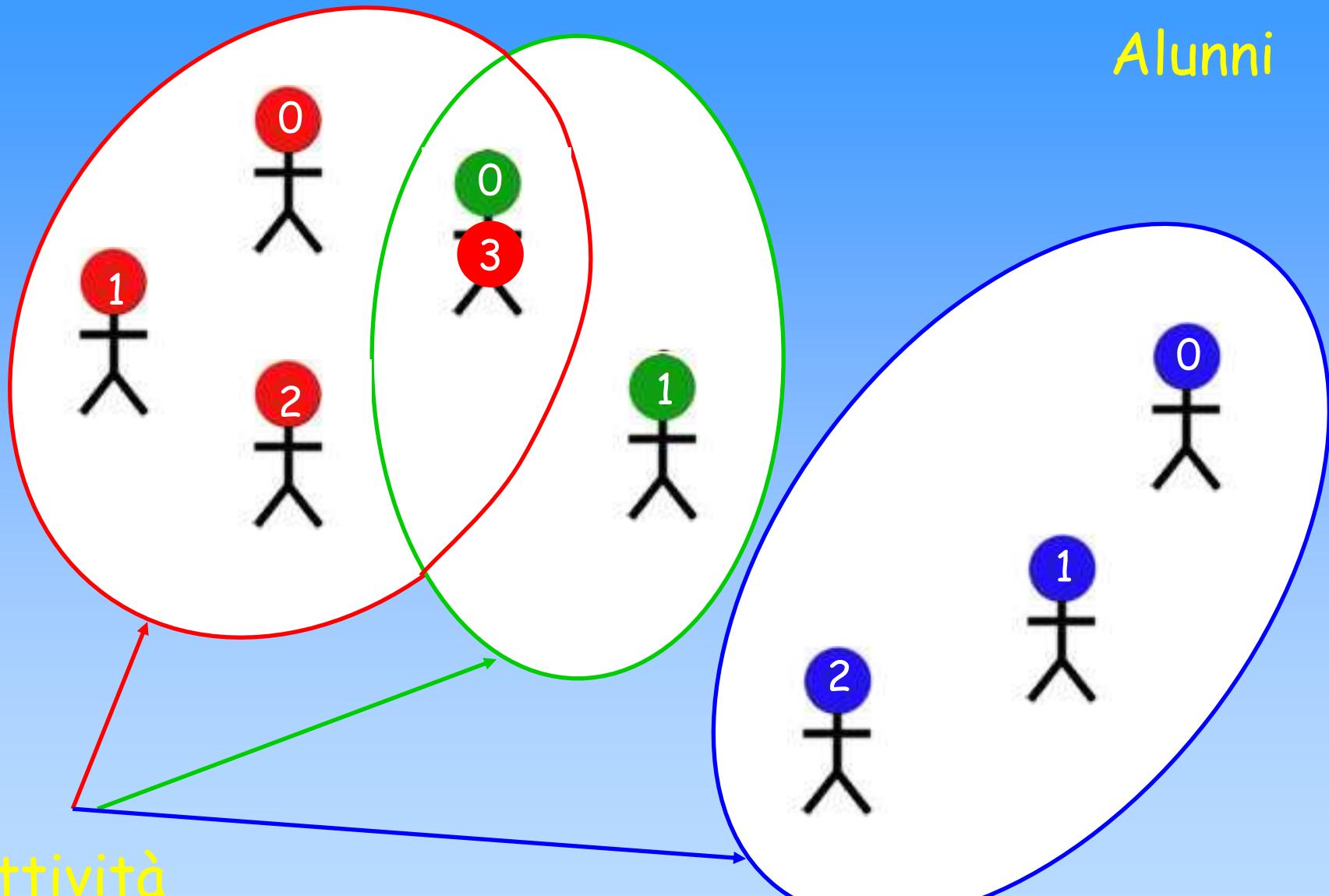
*In MPI i processori sono raggruppati in ....:*



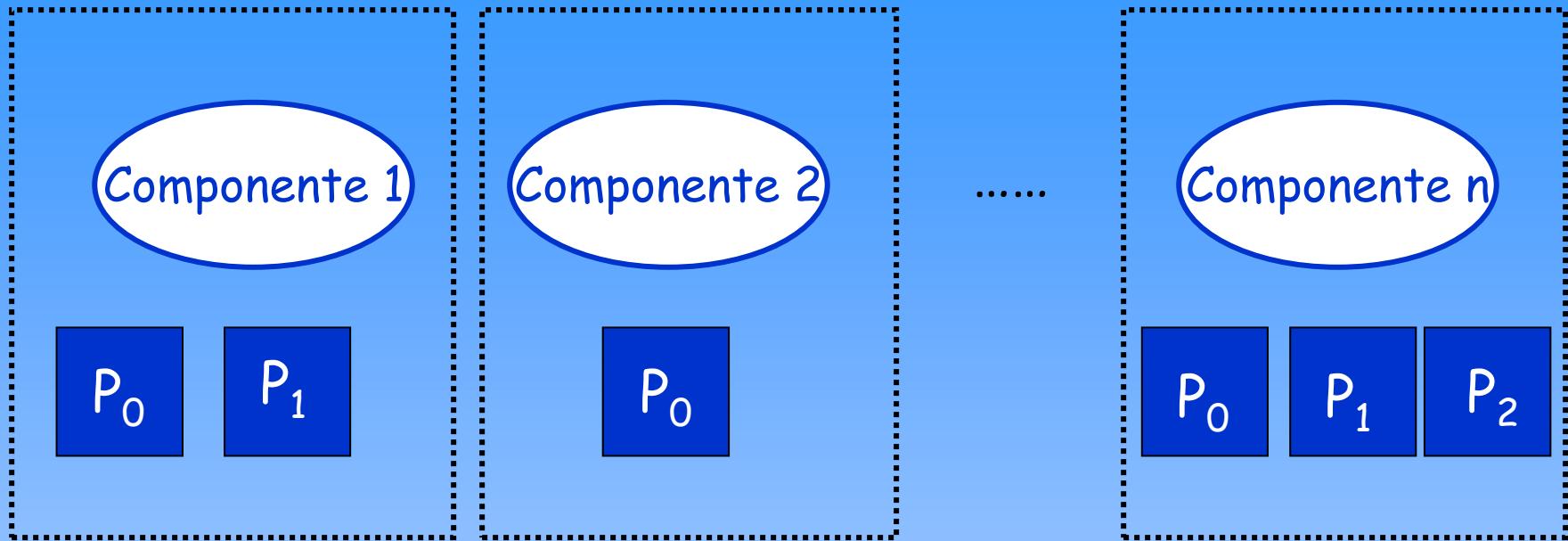
# Il gruppo MPI ... :



# Il gruppo MPI ... :

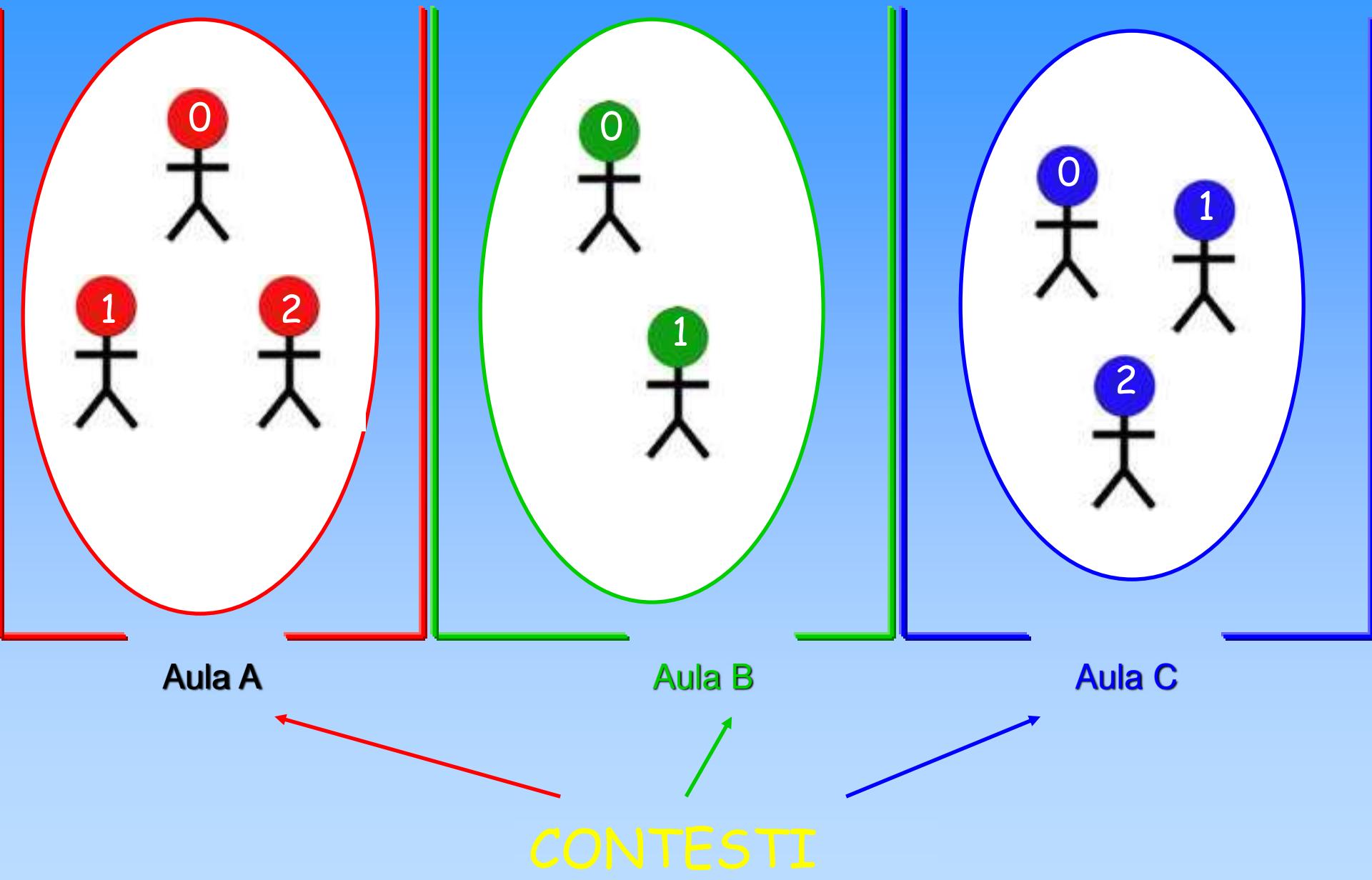


# Alcuni concetti di base: GRUPPO...

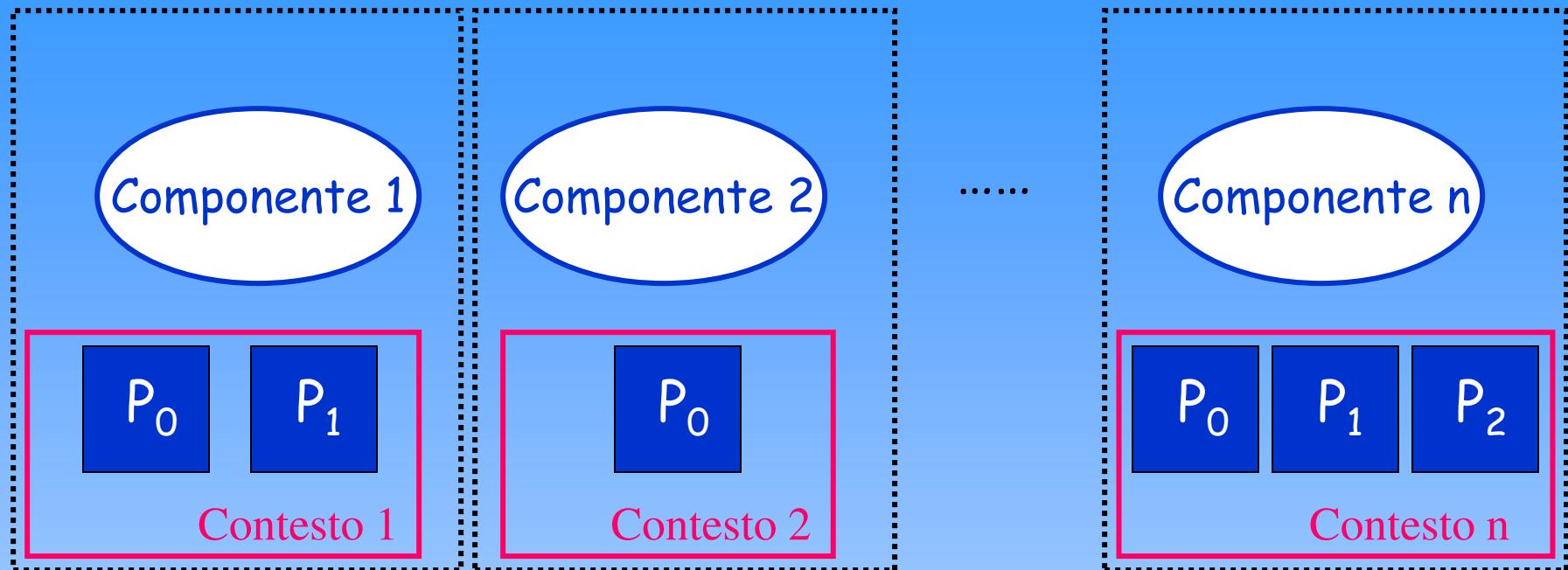


- Ogni componente concorrente del programma viene affidata ad un gruppo di processori.
- In ciascun gruppo ad ogni processore è associato un *identificativo*.
- L'*identificativo* è un intero, compreso tra 0 ed il numero totale di processori appartenenti ad un gruppo decrementato di una unità.
- Processori appartenenti a uno o più gruppi possono avere identificativi diversi, ciascuno relativo ad uno specifico gruppo

# IL CONTESTO MPI :

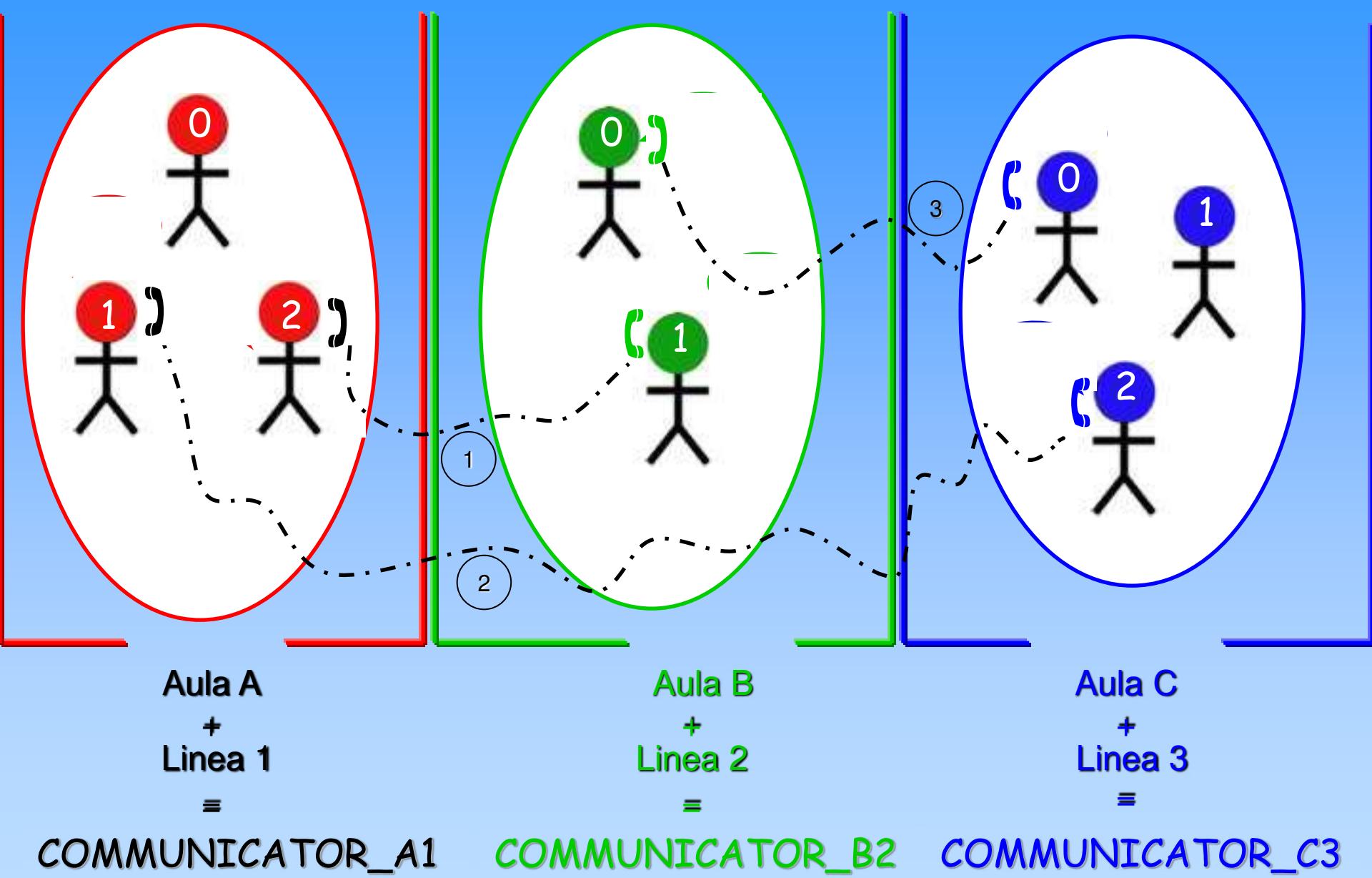


# Alcuni concetti di base: CONTESTO

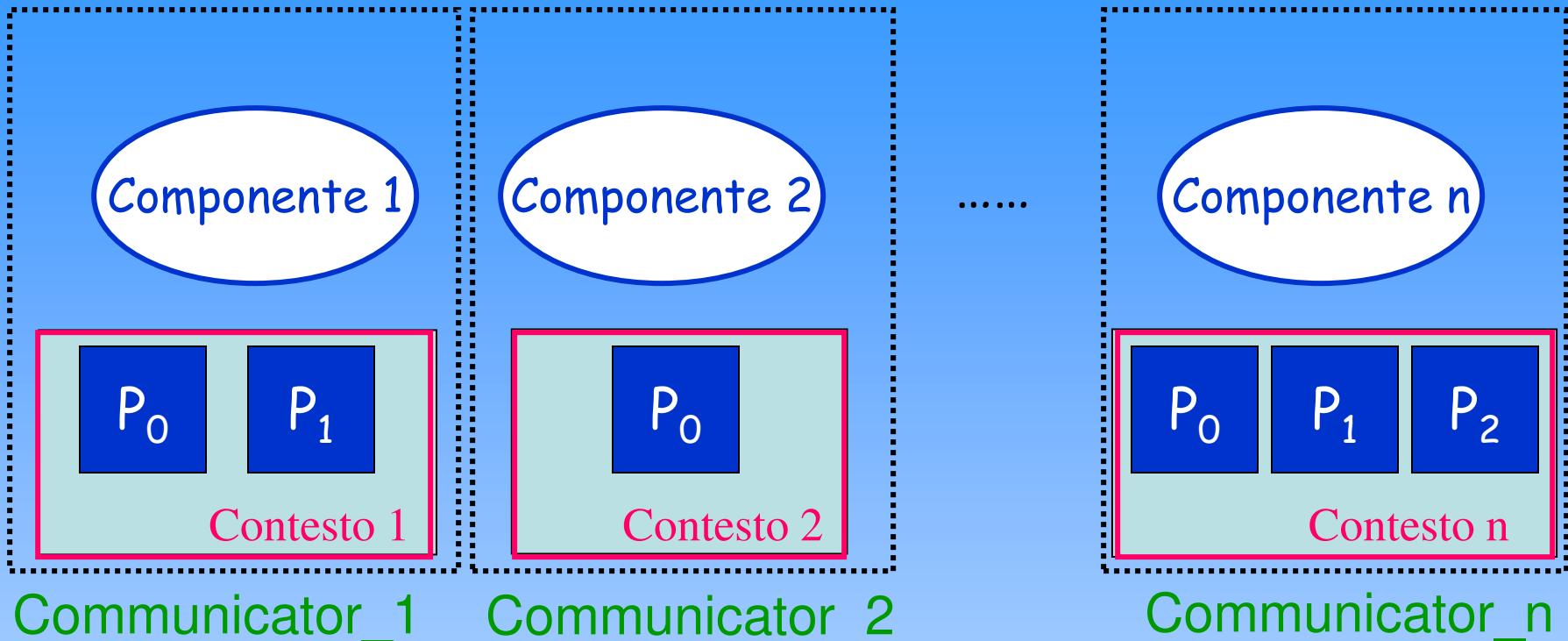


- A ciascun gruppo di processori viene attribuito un identificativo detto **contesto**.
- Il **contesto** definisce l'ambito in cui avvengono le comunicazioni tra processori di uno stesso gruppo.
- Se la spedizione di un messaggio avviene in un **contesto**, la ricezione deve avvenire nello stesso **contesto**.

# Il Communicator MPI :

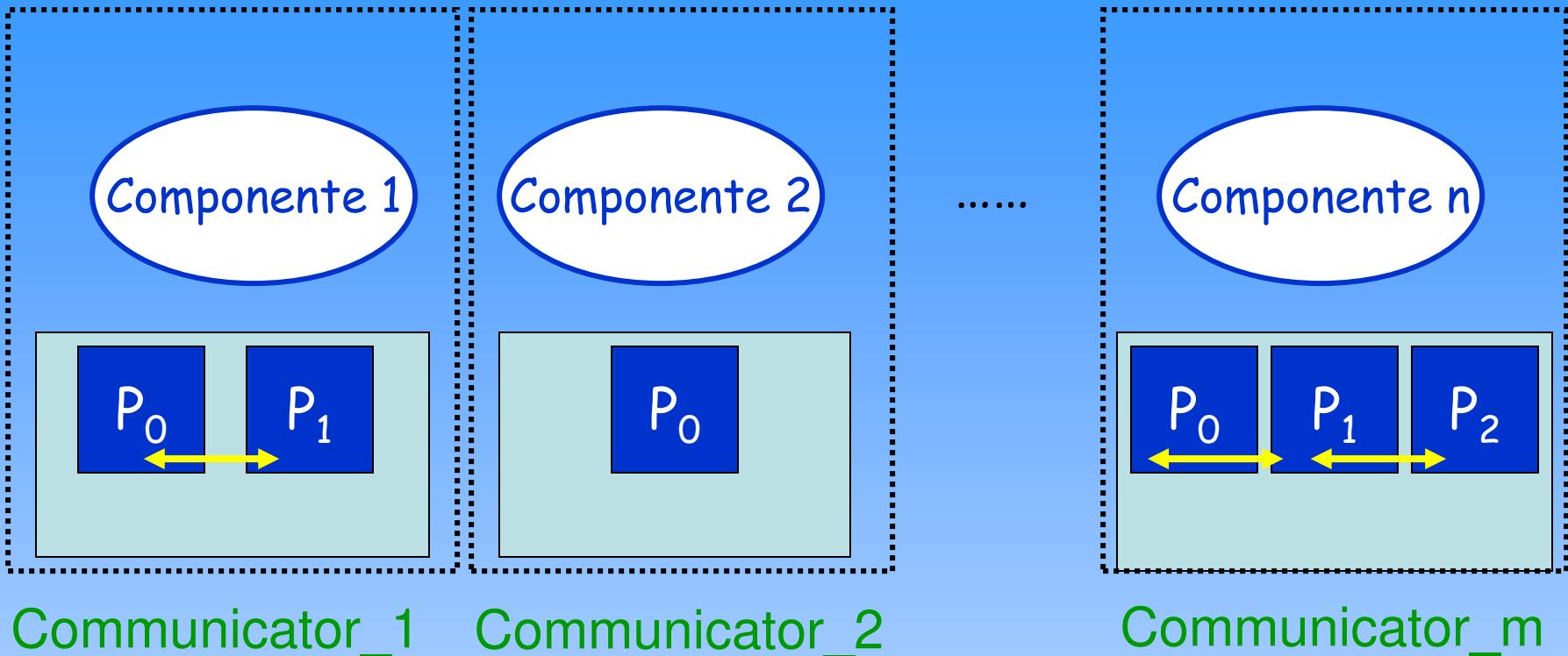


# Alcuni concetti di base: COMMUNICATOR...



- Ad un gruppo di processori appartenenti ad uno stesso contesto viene assegnato un ulteriore identificativo: il *communicator*.
- Il *communicator* racchiude tutte le caratteristiche dell'ambiente di comunicazione: topologia, quali contesti coinvolge, ecc...

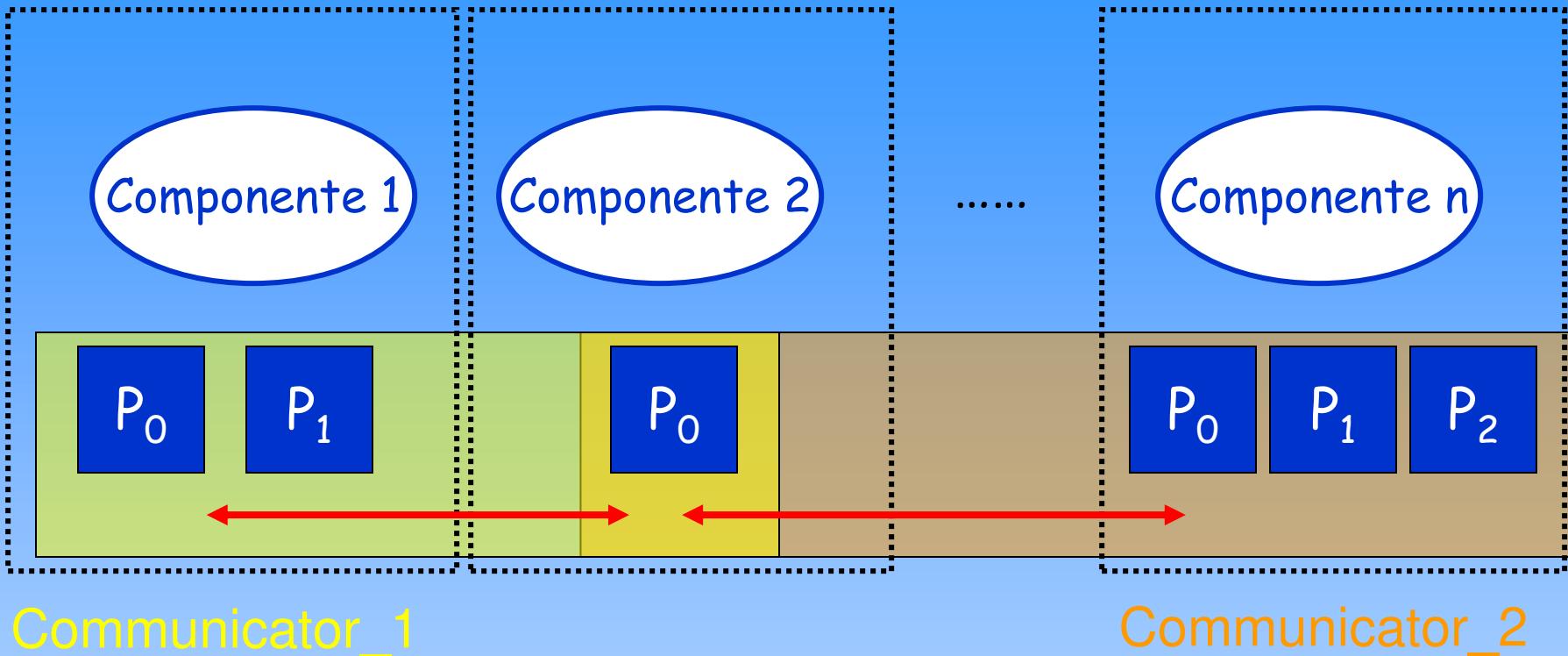
# ...Alcuni concetti di base: COMMUNICATOR...



- Esistono due tipi principali di *communicator*.

- L'*intra-communicator* in cui le comunicazioni avvengono all'interno di un gruppo di processori.

# ...Alcuni concetti di base: COMMUNICATOR...

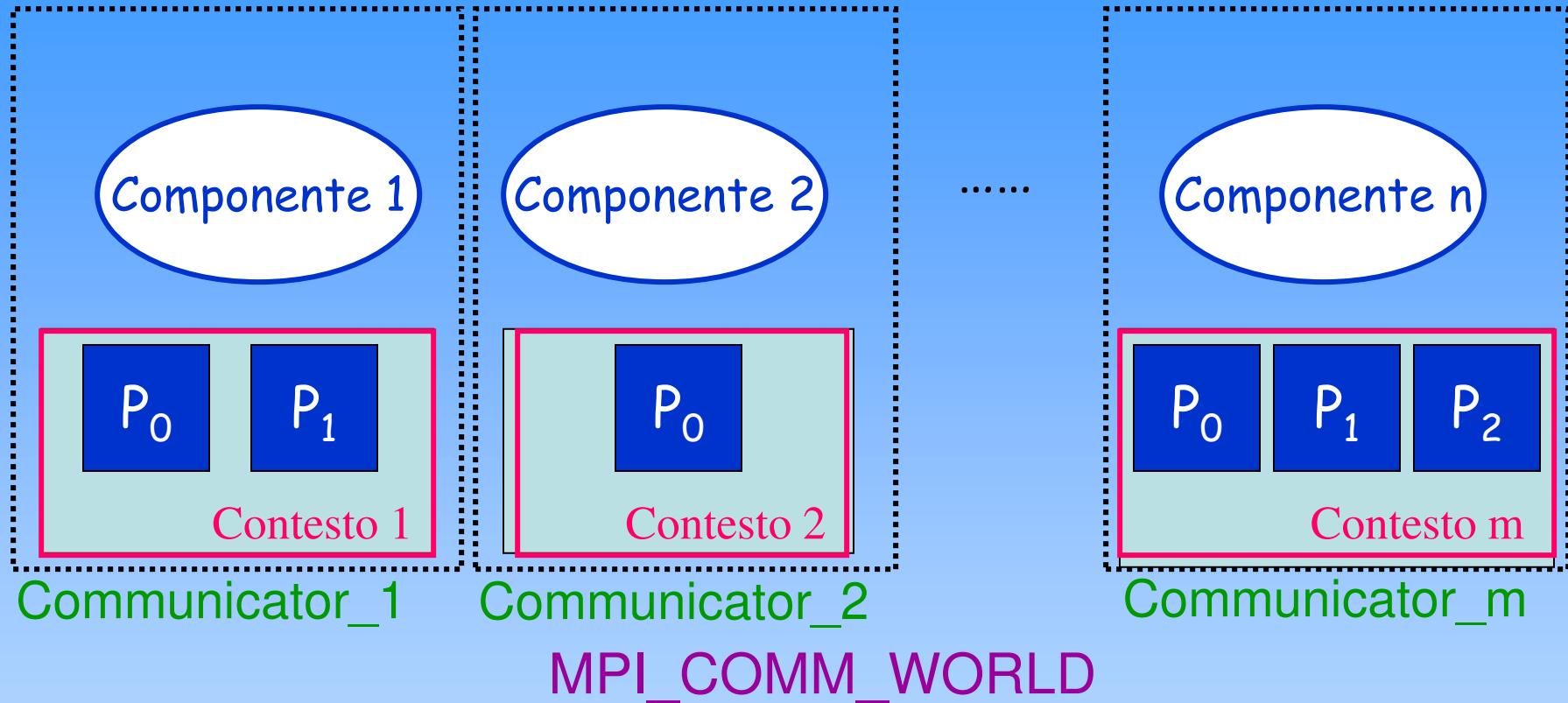


- Esistono due tipi principali di *communicator*.

- L'*intra-communicator* in cui le comunicazioni avvengono all'interno di un gruppo di processori.

- L'*inter-communicator* in cui le comunicazioni avvengono tra gruppi di processori.

# ...Alcuni concetti di base: COMMUNICATOR



- Tutti i processori fanno parte per default di un unico communicator detto **MPI\_COMM\_WORLD**.

# Le funzioni di MPI

MPI è una libreria  
che comprende:

- Funzioni per definire l'ambiente
- Funzioni per comunicazioni uno a uno
- Funzioni per comunicazioni collettive
- Funzioni per operazioni collettive

MPI :

Le funzioni dell'ambiente.

# Un semplice programma :

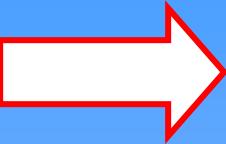
```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    int menum, nproc;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&menu);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);

    printf("Sono %d di %d\n",menu,nproc);
    MPI_Finalize();
    return 0;
}
```

Tutti i processori di MPI\_COMM\_WORLD  
stampano a video il proprio  
identificativo menum ed il numero di processori nproc.

Nel programma ... :



```
#include "mpi.h"
```

mpi.h : Header File

Il file contiene alcune direttive necessarie al preprocessore per l'utilizzo dell'MPI

Nel programma ... :

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    int menum, nproc;
    MPI_Init(&argc,&argv);

    MPI_Comm_rank(MPI_COMM_WORLD,&menum);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);

    printf("Sono %d di %d\n",menum,nproc);

    MPI_Finalize();

    return 0;
}
```



Tutti i processori di MPI\_COMM\_WORLD  
stampano a video il proprio  
identificativo menum ed il numero di processori nproc.

Nel programma ... :



```
MPI_Init(&argc,&argv);
```

- Inizializza l'ambiente di esecuzione MPI
- Inizializza il communicator **MPI\_COMMON\_WORLD**
- I due dati di input: argc ed argv sono gli argomenti del main

## In generale ... :

```
MPI_Init(int *argc, char ***argv);
```

Input: argc, \*\*argv;

- Questa routine inizializza l'ambiente di esecuzione di MPI. Deve essere chiamata una sola volta, prima di ogni altra routine MPI.
- Definisce l'insieme dei processori attivati (communicator).

Nel programma ... :

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    int menum, nproc;
    MPI_Init(&argc,&argv);

    MPI_Comm_rank(MPI_COMM_WORLD,&menum);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);

    printf("Sono %d di %d\n",menum,nproc);

    MPI_Finalize();

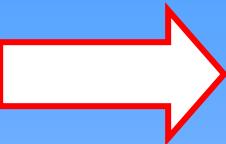
    return 0;
}
```



Tutti i processori di MPI\_COMM\_WORLD  
stamperanno sul video il proprio  
identificativo menum ed il numero di processori nproc.

Nel programma ... :

---



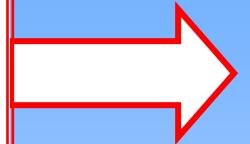
## MPI\_Finalize();

- Questa routine determina la fine del programma MPI.
- Dopo questa routine non è possibile richiamare nessun'altra routine di MPI.

Nel programma ... :

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    int menum, nproc;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&menu);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    printf("Sono %d di %d\n",menu,nproc);
    MPI_Finalize();
    return 0;
}
```



Tutti i processori di MPI\_COMM\_WORLD  
stamperanno sul video il proprio  
identificativo menum ed il numero di processori nproc.

Nel programma ... :



```
MPI_Comm_rank(MPI_COMM_WORLD,&menu);
```

- Questa routine permette al processore chiamante, appartenente al communicator MPI\_COMM\_WORLD, di memorizzare il proprio identificativo nella variabile menu.

## In generale ... :

---

```
MPI_Comm_rank(MPI_Comm comm, int *menum);
```

Input: comm ;  
Output: menum.

- Fornisce ad ogni processore del comunicator comm l'identificativo menum.

Nel programma ... :

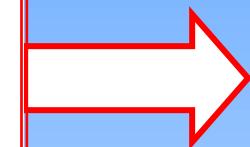
```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    int menum, nproc;
    MPI_Init(&argc,&argv);

    MPI_Comm_rank(MPI_COMM_WORLD,&menu);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);

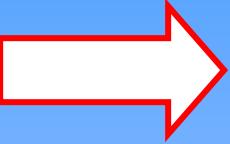
    printf("Sono %d di %d\n",menu,nproc);

    MPI_Finalize();
    return 0;
}
```



Tutti i processori di MPI\_COMM\_WORLD  
stamperanno sul video il proprio  
identificativo menum ed il numero di processori nproc.

Nel programma ... :



```
MPI_Comm_size(MPI_COMM_WORLD,&nproc);
```

- Questa routine permette al processore chiamante di memorizzare nella variabile nproc il numero totale dei processori concorrenti appartenenti al communicator **MPI\_COMM\_WORLD**.

## In generale ... :

```
MPI_Comm_size(MPI_Comm comm, int *nproc);
```

Input: comm ;

Output: nproc.

- Ad ogni processore del communicator `comm` , restituisce in `nproc`, il numero totale di processori che costituiscono `comm`.
- Permette di conoscere quanti processori concorrenti possono essere utilizzati per una determinata operazione.

# Eseguire un programma: esempio di job-script

```
#!/bin/bash

#####
#          #
# The PBS directives  #
#          #
#####
#PBS -q studenti
#PBS -l nodes=8
#PBS -N hello
#PBS -o hello.out
#PBS -e hello.err
#####
# -q coda su cui va eseguito il job #

# -l numero di nodi richiesti #
# -N nome job(stesso del file pbs) #
# -o, -e nome files contenente l'output #
#####
#          #
# qualche informazione sul job   #
#          #
#####

NCPU=`wc -l < $PBS_NODEFILE`
echo -----
echo ' This job is allocated on '${NCPU}' cpu(s)'
echo 'Job is running on node(s):'
cat $PBS_NODEFILE
```

Job-script che compila  
ed esegue Hello.c

# Eseguire un programma: esempio di job-script

```
#!/bin/bash  
#####  
# #  
# The PBS directives #  
# #  
#####  
#PBS -q studenti  
#PBS -l nodes=8  
#PBS -N Hello  
#PBS -o Hello.out  
#PBS -e Hello.err  
#####  
# -q coda su cui va eseguito il job #  
  
# -l numero di nodi richiesti #  
# -N nome job(stesso del file pbs) #  
# -o, -e nome files contenente l'output #  
#####  
# #  
# qualche informazione sul job #  
# #  
#####  
  
NCPU=`wc -l < $PBS_NODEFILE`  
echo -----  
echo ' This job is allocated on '${NCPU}' cpu(s)'  
echo 'Job is running on node(s):'  
cat $PBS_NODEFILE
```

La prima riga deve essere sempre questa

Le righe che iniziano con **#** sono commenti.  
Le righe che iniziano con **#PBS** sono direttive.

In rosso le informazioni tipiche di questa particolare sottomissione

# Eseguire un programma: esempio di job-script

```
PBS_O_WORKDIR=$PBS_O_HOME/ProgettoHello
echo -----
echo PBS: qsub is running on $PBS_O_HOST
echo PBS: originating queue is $PBS_O_QUEUE
echo PBS: executing queue is $PBS_QUEUE
echo PBS: working directory is $PBS_O_WORKDIR
echo PBS: execution mode is $PBS_ENVIRONMENT
echo PBS: job identifier is $PBS_JOBID
echo PBS: job name is $PBS_JOBNAME
echo PBS: node file is $PBS_NODEFILE
echo PBS: current home directory is $PBS_O_HOME
echo PBS: PATH = $PBS_O_PATH
echo -----
echo "Eseguo/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR>Hello $PBS_O_WORKDIR>Hello.c"
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR>Hello $PBS_O_WORKDIR>Hello.c

echo "Eseguo:/usr/lib64/openmpi/1.4-gcc/bin/-machinefile $PBS_NODEFILE E -np $NCPU $PBS_O_WORKDIR>Hello"
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPU $PBS_O_WORKDIR>Hello
```

Stampa informazioni

# Eseguire un programma: esempio di job-script

```
PBS_O_WORKDIR=$PBS_O_HOME/ProgettoHello
echo -----
echo PBS: qsub is running on $PBS_O_HOST
echo PBS: originating queue is $PBS_O_QUEUE
echo PBS: executing queue is $PBS_QUEUE
echo PBS: working directory is $PBS_O_WORKDIR
echo PBS: execution mode is $PBS_ENVIRONMENT
echo PBS: job identifier is $PBS_JOBID
echo PBS: job name is $PBS_JOBNAME
echo PBS: node file is $PBS_NODEFILE
echo PBS: current home directory is $PBS_O_HOME
echo PBS: PATH = $PBS_O_PATH
echo -----
echo "Eseguo/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR>Hello $PBS_O_WORKDIR>Hello.c"
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR>Hello $PBS_O_WORKDIR>Hello.c

echo "Eseguo/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPU
$PBS_O_WORKDIR>Hello"
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPU $PBS_O_WORKDIR>Hello
```

Stampa informazioni

# Eseguire un programma: esempio di job-script

Vediamo nel dettaglio la compilazione e l'esecuzione...

```
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR>Hello $PBS_O_WORKDIR>Hello.c
```

Path assoluto del comando di compilazione (mpicc) e di esecuzione (mpiexec)

```
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPU $PBS_O_WORKDIR>Hello
```

# Eseguire un programma: esempio di job-script

Vediamo nel dettaglio la compilazione e l'esecuzione...

```
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR>Hello $PBS_O_WORKDIR>Hello.c
```



Opzione -o, ben nota!

```
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPU $PBS_O_WORKDIR>Hello
```

# Eseguire un programma: esempio di job-script

Vediamo nel dettaglio la compilazione e l'esecuzione...

```
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR>Hello $PBS_O_WORKDIR>Hello.c
```

Nome dell'eseguibile che si vuole creare (path assoluto)

```
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPU $PBS_O_WORKDIR>Hello
```

# Eseguire un programma: esempio di job-script

Vediamo nel dettaglio la compilazione e l'esecuzione...

```
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR>Hello $PBS_O_WORKDIR>Hello.c
```

Nome del sorgente che si vuole compilare (path assoluto)

```
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPU $PBS_O_WORKDIR>Hello
```

# Eseguire un programma: esempio di job-script

Vediamo nel dettaglio la compilazione e l'esecuzione...

```
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR>Hello $PBS_O_WORKDIR>Hello.c
```

Elenco delle macchine utilizzabili

```
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPU $PBS_O_WORKDIR>Hello
```

# Eseguire un programma: esempio di job-script

Vediamo nel dettaglio la compilazione e l'esecuzione...

```
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR>Hello $PBS_O_WORKDIR>Hello.c
```

Numero di processi che si devono lanciare

```
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPU $PBS_O_WORKDIR>Hello
```

# Eseguire un programma: esempio di job-script

Vediamo nel dettaglio la compilazione e l'esecuzione...

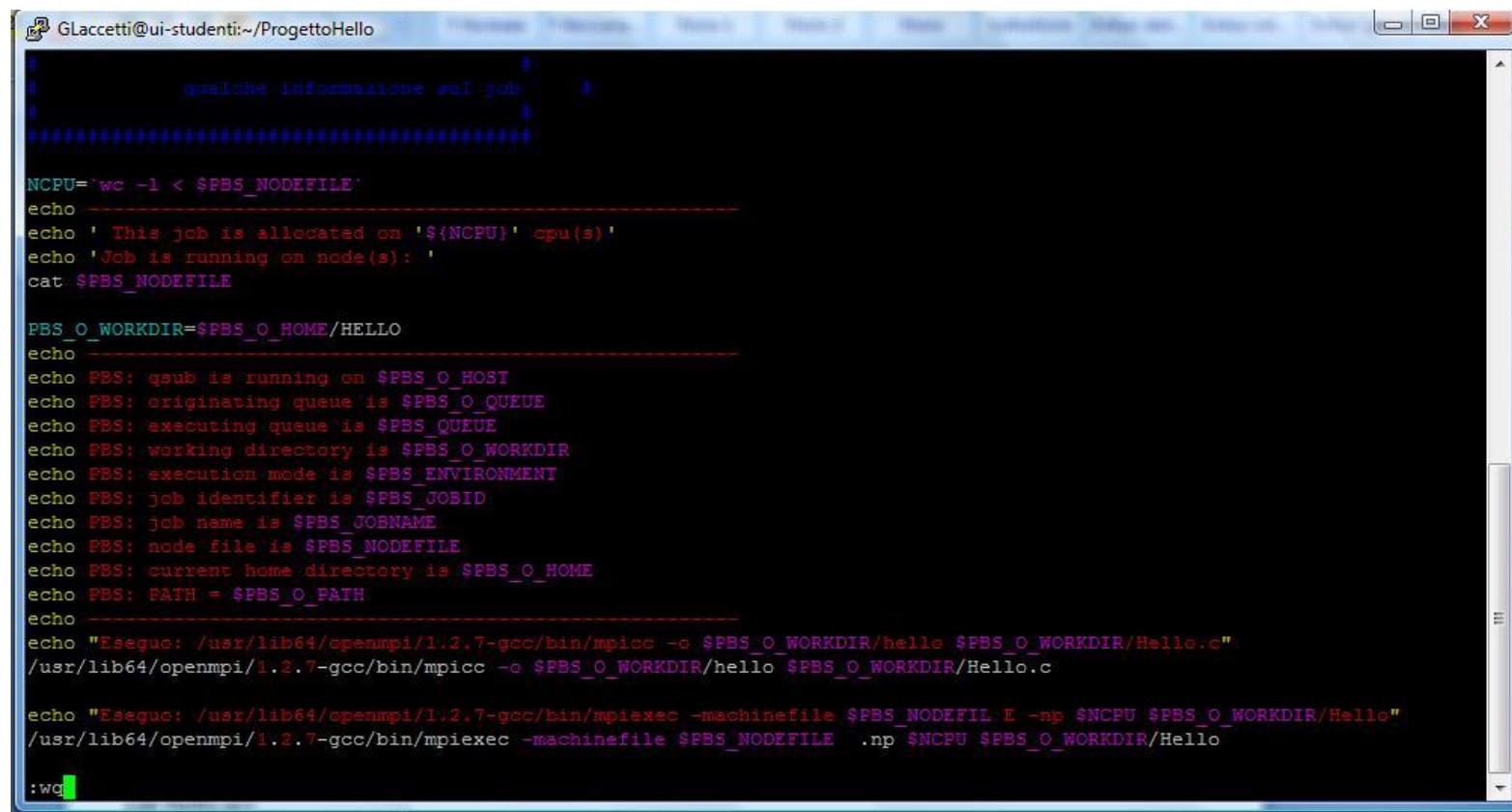
```
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR>Hello $PBS_O_WORKDIR>Hello.c
```

Esegibile da lanciare

```
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPU $PBS_O_WORKDIR>Hello
```

# Eseguire un programma

Una volta che si è scritto il file PBS



The screenshot shows a terminal window titled "Glaccetti@ui-studenti:~/ProgettoHello". The window displays a shell script being run. The script starts with a header section printing job information and node allocation details. It then sets the working directory to \$PBS\_O\_HOME/HELLO and prints PBS environment variables. Finally, it executes two commands: mpicc to compile a C program and mpiexec to run it with multiple processes.

```
Glaccetti@ui-studenti:~/ProgettoHello
$ qualche informazione sul job
$ NCPUs=`wc -l < $PBS_NODEFILE`
echo -
echo ' This job is allocated on '$NCPUs' cpu(s)'
echo 'Job is running on node(s):'
cat $PBS_NODEFILE

PBS_O_WORKDIR=$PBS_O_HOME/HELLO
echo -
echo PBS: qsub is running on $PBS_O_HOST
echo PBS: originating queue is $PBS_O_QUEUE
echo PBS: executing queue is $PBS_QUEUE
echo PBS: working directory is $PBS_O_WORKDIR
echo PBS: execution mode is $PBS_ENVIRONMENT
echo PBS: job identifier is $PBS_JOBID
echo PBS: job name is $PBS_JOBNAME
echo PBS: node file is $PBS_NODEFILE
echo PBS: current home directory is $PBS_O_HOME
echo PBS: PATH = $PBS_O_PATH
echo -
echo "Esegue: /usr/lib64/openmpi/1.2.7-gcc/bin/mpicc -o $PBS_O_WORKDIR/hello $PBS_O_WORKDIR/Hello.c"
/usr/lib64/openmpi/1.2.7-gcc/bin/mpicc -o $PBS_O_WORKDIR/hello $PBS_O_WORKDIR/Hello.c

echo "Esegue: /usr/lib64/openmpi/1.2.7-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPUs $PBS_O_WORKDIR>Hello"
/usr/lib64/openmpi/1.2.7-gcc/bin/mpiexec -machinefile $PBS_NODEFILE .np $NCPUs $PBS_O_WORKDIR>Hello

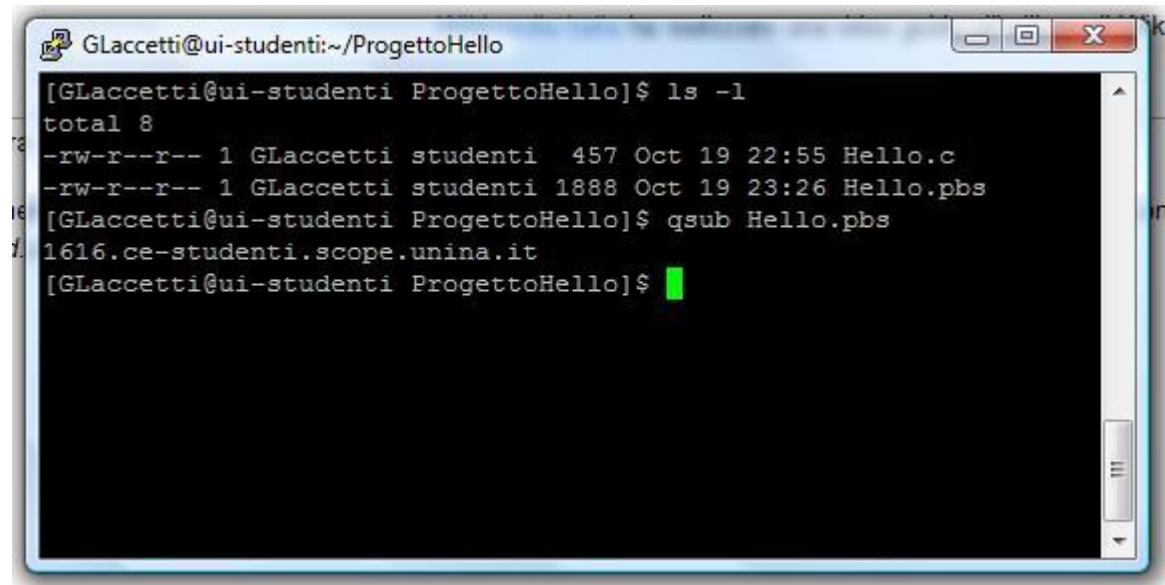
:wq
```

# Eseguire un programma

Una volta che si è scritto il file PBS, e si è quindi preparata l'esecuzione parallela, si lancia con il comando

**qsub Hello.pbs**

qsub è in grado di interpretare le direttive contenute nel PBS



The screenshot shows a terminal window titled "GLaccetti@ui-studenti:~/ProgettoHello". The user has run the command "ls -l" which lists the contents of the directory: "total 8", "Hello.c", and "Hello.pbs". The user then runs "qsub Hello.pbs" which triggers the submission of the PBS script. The terminal prompt "[GLaccetti@ui-studenti ProgettoHello]\$" is visible at the bottom.

# Eseguire un programma

---

Una volta che si è scritto il file PBS, e si è quindi preparata l'esecuzione parallela, si lancia con il comando

**qsub Hello.pbs**

qsub è in grado di interpretare le direttive contenute nel PBS

bisognerà aspettare che il job venga gestito dal sistema (è in coda con altri job) e che termini la sua esecuzione.

A questo punto sarà possibile visualizzare l'output e l'error con i comandi:

**cat Hello.err**

**cat Hello.out**

# Eseguire un programma

---

Per visualizzare i job:

**qstat**

Per visualizzare la coda ed il loro stato

(E=eseguito,R=in esecuzione, Q= è stato accodato):

**qstat -q**

Per eliminare un job dalla coda:

**qdel jobid**

# Esercizi

Scrivere, compilare ed eseguire sul cluster attraverso il PBS i seguenti esercizi.  
Inviare il codice, il PBS e uno o più esempi di output a **valeria.mele@unina.it**

1. N processi. Ognuno stampa il proprio identificativo nel communicator globale
  
2. N processi. In input: intero M>200.  
Ogni processo con identificativo >0 divide M per il proprio identificativo. Il processo 0 divide M per il numero di processi N.  
Ogni processo stampa il risultato ottenuto.



Entro una settimana

---

# **FINE LEZIONE**

# Parallel and Distributed Computing

Prodotto Matrice-Matrice

---

Prof. G. Laccetti

a.a. 2021-2022

# Problema

---

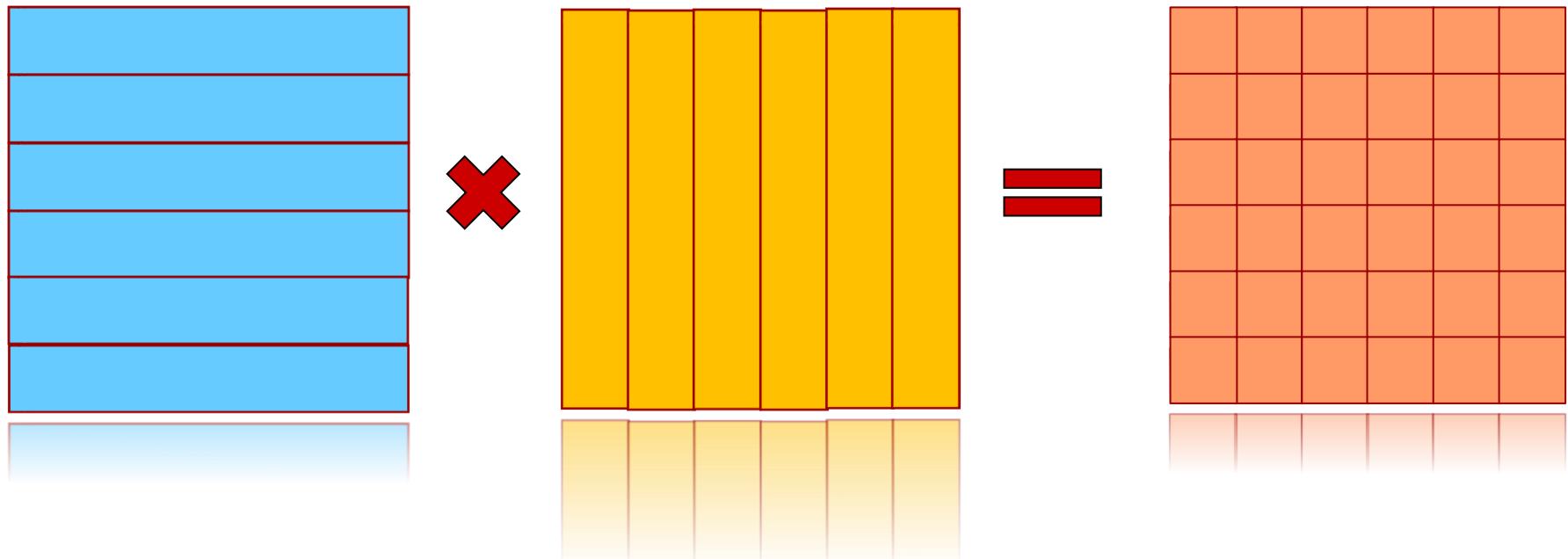
Progettazione  
di un algoritmo parallelo  
per architettura **MIMD**  
a memoria distribuita  
per il calcolo del prodotto righe per colonne  
di 2 matrici  $A$  e  $B$ :

$$C = A \bullet B, \quad A, B \in \mathbb{R}^{n \times n}$$

# Quali sono i sotto-problemi indipendenti?

---

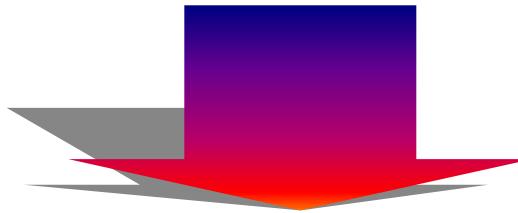
Gli elementi di  $C$  sono  
calcolati effettuando i  
prodotti scalari di ciascuna riga di  $A$  per  
ciascuna colonna di  $B$



# Quali sono i sotto-problemi indipendenti?

---

Gli elementi di  $C$  sono  
calcolati effettuando i  
**prodotti scalari** di ciascuna riga di  $A$  per  
ciascuna colonna di  $B$



I prodotti scalari sono calcolati  
in maniera indipendente  
l'uno dall'altro

# Altra possibile decomposizione

---

Decomposizione in BLOCCHI di RIGHE

+

Decomposizione in BLOCCHI di COLONNE

=

Decomposizione in BLOCCHI QUADRATI

## Esempio: matrici a blocchi 3x3 = 9

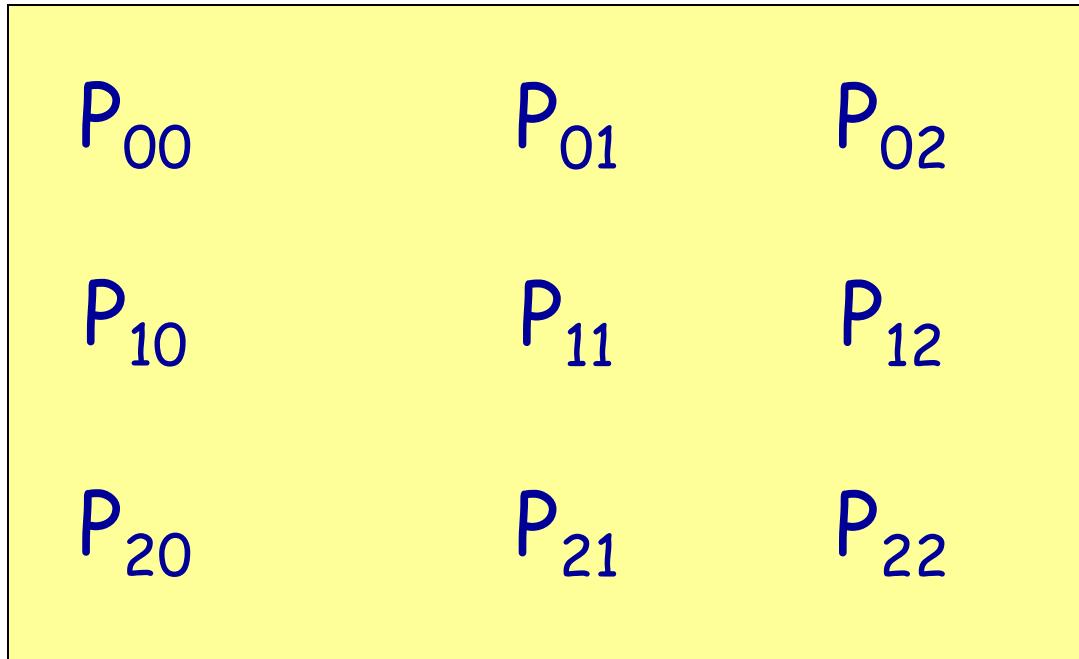
---

$$\begin{bmatrix} C_{00} & C_{01} & C_{02} \\ C_{10} & C_{11} & C_{12} \\ C_{20} & C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{00} & B_{01} & B_{02} \\ B_{10} & B_{11} & B_{12} \\ B_{20} & B_{21} & B_{22} \end{bmatrix}$$

Supponiamo di disporre di  $p \times p$  processori  
secondo una griglia di dimensione  $p \times p$ ...

# Esempio: $3 \times 3 = 9$ processori

---



Al processore  $P_{ij}$  assegniamo i blocchi  $A_{ij}$  e  $B_{ij}$

# Distribuzione dei dati

$P_{00}$	$P_{01}$	$P_{02}$
$A_{00} \ B_{00}$	$A_{01} \ B_{01}$	$A_{02} \ B_{02}$
$P_{10}$	$P_{11}$	$P_{12}$
$A_{10} \ B_{10}$	$A_{11} \ B_{11}$	$A_{12} \ B_{12}$
$P_{20}$	$P_{21}$	$P_{22}$
$A_{20} \ B_{20}$	$A_{21} \ B_{21}$	$A_{22} \ B_{22}$

# IDEA!

---

Con i dati così distribuiti

Vogliamo che il processore  $P_{ij}$

Calcoli il blocco  $C_{ij}$

# Ovvero...

---

$$C_{00} =$$

$$A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$$

$$C_{01} =$$

$$A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21}$$

$$C_{02} =$$

$$A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22}$$

$$C_{10} =$$

$$A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20}$$

$$C_{11} =$$

$$A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} =$$

$$A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22}$$

$$C_{20} =$$

$$A_{20}B_{00} + A_{21}B_{10} + A_{22}B_{20}$$

$$C_{21} =$$

$$A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} =$$

$$A_{20}B_{02} + A_{21}B_{12} + A_{22}B_{22}$$

# Ovvero...

---

$$A_{00} \ B_{00}$$

$$C_{00} =$$

$$A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$$

$$A_{01} \ B_{01}$$

$$C_{01} =$$

$$A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21}$$

$$A_{02} \ B_{02}$$

$$C_{02} =$$

$$A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22}$$

$$A_{10} \ B_{10}$$

$$C_{10} =$$

$$A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20}$$

$$A_{11} \ B_{11}$$

$$C_{11} =$$

$$A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21}$$

$$A_{12} \ B_{12}$$

$$C_{12} =$$

$$A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22}$$

$$A_{20} \ B_{20}$$

$$C_{20} =$$

$$A_{20}B_{00} + A_{21}B_{10} + A_{22}B_{20}$$

$$A_{21} \ B_{21}$$

$$C_{21} =$$

$$A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21}$$

$$A_{22} \ B_{22}$$

$$C_{22} =$$

$$A_{20}B_{02} + A_{21}B_{12} + A_{22}B_{22}$$

# Osservazione I

---

Con la distribuzione  
dei dati effettuata  
solo i processori sulla diagonale  
della griglia, ovvero  $P_{ii}$   
possono calcolare  
"un contributo" del blocco  $C_{ii}$

Ovvero...

ovvero...

---

$$A_{00} \ B_{00}$$

$$C_{00} =$$

$$A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$$

$$A_{01} \ B_{01}$$

$$C_{01} =$$

$$A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21}$$

$$A_{02} \ B_{02}$$

$$C_{02} =$$

$$A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22}$$

$$A_{10} \ B_{10}$$

$$C_{10} =$$

$$A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20}$$

$$A_{11} \ B_{11}$$

$$C_{11} =$$

$$A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21}$$

$$A_{12} \ B_{12}$$

$$C_{12} =$$

$$A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22}$$

$$A_{20} \ B_{20}$$

$$C_{20} =$$

$$A_{20}B_{00} + A_{21}B_{10} + A_{22}B_{20}$$

$$A_{21} \ B_{21}$$

$$C_{21} =$$

$$A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21}$$

$$A_{22} \ B_{22}$$

$$C_{22} =$$

$$A_{20}B_{02} + A_{21}B_{12} + A_{22}B_{22}$$

## Osservazione II

$$A_{00} \ B_{00}$$

$$C_{00} =$$

$$A_{00} B_{00} + A_{01} B_{10} + A_{02} B_{20}$$

$$A_{01} \ B_{01}$$

$$C_{01} =$$

$$A_{00} B_{01} + A_{01} B_{11} + A_{02} B_{21}$$

$$A_{02} \ B_{02}$$

$$C_{02} =$$

$$A_{00} B_{02} + A_{01} B_{12} + A_{02} B_{22}$$

$$A_{10} \ B_{10}$$

$$C_{10} =$$

$$A_{10} B_{00} + A_{11} B_{10} + A_{12} B_{20}$$

$$A_{11} \ B_{11}$$

$$C_{11} =$$

$$A_{10} B_{01} + A_{11} B_{11} + A_{12} B_{21}$$

$$A_{12} \ B_{12}$$

$$C_{12} =$$

$$A_{10} B_{02} + A_{11} B_{12} + A_{12} B_{22}$$

$$A_{20} \ B_{20}$$

$$C_{20} =$$

$$A_{20} B_{00} + A_{21} B_{10} + A_{22} B_{20}$$

$$A_{21} \ B_{21}$$

$$C_{21} =$$

$$A_{20} B_{01} + A_{21} B_{11} + A_{22} B_{21}$$

$$A_{22} \ B_{22}$$

$$C_{22} =$$

$$A_{20} B_{02} + A_{21} B_{12} + \boxed{A_{22}} B_{22}$$

Nel calcolo dei  $C_{ij}$  ...

- $A_{00}$  è presente nel calcolo della I riga di  $C$  ( $C_{00}, C_{01}, C_{02}$ )
- $A_{11}$  è presente nel calcolo della II riga di  $C$  ( $C_{10}, C_{11}, C_{12}$ )
- $A_{22}$  è presente nel calcolo della III riga di  $C$  ( $C_{20}, C_{21}, C_{22}$ )

# IDEA!

---

I processori sulla diagonale:

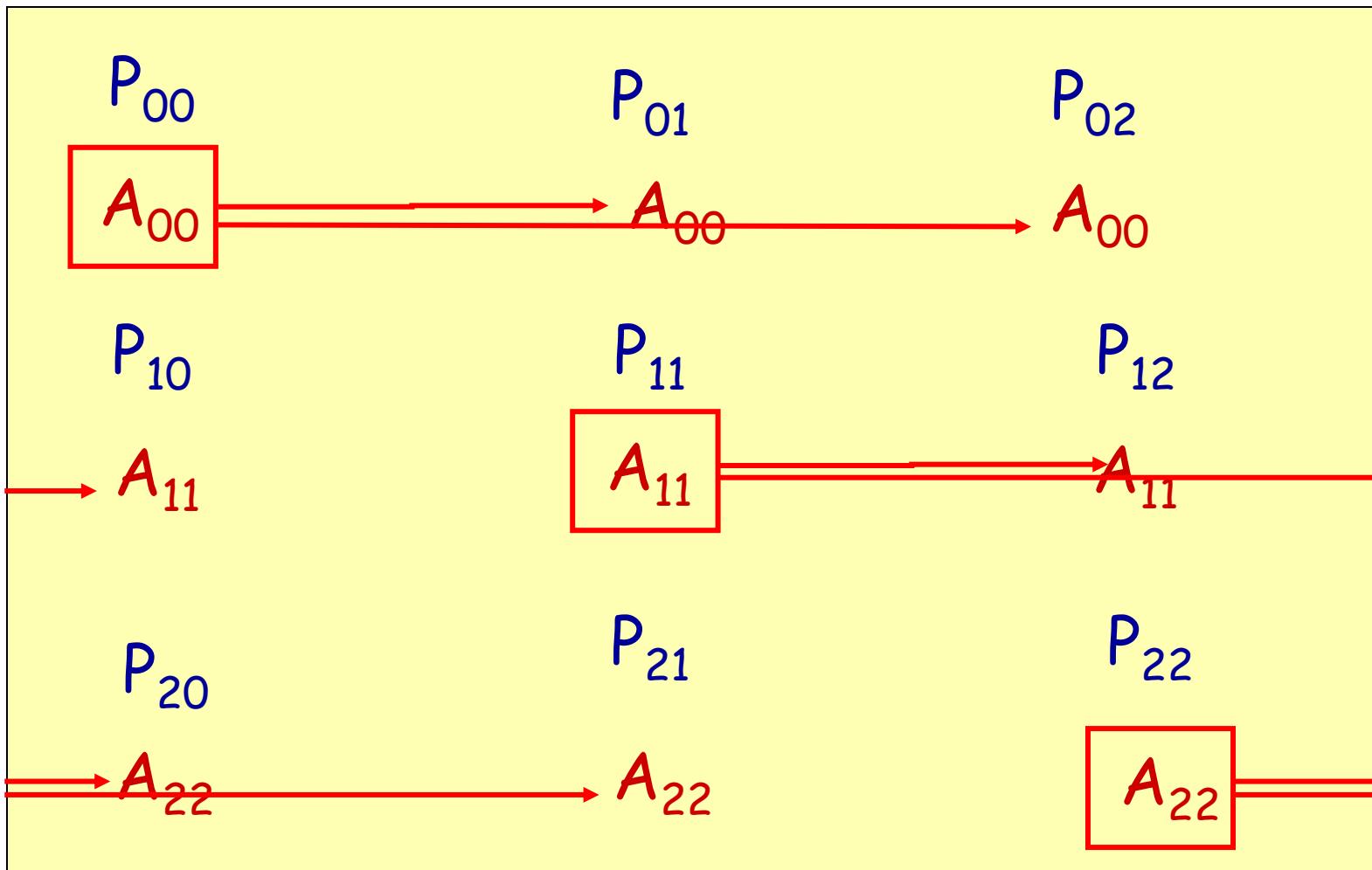
$P_{00}$ ,  $P_{11}$  e  $P_{22}$

inviano il proprio blocco di  $A$ ,

ovvero  $A_{00}$ ,  $A_{11}$  e  $A_{22}$

Rispettivamente ai processori della riga 0,1,2  
(ovvero ai processori che si trovano sulla loro stessa riga!)

# I passo: invio dei blocchi diagonali di A



# Quindi, ogni processore calcola...

$$A_{00} \ B_{00}$$

$$C_{00} =$$

$$A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$$

$$A_{00} \ A_{01} \ B_{01}$$

$$C_{01} =$$

$$A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21}$$

$$A_{00} \ A_{02} \ B_{02}$$

$$C_{02} =$$

$$A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22}$$

$$A_{11} \ A_{10} \ B_{10}$$

$$C_{10} =$$

$$A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20}$$

$$A_{11} \ B_{11}$$

$$C_{11} =$$

$$A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21}$$

$$A_{11} \ A_{12} \ B_{12}$$

$$C_{12} =$$

$$A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22}$$

$$A_{22} \ A_{20} \ B_{20}$$

$$C_{20} =$$

$$A_{20}B_{00} + A_{21}B_{10} + A_{22}B_{20}$$

$$A_{22} \ A_{21} \ B_{21}$$

$$C_{21} =$$

$$A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21}$$

$$A_{22} \ B_{22}$$

$$C_{22} =$$

$$A_{20}B_{02} + A_{21}B_{12} + A_{22}B_{22}$$

# Quindi, ogni processore calcola...

$$A_{00} \ B_{00}$$

$$C_{00} =$$

$$A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$$

$$A_{00} \ A_{01} \ B_{01}$$

$$C_{01} =$$

$$A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21}$$

$$A_{00} \ A_{02} \ B_{02}$$

$$C_{02} =$$

$$A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22}$$

$$A_{11} \ A_{10} \ B_{10}$$

$$C_{10} =$$

$$A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20}$$

$$A_{11} \ B_{11}$$

$$C_{11} =$$

$$A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21}$$

$$A_{11} \ A_{12} \ B_{12}$$

$$C_{12} =$$

$$A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22}$$

$$A_{22} \ A_{20} \ B_{20}$$

$$C_{20} =$$

$$A_{20}B_{00} + A_{21}B_{10} + A_{22}B_{20}$$

$$A_{22} \ A_{21} \ B_{21}$$

$$C_{21} =$$

$$A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21}$$

$$A_{22} \ B_{22}$$

$$C_{22} =$$

$$A_{20}B_{02} + A_{21}B_{12} + A_{22}B_{22}$$

## II passo

$$A_{00} \ B_{00}$$

$$C_{00} =$$

$$A_{00} \ B_{00} + A_{01} \boxed{B_{10}} + A_{02} \ B_{20}$$

$$A_{00} \ A_{01} \ B_{01}$$

$$C_{01} =$$

$$A_{00} \ B_{01} + A_{01} \boxed{B_{11}} + A_{02} \ B_{21}$$

$$A_{00} \ A_{02} \ B_{02}$$

$$C_{02} =$$

$$A_{00} \ B_{02} + A_{01} \boxed{B_{12}} + A_{02} \ B_{22}$$

$$A_{11} \ A_{10} \ B_{10}$$

$$C_{10} =$$

$$A_{10} \ B_{00} + A_{11} \ B_{10} + \boxed{A_{12}} \ B_{20}$$

$$A_{11} \ B_{11}$$

$$C_{11} =$$

$$A_{10} \ B_{01} + A_{11} \ B_{11} + \boxed{A_{12}} \ B_{21}$$

$$A_{11} \ A_{12} \ B_{12}$$

$$C_{12} =$$

$$A_{10} \ B_{02} + A_{11} \ B_{12} + \boxed{A_{12}} \ B_{22}$$

$$A_{22} \ A_{20} \ B_{20}$$

$$C_{20} =$$

$$\boxed{A_{20}} \ B_{00} + A_{21} \ B_{10} + A_{22} \ B_{20}$$

$$A_{22} \ A_{21} \ B_{21}$$

$$C_{21} =$$

$$\boxed{A_{20}} \ B_{01} + A_{21} \ B_{11} + A_{22} \ B_{21}$$

$$A_{22} \ B_{22}$$

$$C_{22} =$$

$$\boxed{A_{20}} \ B_{02} + A_{21} \ B_{12} + A_{22} \ B_{22}$$

Nel calcolo dei  $C_{ij}$  ...

•  $A_{01}$  è presente nel calcolo della I riga di  $C$  ( $C_{00}, C_{01}, C_{02}$ )

•  $A_{12}$  è presente nel calcolo della II riga di  $C$  ( $C_{10}, C_{11}, C_{12}$ )

G. Accetta •  $A_{20}$  è presente nel calcolo della III riga di  $C$  ( $C_{20}, C_{21}, C_{22}$ )

# IDEA!

---

I processori:

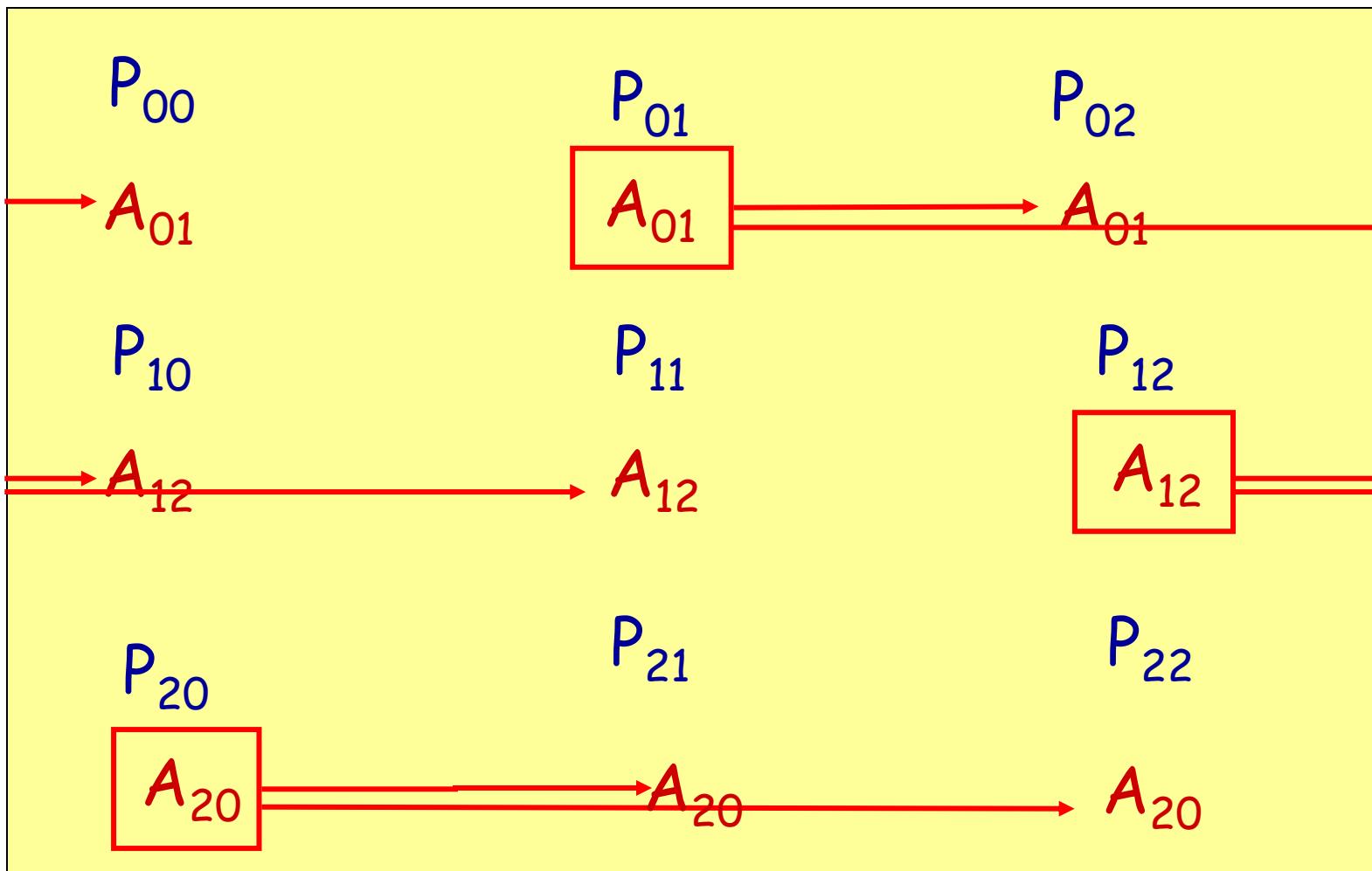
$P_{01}$ ,  $P_{12}$  e  $P_{20}$

inviano il proprio blocco di  $A$ ,

ovvero  $A_{01}$ ,  $A_{12}$  e  $A_{20}$

Rispettivamente ai processori della riga 0,1,2  
(ovvero ai processori che si trovano sulla loro stessa riga!)

## II passo: invio dei blocchi di A



# Osservazione: per effettuare i prodotti ...

$$A_{01} \quad B_{00}$$

$$C_{00} =$$

$$A_{00} \quad B_{00} + A_{01} \quad B_{10} + A_{02} \quad B_{20}$$

$$A_{00} \quad A_{01} \quad B_{01}$$

$$C_{01} =$$

$$A_{00} \quad B_{01} + A_{01} \quad B_{11} + A_{02} \quad B_{21}$$

$$A_{01} \quad A_{02} \quad B_{02}$$

$$C_{02} =$$

$$A_{00} \quad B_{02} + A_{01} \quad B_{12} + A_{02} \quad B_{22}$$

$$A_{12} \quad A_{10} \quad B_{10}$$

$$C_{10} =$$

$$A_{10} \quad B_{00} + A_{11} \quad B_{10} + A_{12} \quad B_{20}$$

$$A_{12} \quad B_{11}$$

$$C_{11} =$$

$$A_{10} \quad B_{01} + A_{11} \quad B_{11} + A_{12} \quad B_{21}$$

$$A_{11} \quad A_{12} \quad B_{12}$$

$$C_{12} =$$

$$A_{10} \quad B_{02} + A_{11} \quad B_{12} + A_{12} \quad B_{22}$$

$$A_{22} \quad A_{20} \quad B_{20}$$

$$C_{20} =$$

$$A_{20} \quad B_{00} + A_{21} \quad B_{10} + A_{22} \quad B_{20}$$

$$A_{20} \quad A_{21} \quad B_{21}$$

$$C_{21} =$$

$$A_{20} \quad B_{01} + A_{21} \quad B_{11} + A_{22} \quad B_{21}$$

$$A_{20} \quad B_{22}$$

$$C_{22} =$$

$$A_{20} \quad B_{02} + A_{21} \quad B_{12} + A_{22} \quad B_{22}$$

Ciascun processore ha bisogno anche del corrispondente blocco di  $B$ !

(E precisamente del blocco di  $B$

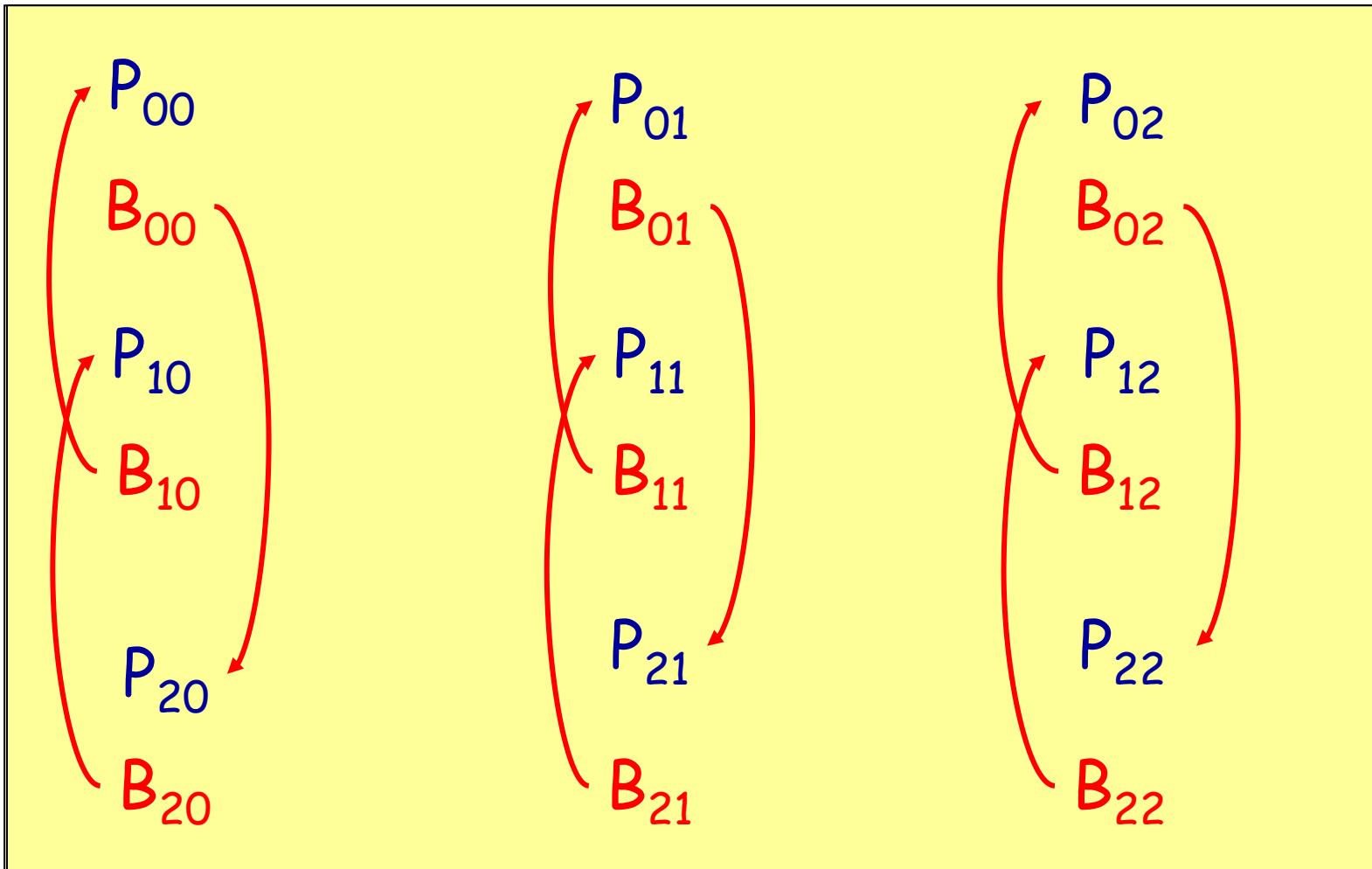
del processore nella sua stessa colonna ma nella riga successiva! )

# IDEA!

---

Per ogni colonna della griglia di processori,  
ciascun processore  $P_{ij}$   
**Invia il proprio blocco  $B_{ij}$ ,**  
al processore situato nella stessa colonna  
e sulla riga precedente!

# Comunicazione dei blocchi di B



# Quindi, ogni processore calcola...

$$A_{01} \ A_{10}$$

$$C_{00} =$$

$$A_{00} \ B_{00} + A_{01} \ B_{10} + A_{02} \ B_{20}$$

$$A_{00} \ A_{01} \ B_{11}$$

$$C_{01} =$$

$$A_{00} \ B_{01} + A_{01} \ B_{11} + A_{02} \ B_{21}$$

$$A_{01} \ A_{02} \ B_{12}$$

$$C_{02} =$$

$$A_{00} \ B_{02} + A_{01} \ B_{12} + A_{02} \ B_{22}$$

$$A_{12} \ A_{10} \ B_{20}$$

$$C_{10} =$$

$$A_{10} \ B_{00} + A_{11} \ B_{10} + A_{12} \ B_{20}$$

$$A_{12} \ B_{21}$$

$$C_{11} =$$

$$A_{10} \ B_{01} + A_{11} \ B_{11} + A_{12} \ B_{21}$$

$$A_{11} \ A_{12} \ B_{22}$$

$$C_{12} =$$

$$A_{10} \ B_{02} + A_{11} \ B_{12} + A_{12} \ B_{22}$$

$$A_{22} \ A_{20} \ B_{00}$$

$$C_{20} =$$

$$A_{20} \ B_{00} + A_{21} \ B_{10} + A_{22} \ B_{20}$$

$$A_{20} \ A_{21} \ B_{01}$$

$$C_{21} =$$

$$A_{20} \ B_{01} + A_{21} \ B_{11} + A_{22} \ B_{21}$$

$$A_{20} \ B_{02}$$

$$C_{22} =$$

$$A_{20} \ B_{02} + A_{21} \ B_{12} + A_{22} \ B_{22}$$

# Quindi, ogni processore calcola...

$$A_{01} \ A_{10}$$

$$C_{00} =$$

$$A_{00} \ B_{00} + A_{01} \ B_{10} + A_{02} \ B_{20}$$

$$A_{00} \ A_{01} \ B_{11}$$

$$C_{01} =$$

$$A_{00} \ B_{01} + A_{01} \ B_{11} + A_{02} \ B_{21}$$

$$A_{01} \ A_{02} \ B_{12}$$

$$C_{02} =$$

$$A_{00} \ B_{02} + A_{01} \ B_{12} + A_{02} \ B_{22}$$

$$A_{12} \ A_{10} \ B_{20}$$

$$C_{10} =$$

$$A_{10} \ B_{00} + A_{11} \ B_{10} + A_{12} \ B_{20}$$

$$A_{12} \ B_{21}$$

$$C_{11} =$$

$$A_{10} \ B_{01} + A_{11} \ B_{11} + A_{12} \ B_{21}$$

$$A_{11} \ A_{12} \ B_{22}$$

$$C_{12} =$$

$$A_{10} \ B_{02} + A_{11} \ B_{12} + A_{12} \ B_{22}$$

$$A_{22} \ A_{20} \ B_{00}$$

$$C_{20} =$$

$$A_{20} \ B_{00} + A_{21} \ B_{10} + A_{22} \ B_{20}$$

$$A_{20} \ A_{21} \ B_{01}$$

$$C_{21} =$$

$$A_{20} \ B_{01} + A_{21} \ B_{11} + A_{22} \ B_{21}$$

$$A_{20} \ B_{02}$$

$$C_{22} =$$

$$A_{20} \ B_{02} + A_{21} \ B_{12} + A_{22} \ B_{22}$$

# III passo

$$A_{01} \ B_{10}$$

$$C_{00} =$$

$$A_{00} \ B_{00} + A_{01} \ B_{10} + A_{02} \ B_{20}$$

$$A_{00} \ A_{01} \ B_{11}$$

$$C_{01} =$$

$$A_{00} \ B_{01} + A_{01} \ B_{11} + A_{02} \ B_{21}$$

$$A_{01} \ A_{02} \ B_{12}$$

$$C_{02} =$$

$$A_{00} \ B_{02} + A_{01} \ B_{12} + A_{02} \ B_{22}$$

$$A_{12} \ A_{10} \ B_{20}$$

$$C_{10} =$$

$$A_{10} \ B_{00} + A_{11} \ B_{10} + A_{12} \ B_{20}$$

$$A_{12} \ B_{21}$$

$$C_{11} =$$

$$A_{10} \ B_{01} + A_{11} \ B_{11} + A_{12} \ B_{21}$$

$$A_{11} \ A_{12} \ B_{22}$$

$$C_{12} =$$

$$A_{10} \ B_{02} + A_{11} \ B_{12} + A_{12} \ B_{22}$$

$$A_{22} \ A_{20} \ B_{00}$$

$$C_{20} =$$

$$A_{20} \ B_{00} + A_{21} \ B_{10} + A_{22} \ B_{20}$$

$$A_{20} \ A_{21} \ B_{01}$$

$$C_{21} =$$

$$A_{20} \ B_{01} + A_{21} \ B_{11} + A_{22} \ B_{21}$$

$$A_{20} \ B_{02}$$

$$C_{22} =$$

$$A_{20} \ B_{02} + A_{21} \ B_{12} + A_{22} \ B_{22}$$

Nel calcolo dei  $C_{ij}$  ...

- $A_{02}$  è presente nel calcolo della I riga di  $C$  ( $C_{00}, C_{01}, C_{02}$ )

- $A_{10}$  è presente nel calcolo della II riga di  $C$  ( $C_{10}, C_{11}, C_{12}$ )

- $A_{21}$  è presente nel calcolo della III riga di  $C$  ( $C_{20}, C_{21}, C_{22}$ )

# IDEA!

---

I processori:

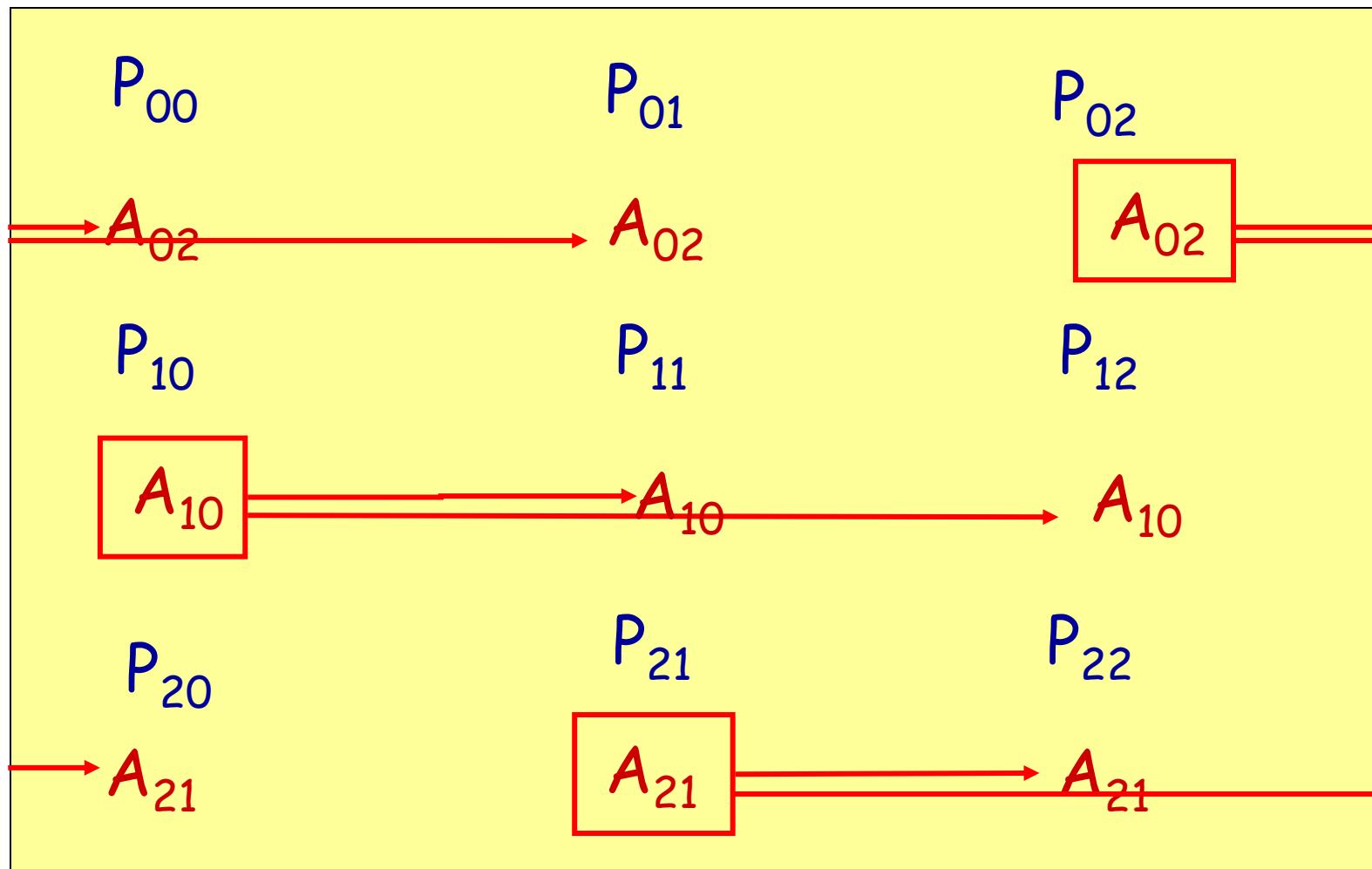
$P_{02}$ ,  $P_{10}$  e  $P_{21}$

inviano il proprio blocco di  $A$ ,

ovvero  $A_{02}$ ,  $A_{10}$  e  $A_{21}$

Rispettivamente ai processori della riga 0,1,2  
(ovvero ai processori che si trovano sulla loro stessa riga!)

## II passo: invio dei blocchi di A



# Osservazione: per effettuare i prodotti ...

$$A_{02} \ A_{10}$$

$$C_{00} =$$

$$A_{00} \ B_{00} + A_{01} \ B_{10} + A_{02} \ B_{20}$$

$$A_{02} \ A_{01} \ B_{11}$$

$$C_{01} =$$

$$A_{00} \ B_{01} + A_{01} \ B_{11} + A_{02} \ B_{21}$$

$$A_{01} \ A_{02} \ B_{12}$$

$$C_{02} =$$

$$A_{00} \ B_{02} + A_{01} \ B_{12} + A_{02} \ B_{22}$$

$$A_{12} \ A_{10} \ B_{20}$$

$$C_{10} =$$

$$A_{10} \ B_{00} + A_{11} \ B_{10} + A_{12} \ B_{20}$$

$$A_{10} \ B_{21}$$

$$C_{11} =$$

$$A_{10} \ B_{01} + A_{11} \ B_{11} + A_{12} \ B_{21}$$

$$A_{10} \ A_{12} \ B_{22}$$

$$C_{12} =$$

$$A_{10} \ B_{02} + A_{11} \ B_{12} + A_{12} \ B_{22}$$

$$A_{21} \ A_{20} \ B_{00}$$

$$C_{20} =$$

$$A_{20} \ B_{00} + A_{21} \ B_{10} + A_{22} \ B_{20}$$

$$A_{20} \ A_{21} \ B_{01}$$

$$C_{21} =$$

$$A_{20} \ B_{01} + A_{21} \ B_{11} + A_{22} \ B_{21}$$

$$A_{21} \ B_{02}$$

$$C_{22} =$$

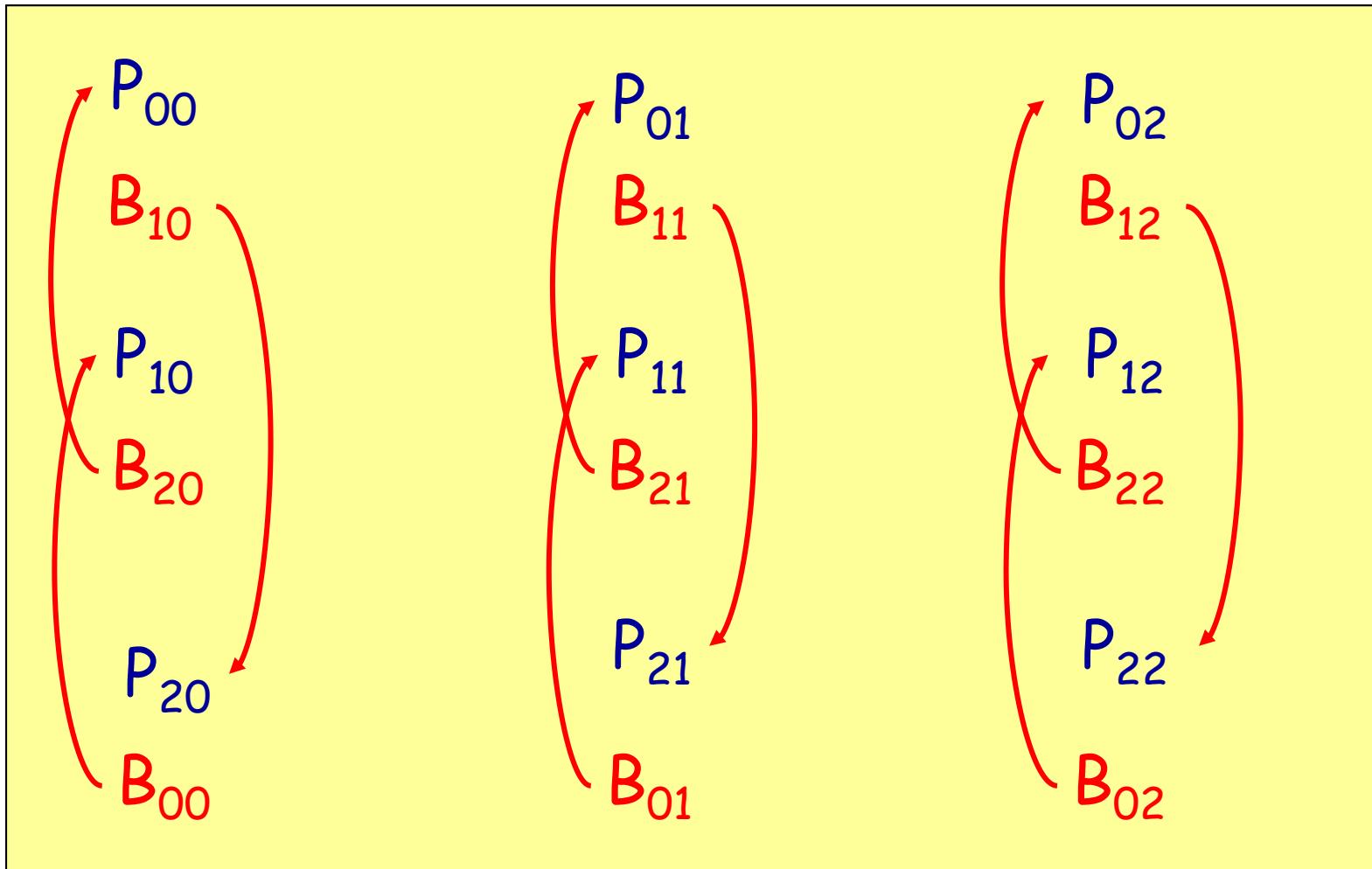
$$A_{20} \ B_{02} + A_{21} \ B_{12} + A_{22} \ B_{22}$$

Ciascun processore ha bisogno anche del corrispondente blocco di  $B$ !

(E precisamente del blocco di  $B$

del processore nella sua stessa colonna ma nella riga successiva! )

# Comunicazione dei blocchi di B



# Quindi, ogni processore calcola...

$$A_{02} \ A_{20}$$

$$C_{00} =$$

$$A_{00} \ B_{00} + A_{01} \ B_{10} + A_{02} \ B_{20}$$

$$A_{02} \ A_{01} \ B_{21}$$

$$C_{01} =$$

$$A_{00} \ B_{01} + A_{01} \ B_{11} + A_{02} \ B_{21}$$

$$A_{01} \ A_{02} \ B_{22}$$

$$C_{02} =$$

$$A_{00} \ B_{02} + A_{01} \ B_{12} + A_{02} \ B_{22}$$

$$A_{12} \ A_{10} \ B_{00}$$

$$C_{10} =$$

$$A_{10} \ B_{00} + A_{11} \ B_{10} + A_{12} \ B_{20}$$

$$A_{10} \ B_{01}$$

$$C_{11} =$$

$$A_{10} \ B_{01} + A_{11} \ B_{11} + A_{12} \ B_{21}$$

$$A_{10} \ A_{12} \ B_{02}$$

$$C_{12} =$$

$$A_{10} \ B_{02} + A_{11} \ B_{12} + A_{12} \ B_{22}$$

$$A_{21} \ A_{20} \ B_{10}$$

$$C_{20} =$$

$$A_{20} \ B_{00} + A_{21} \ B_{10} + A_{22} \ B_{20}$$

$$A_{20} \ A_{21} \ B_{11}$$

$$C_{21} =$$

$$A_{20} \ B_{01} + A_{21} \ B_{11} + A_{22} \ B_{21}$$

$$A_{21} \ B_{12}$$

$$C_{22} =$$

$$A_{20} \ B_{02} + A_{21} \ B_{12} + A_{22} \ B_{22}$$

# Quindi, ogni processore calcola...

$$A_{02} \ A_{20}$$

$$C_{00} =$$

$$A_{00} \ B_{00} + A_{01} \ B_{10} + A_{02} \ B_{20}$$

$$A_{02} \ A_{01} \ B_{21}$$

$$C_{01} =$$

$$A_{00} \ B_{01} + A_{01} \ B_{11} + A_{02} \ B_{21}$$

$$A_{01} \ A_{02} \ B_{22}$$

$$C_{02} =$$

$$A_{00} \ B_{02} + A_{01} \ B_{12} + A_{02} \ B_{22}$$

$$A_{12} \ A_{10} \ B_{00}$$

$$C_{10} =$$

$$A_{10} \ B_{00} + A_{11} \ B_{10} + A_{12} \ B_{20}$$

$$A_{10} \ B_{01}$$

$$C_{11} =$$

$$A_{10} \ B_{01} + A_{11} \ B_{11} + A_{12} \ B_{21}$$

$$A_{10} \ A_{12} \ B_{02}$$

$$C_{12} =$$

$$A_{10} \ B_{02} + A_{11} \ B_{12} + A_{12} \ B_{22}$$

$$A_{21} \ A_{20} \ B_{10}$$

$$C_{20} =$$

$$A_{20} \ B_{00} + A_{21} \ B_{10} + A_{22} \ B_{20}$$

$$A_{20} \ A_{21} \ B_{11}$$

$$C_{21} =$$

$$A_{20} \ B_{01} + A_{21} \ B_{11} + A_{22} \ B_{21}$$

$$A_{21} \ B_{12}$$

$$C_{22} =$$

$$A_{20} \ B_{02} + A_{21} \ B_{12} + A_{22} \ B_{22}$$

# Quindi, ogni processore calcola...

$$C_{00} =$$

$$A_{00} B_{00} + A_{01} B_{10} + A_{02} B_{20}$$

$$C_{01} =$$

$$A_{00} B_{01} + A_{01} B_{11} + A_{02} B_{21}$$

$$C_{02} =$$

$$A_{00} B_{02} + A_{01} B_{12} + A_{02} B_{22}$$

$$C_{10} =$$

$$A_{10} B_{00} + A_{11} B_{10} + A_{12} B_{20}$$

$$C_{11} =$$

$$A_{10} B_{01} + A_{11} B_{11} + A_{12} B_{21}$$

$$C_{12} =$$

$$A_{10} B_{02} + A_{11} B_{12} + A_{12} B_{22}$$

$$C_{20} =$$

$$A_{20} B_{00} + A_{21} B_{10} + A_{22} B_{20}$$

$$C_{21} =$$

$$A_{20} B_{01} + A_{21} B_{11} + A_{22} B_{21}$$

$$C_{22} =$$

$$A_{20} B_{02} + A_{21} B_{12} + A_{22} B_{22}$$

Dopo  $p$  passi ogni processore  $P_{ij}$  ha calcolato  
il corrispondente blocco  $C_{ij}$

# IV Strategia: in generale

---

Broadcast Multiply Rolling

La strategia è costituita da **p** passi.

Si parte dalla diagonale principale

della griglia di processori,

ad ogni passo k, si considera

la k-ma diagonale situata al di sopra di quella principale.

I processori situati lungo la diagonale

effettuano una comunicazione collettiva

del blocco di A in loro possesso a tutti i processori

della medesima riga.

# IV Strategia: in generale

---

## Broadcast Multiply Rolling

Inoltre, ad ogni passo  
ciascun processore effettua  
una comunicazione del proprio blocco di B  
al processore situato  
nella stessa colonna  
e nella riga precedente!

---

Fine

# Calcolo Parallello e Distribuito

## Prodotto Matrice-Matrice

---

prof. Giuliano Laccetti a.a. 2021-2022 - prodotto mat-mat  
14/12/2021

Materiale tratto da slide, appunti, lezioni di Calcolo Parallello e Distribuito del prof. A. Murli  
e dal testo  
A. Murli - Lezioni di Calcolo Parallello, ed. Liguori

---

prof. Giuliano Laccetti a.a. 2021-2022 - prodotto mat-mat  
14/12/2021

# Calcolo Parallello e Distribuito

## Prodotto Matrice-Matrice

---

prof. Giuliano Laccetti a.a. 2021-2022 - prodotto mat-mat  
14/12/2021

Materiale tratto da slide, appunti, lezioni di Calcolo Parallello e Distribuito del prof. A. Murli  
e dal testo  
A. Murli - Lezioni di Calcolo Parallello, ed. Liguori

# Problema

---

Progettazione  
di un algoritmo parallelo  
per architettura **MIMD**  
a memoria distribuita  
per il calcolo del prodotto righe per colonne  
di 2 matrici  $A$  e  $B$ :

$$C = A \bullet B, \quad A, B \in \mathbb{R}^{n \times n}$$

# Qual è l'algoritmo sequenziale ?

```
for i=0,n-1 do  
    for j=0,n-1 do  
        cij = 0  
        for k=0,n-1 do  
            cij = cij + aik bkj  
        endfor  
    endfor  
endfor
```

Prodotto Matrice-Matrice

$$A \bullet B = C, \quad A, B \in \mathbb{R}^{n \times n}$$

# Qual è l'algoritmo sequenziale ?

```
for i=0,n-1 do  
  for j=0,n-1 do  
    cij = 0  
    for k=0,n-1 do  
      cij = cij + aik bkj  
    endfor  
  endfor  
endfor
```

Su un calcolatore tradizionale  
la matrice  $C$   
viene "generalmente" calcolata  
componente per componente  
secondo un ordine prestabilito

*Il generico elemento di  $C$   
è il prodotto scalare della  
 $i$ -esima riga di  $A$  per la  
 $j$ -esima colonna di  $B$*

# Domanda

---

Qual è  
l'algoritmo parallelo  
?

ovvero

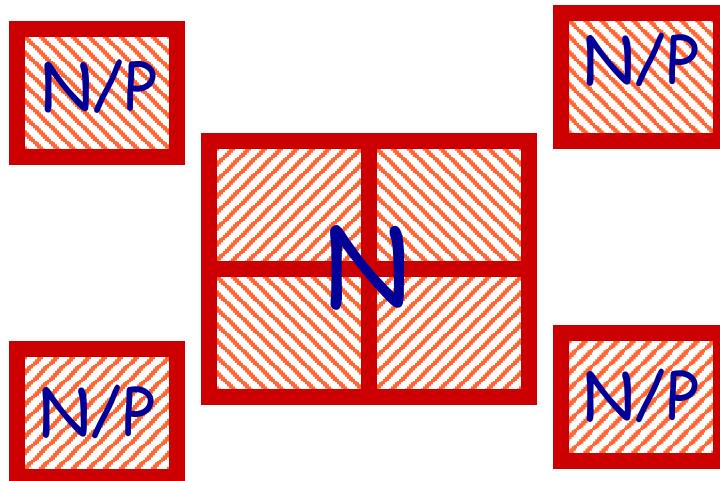


Come decomporre  
il problema  
Matrice-Matrice ?

# DECOMPOSIZIONE: IDEA GENERALE

---

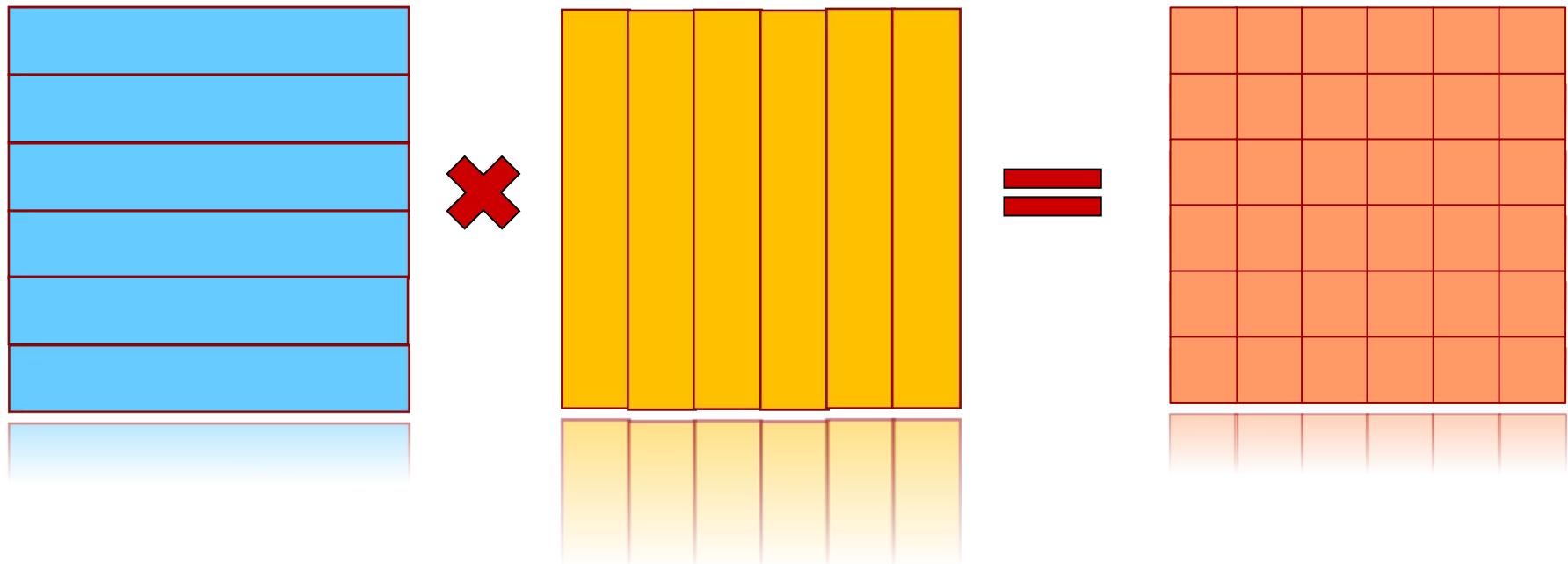
Decomporre un problema di dimensione  $N$   
in  $P$  sottoproblemi di dimensione  $N/P$   
e risolverli contemporaneamente  
su più calcolatori



# Quali sono i sotto-problemi indipendenti?

---

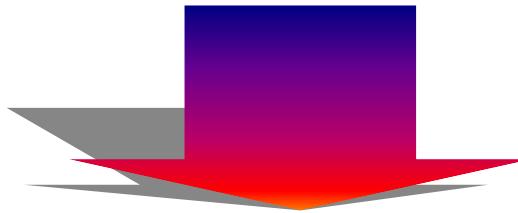
Gli elementi di  $C$  sono  
calcolati effettuando i  
prodotti scalari di ciascuna riga di  $A$  per  
ciascuna colonna di  $B$



# Quali sono i sotto-problemi indipendenti?

---

Gli elementi di  $C$  sono  
calcolati effettuando i  
**prodotti scalari** di ciascuna riga di  $A$  per  
ciascuna colonna di  $B$



I prodotti scalari sono calcolati  
in maniera indipendente  
l'uno dall'altro

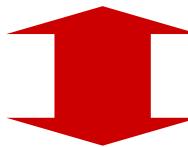
# IDEA!

---

Decomposizione del problema  
Matrice per Matrice



Partizionamento delle matrici A e B  
**IN BLOCCHI**



Riformulazione dell'algoritmo sequenziale  
**"A BLOCCHI"**

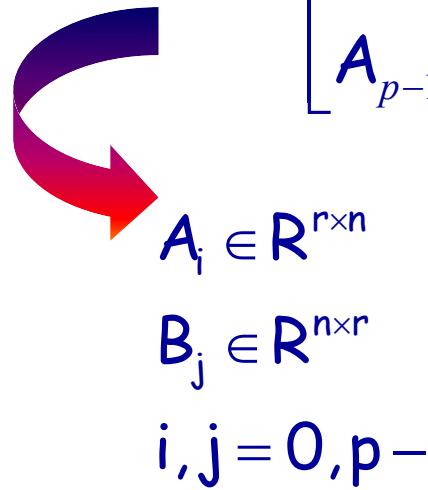


Parallelismo dell'algoritmo  
**"A BLOCCHI"**

# Algoritmo a blocchi - I strategia

Decomposizione di  $A$  per  $p$  blocchi di righe e  
di  $B$  per  $p$  blocchi di colonne

$$\begin{bmatrix} A_0 \\ A_1 \\ \dots \\ A_{p-1} \end{bmatrix} \bullet \begin{bmatrix} B_0 B_1 \dots B_{p-1} \end{bmatrix} = \begin{bmatrix} C_{00} & C_{01} & \dots & C_{0,p-1} \\ C_{10} & \dots & \dots & C_{1,p-1} \\ \dots & \dots & \dots & \dots \\ C_{p-1,0} & .. & .. & C_{p-1,p-1} \end{bmatrix}$$


$$A_i \in \mathbb{R}^{r \times n}$$
$$B_j \in \mathbb{R}^{n \times r}$$
$$i, j = 0, p-1$$

# Algoritmo a blocchi - I strategia

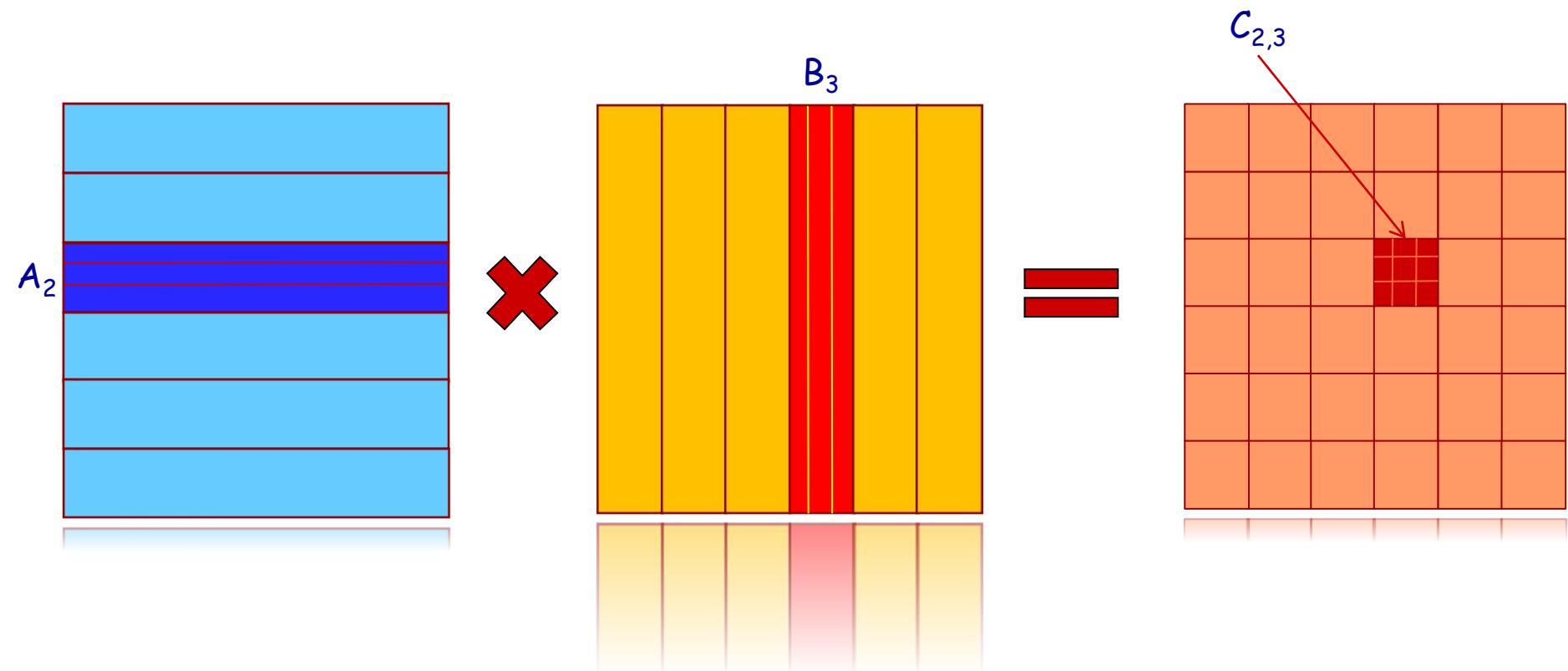
Decomposizione di  $A$  per  $p$  blocchi di righe e  
di  $B$  per  $p$  blocchi di colonne

$$\begin{array}{l} A_i \in \mathbb{R}^{r \times n} \\ B_j \in \mathbb{R}^{n \times r} \\ i, j = 0, p-1 \end{array} \quad \Rightarrow \quad A_i B_j = C_{ij}$$

Prodotti matrice-matrice  
di dimensione minore!

```
begin
    C=0
    for i=0 to p-1 do
        for j=0 to p-1 do
            Cij=AiBj
        endfor
    endfor
end
```

# Algoritmo a blocchi - I strategia



# Algoritmo a blocchi - II strategia

Decomposizione di  $A$  per  $p$  blocchi di colonne e  
di  $B$  per  $p$  blocchi di righe

$$\begin{bmatrix} A_0 & A_1 & \dots & A_{p-1} \end{bmatrix} \bullet \begin{bmatrix} B_0 \\ B_1 \\ \dots \\ B_{p-1} \end{bmatrix} = \sum_{i=0}^{p-1} C^i$$



$$A_i \in R^{n \times r}$$

$$B_i \in R^{r \times n}$$

$$i = 0, p-1$$



$$A_i B_i = C^i$$

# Algoritmo a blocchi - II strategia

Decomposizione di  $A$  per  $p$  blocchi di colonne e  
di  $B$  per  $p$  blocchi di righe

$A_i \in R^{n \times r}$

$B_i \in R^{r \times n}$

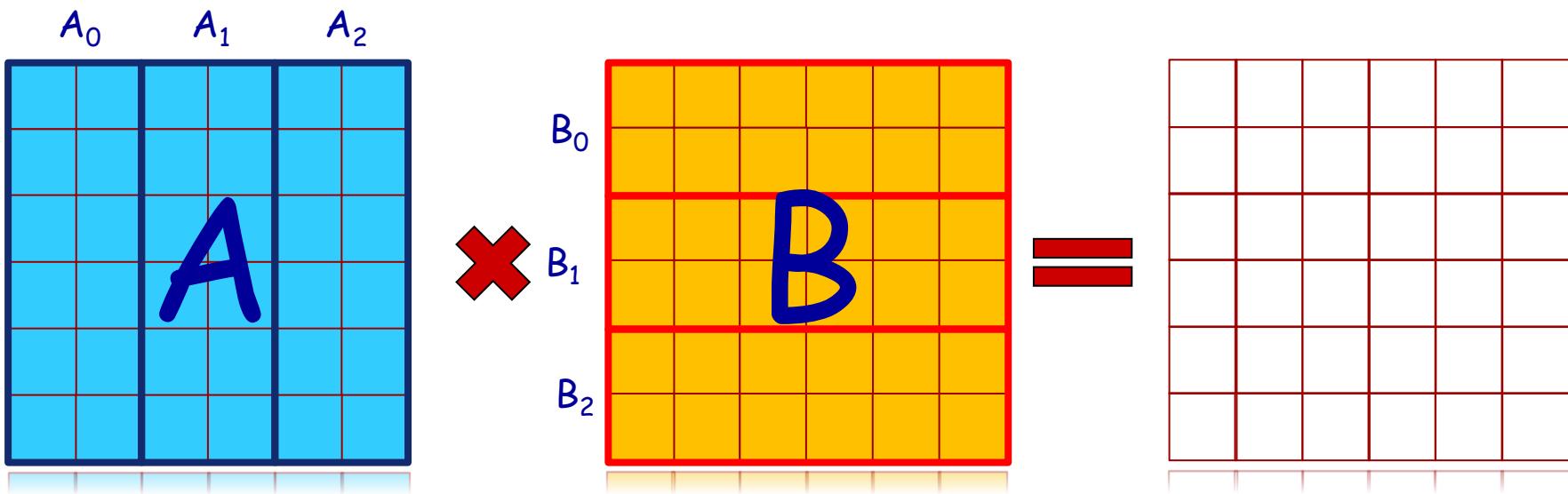
$i = 0, p-1$

$\Rightarrow A_i B_i = C^i$

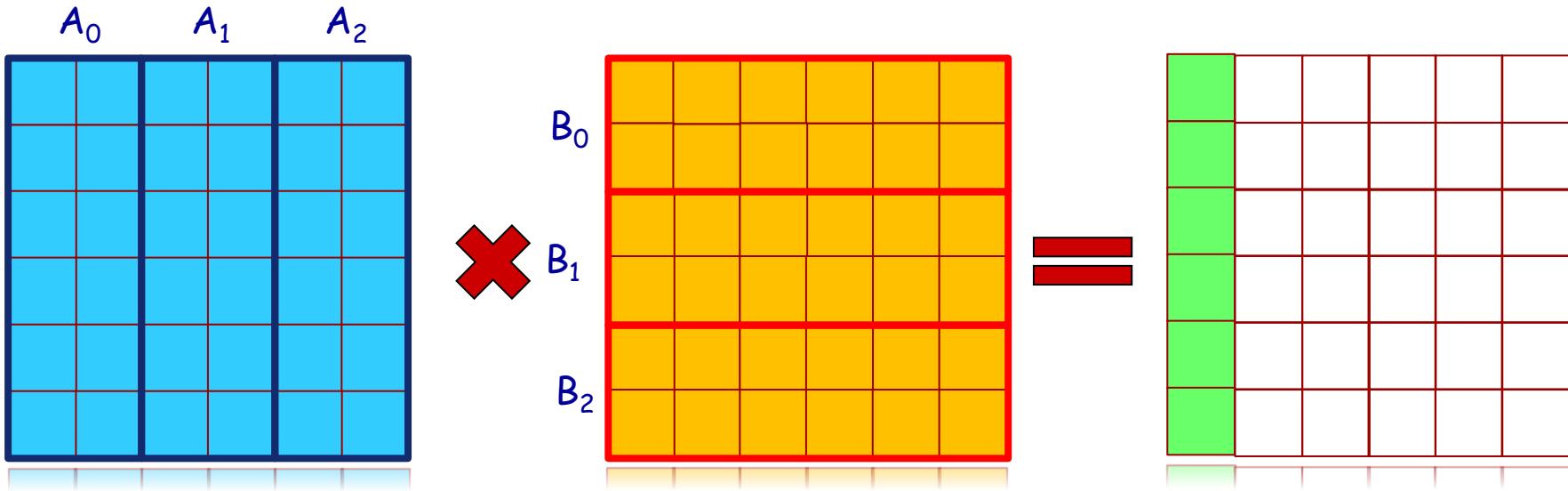
Prodotti matrice-matrice  
di dimensione minore!

```
begin
    C=0
    for i=0 to p-1 do
        C=AiBi
        C=C+Ci
    endfor
end
```

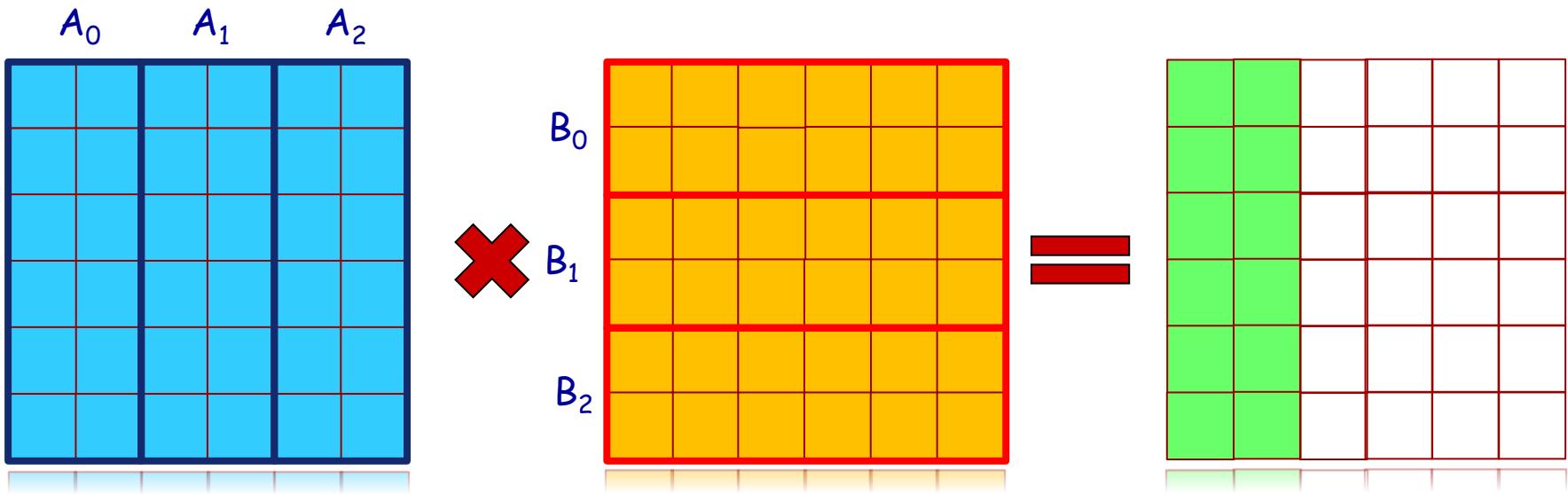
# Algoritmo a blocchi - II strategia



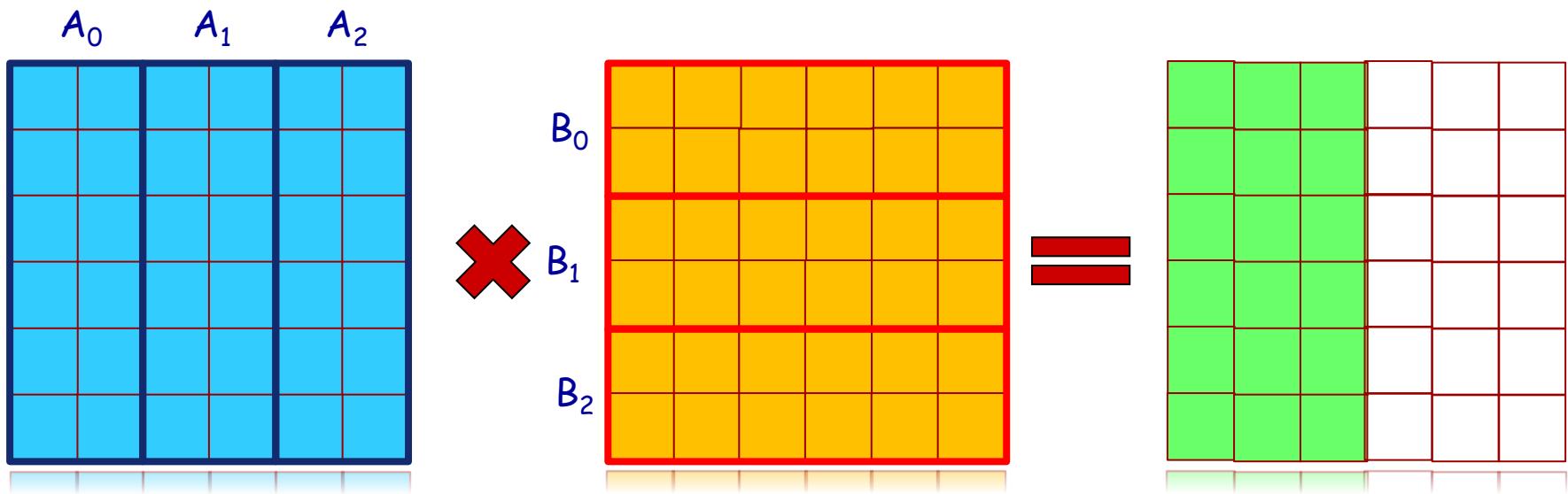
# Algoritmo a blocchi - II strategia



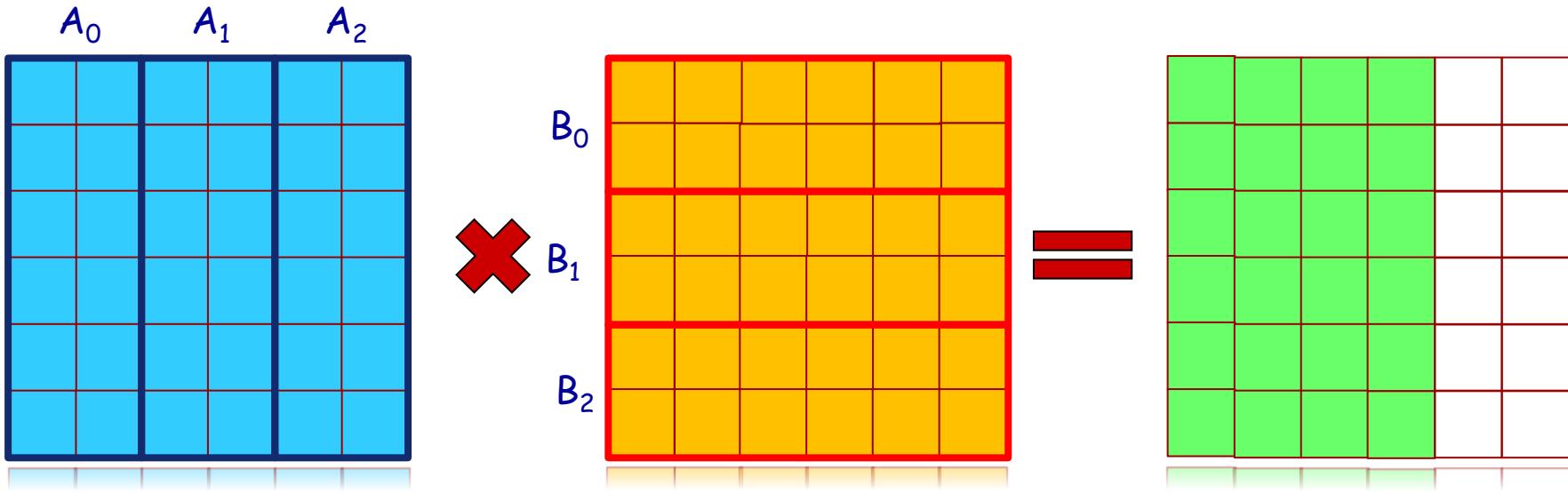
# Algoritmo a blocchi - II strategia



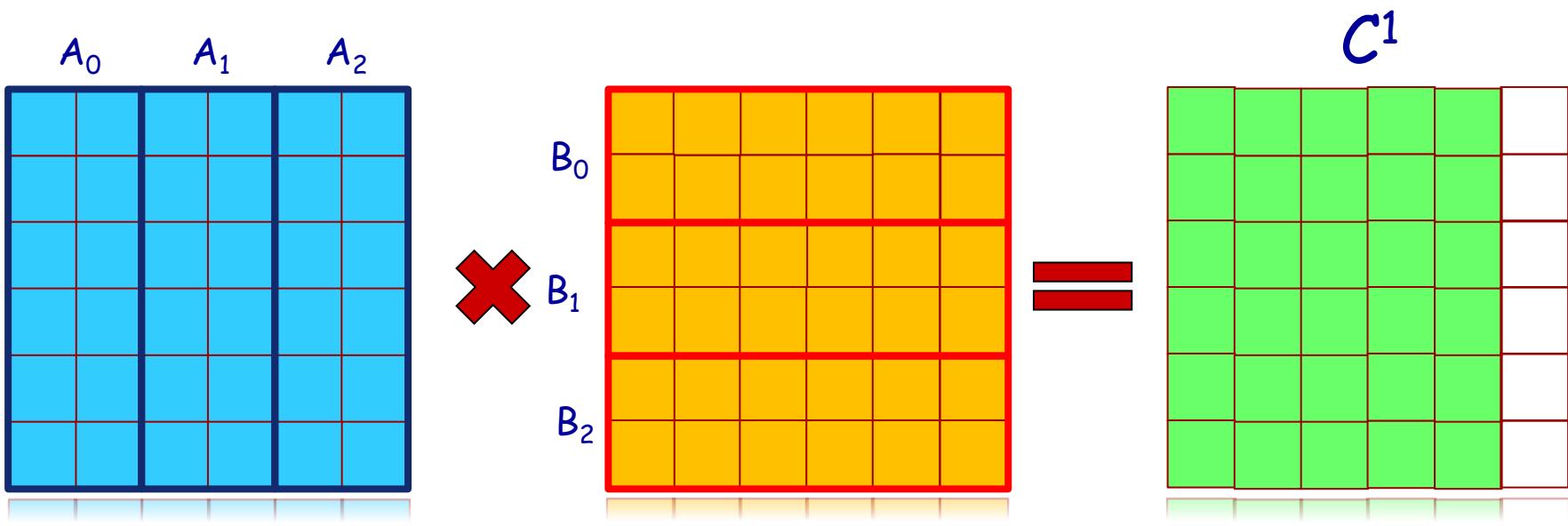
# Algoritmo a blocchi - II strategia



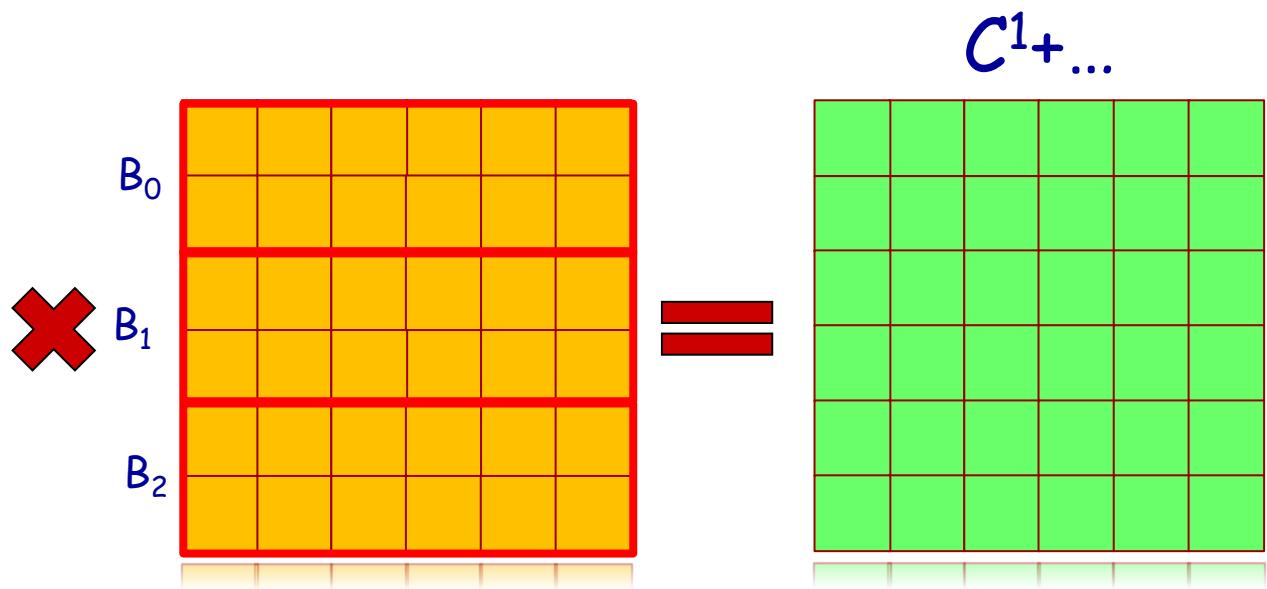
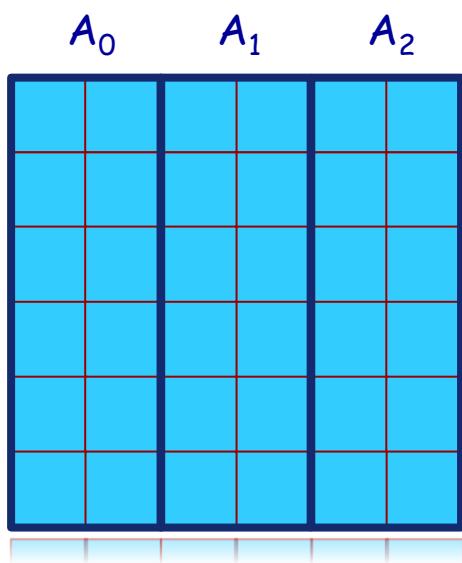
# Algoritmo a blocchi - II strategia



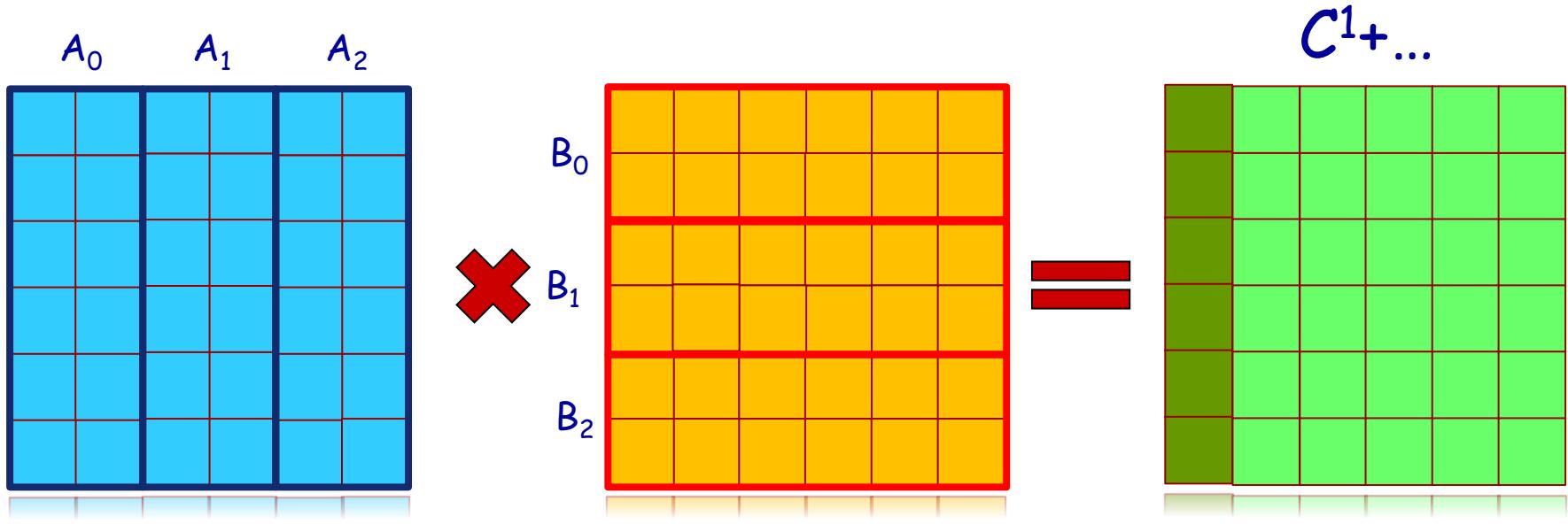
# Algoritmo a blocchi - II strategia



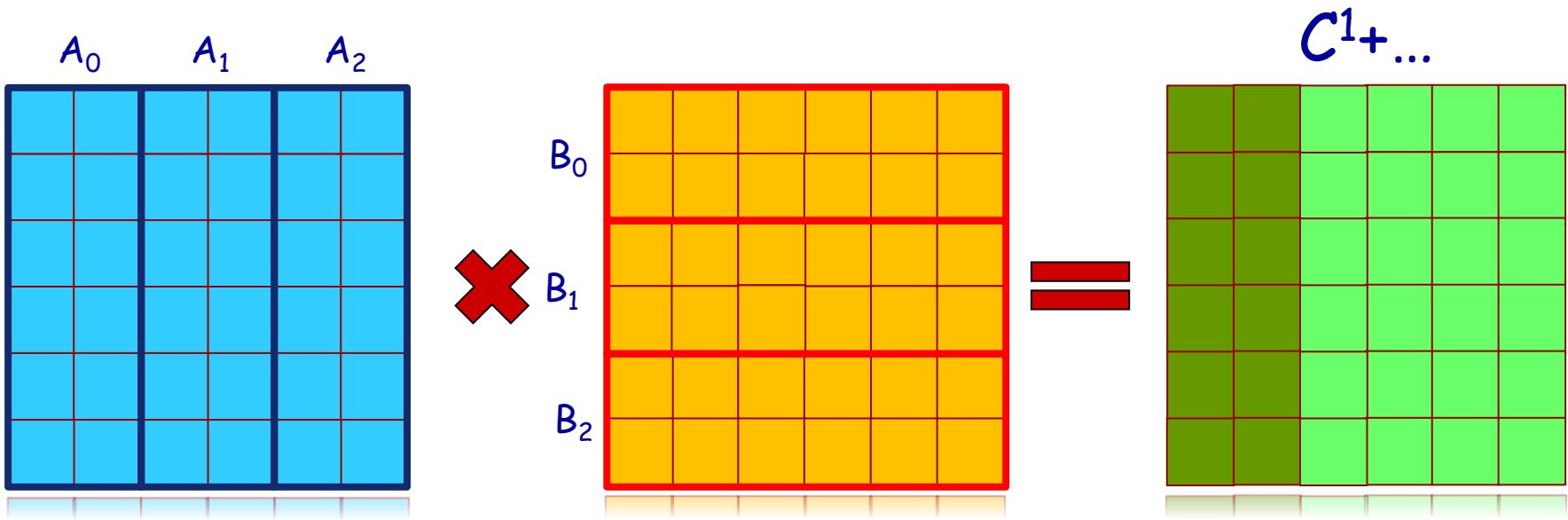
# Algoritmo a blocchi - II strategia



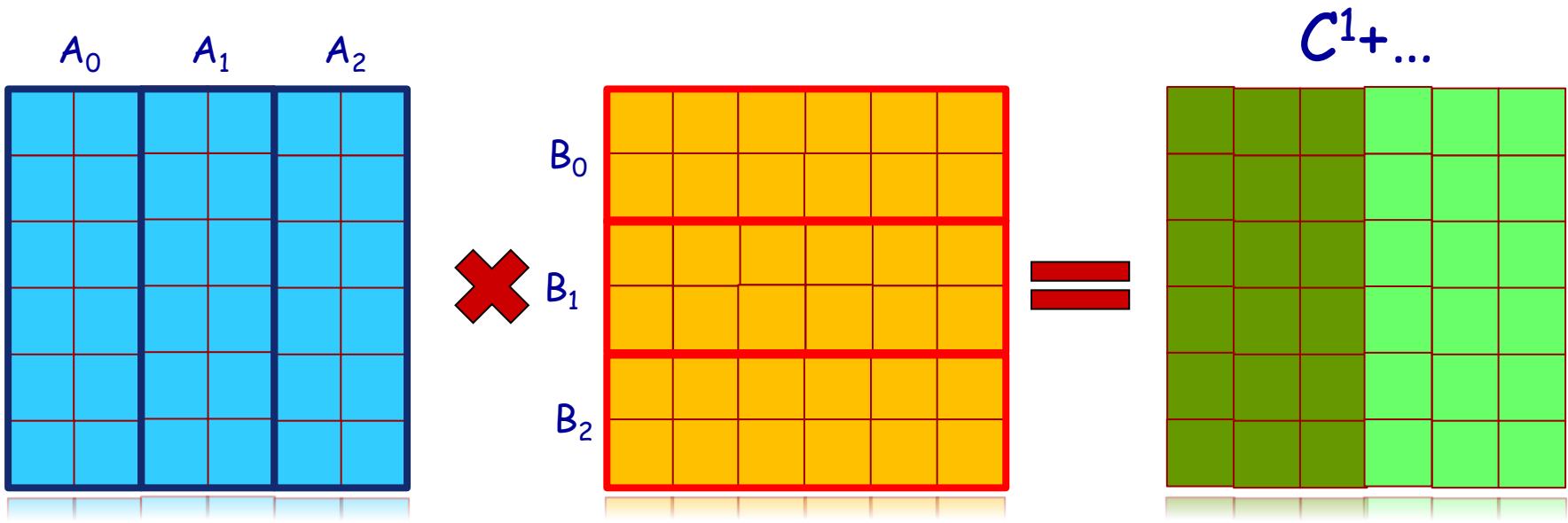
# Algoritmo a blocchi - II strategia



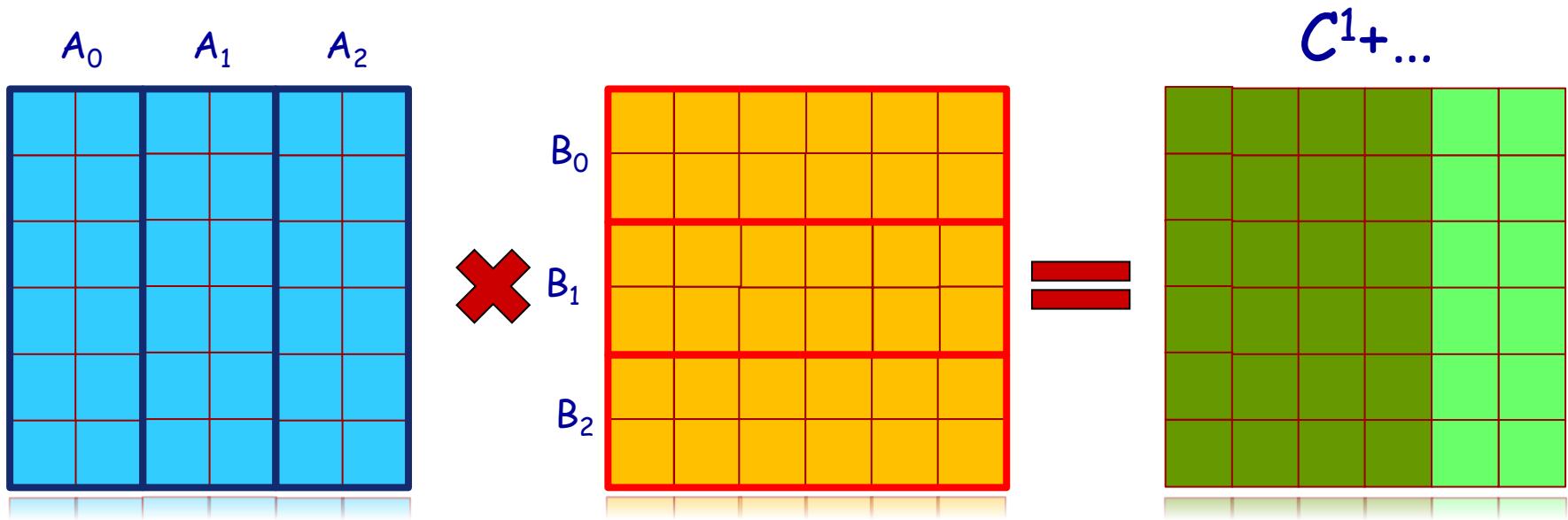
# Algoritmo a blocchi - II strategia



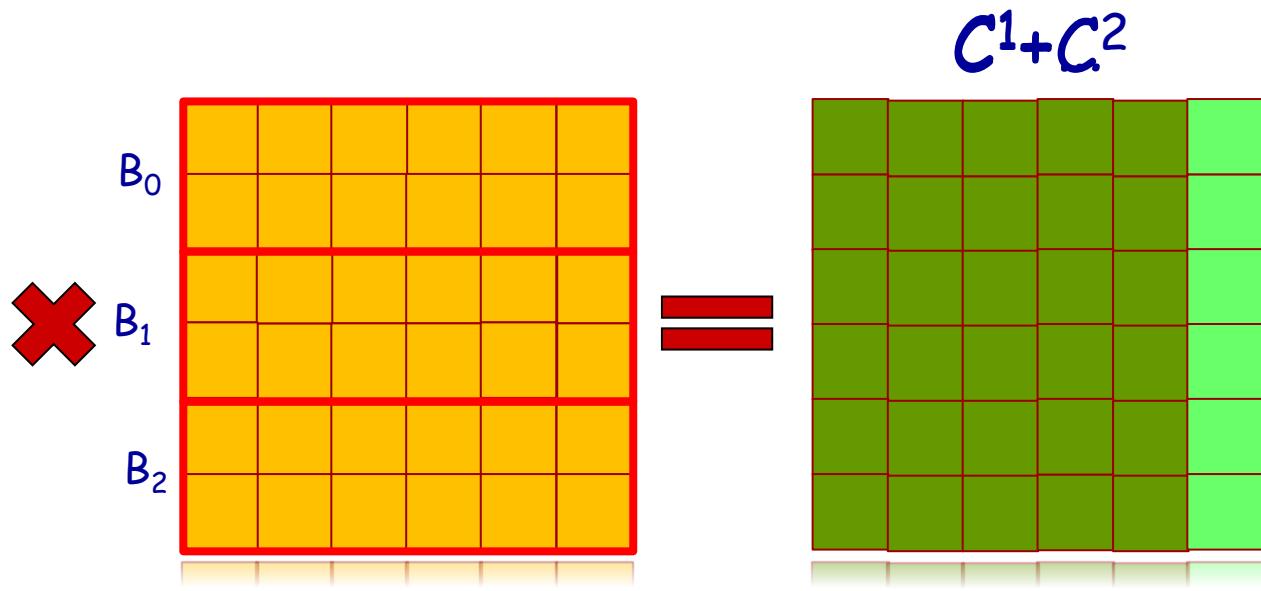
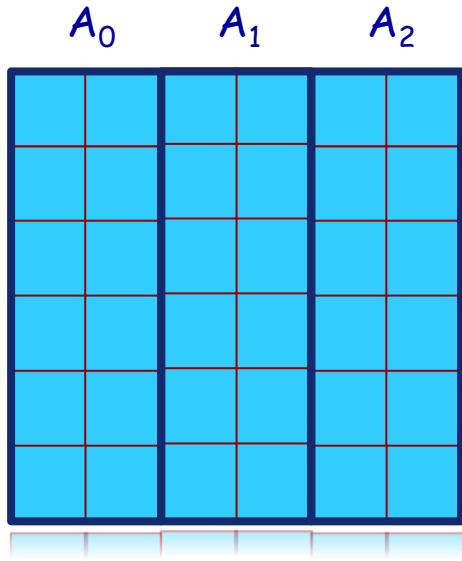
# Algoritmo a blocchi - II strategia



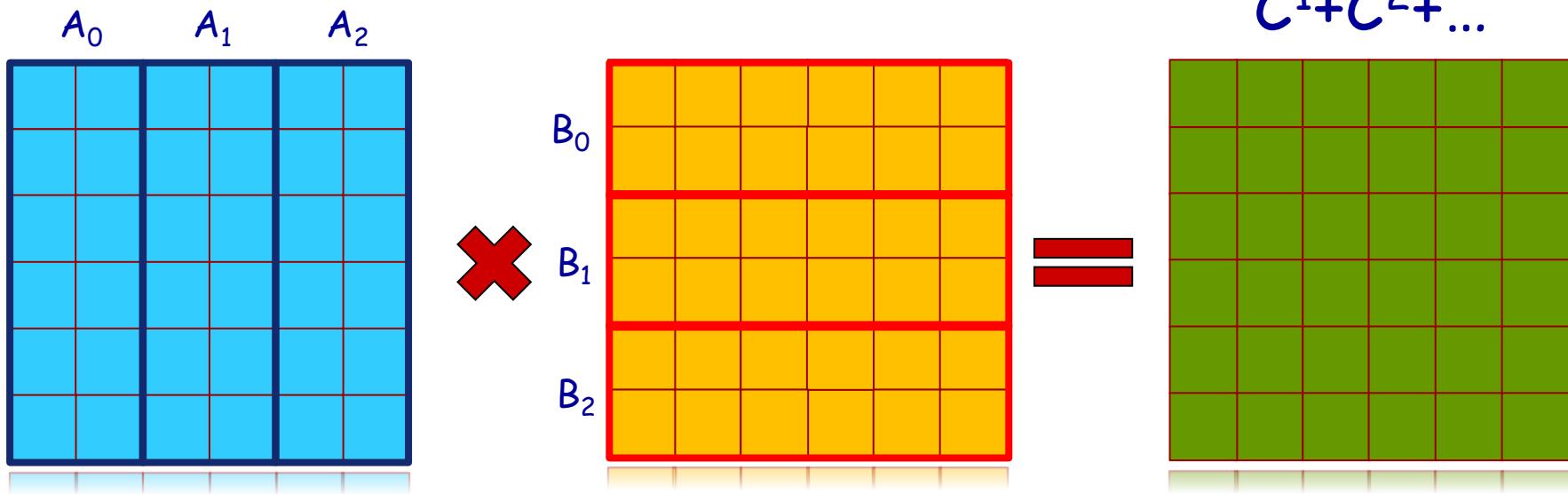
# Algoritmo a blocchi - II strategia



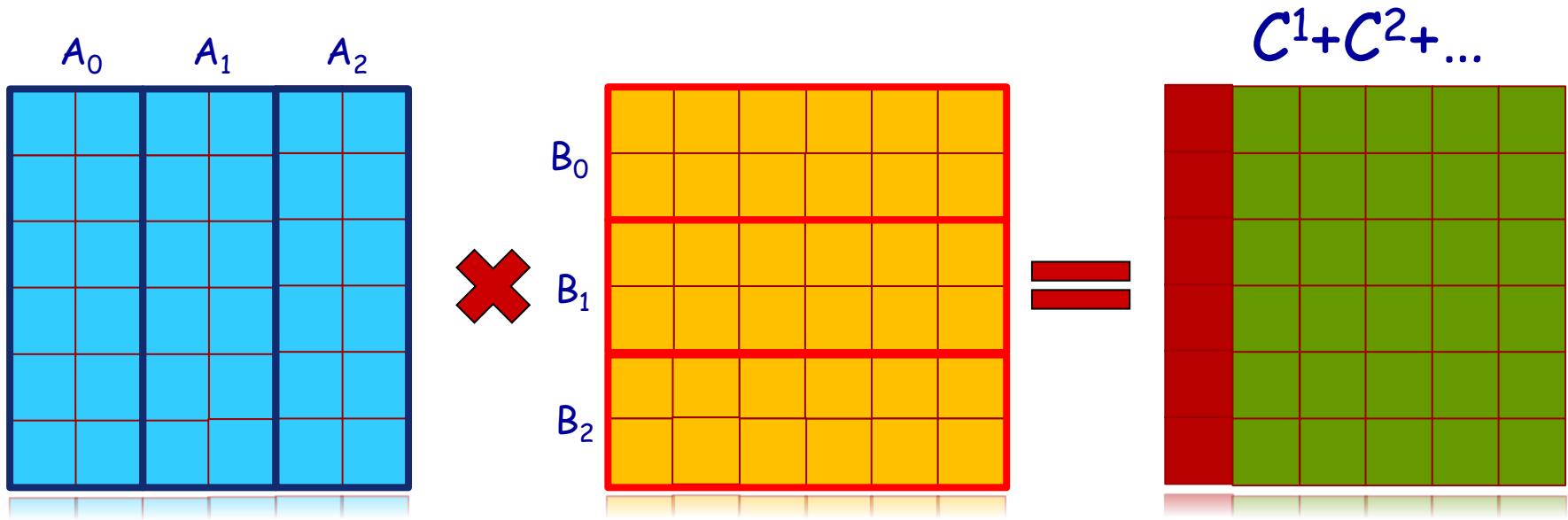
# Algoritmo a blocchi - II strategia



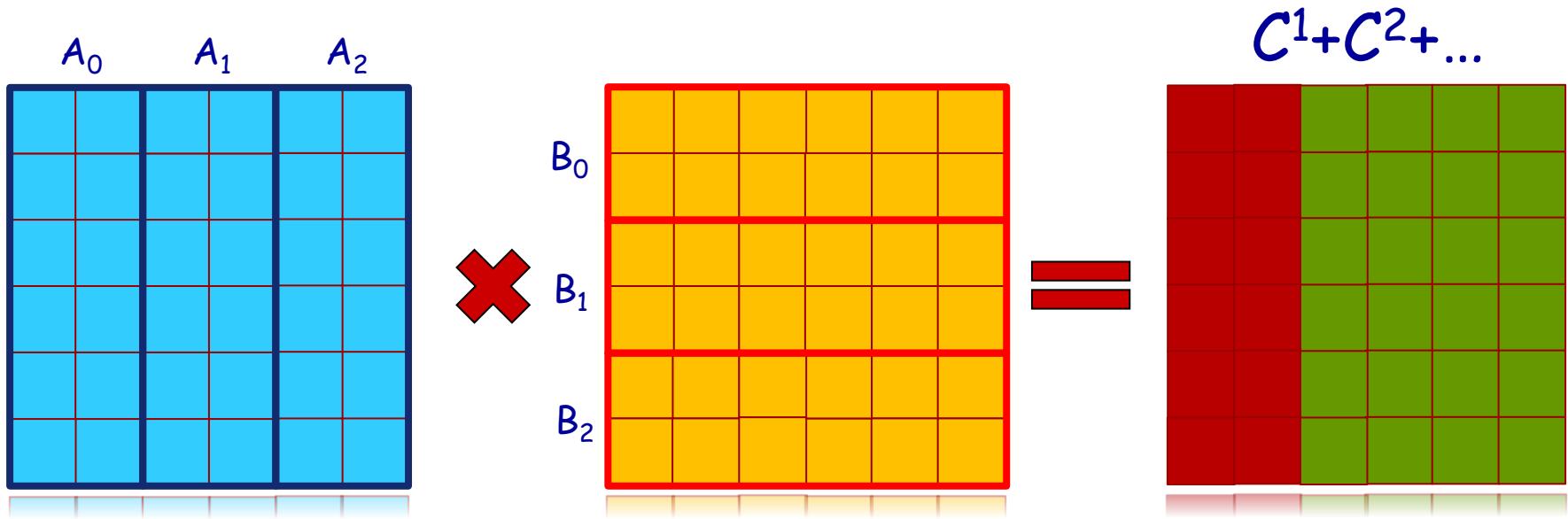
# Algoritmo a blocchi - II strategia



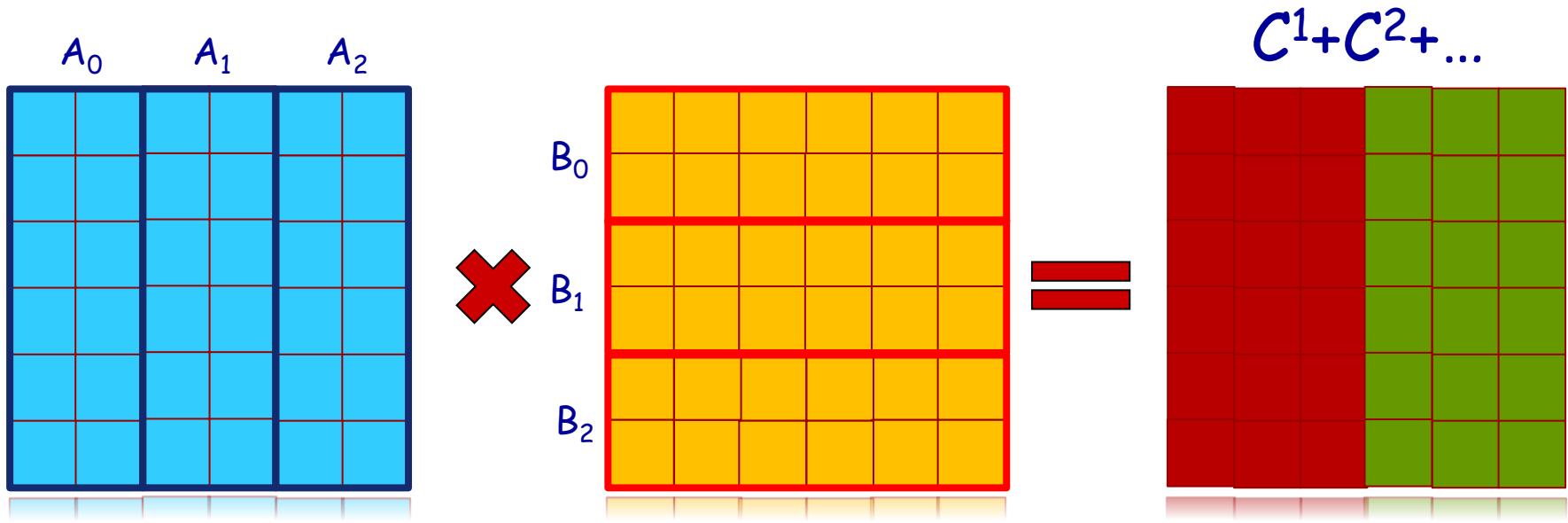
# Algoritmo a blocchi - II strategia



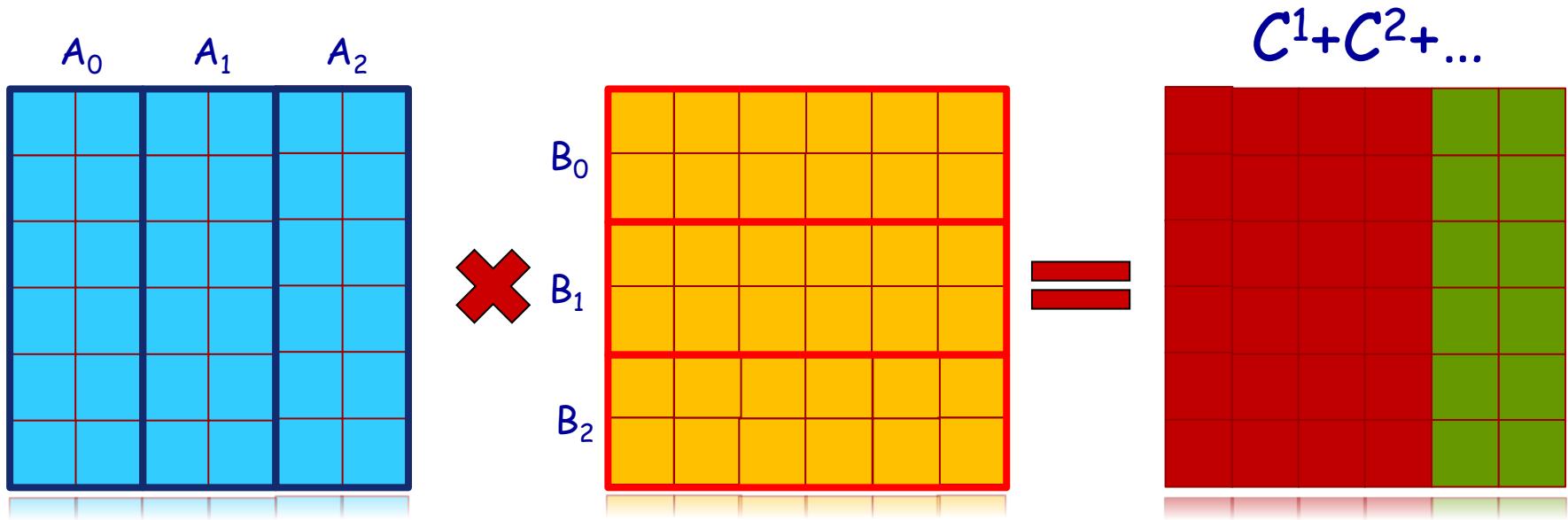
# Algoritmo a blocchi - II strategia



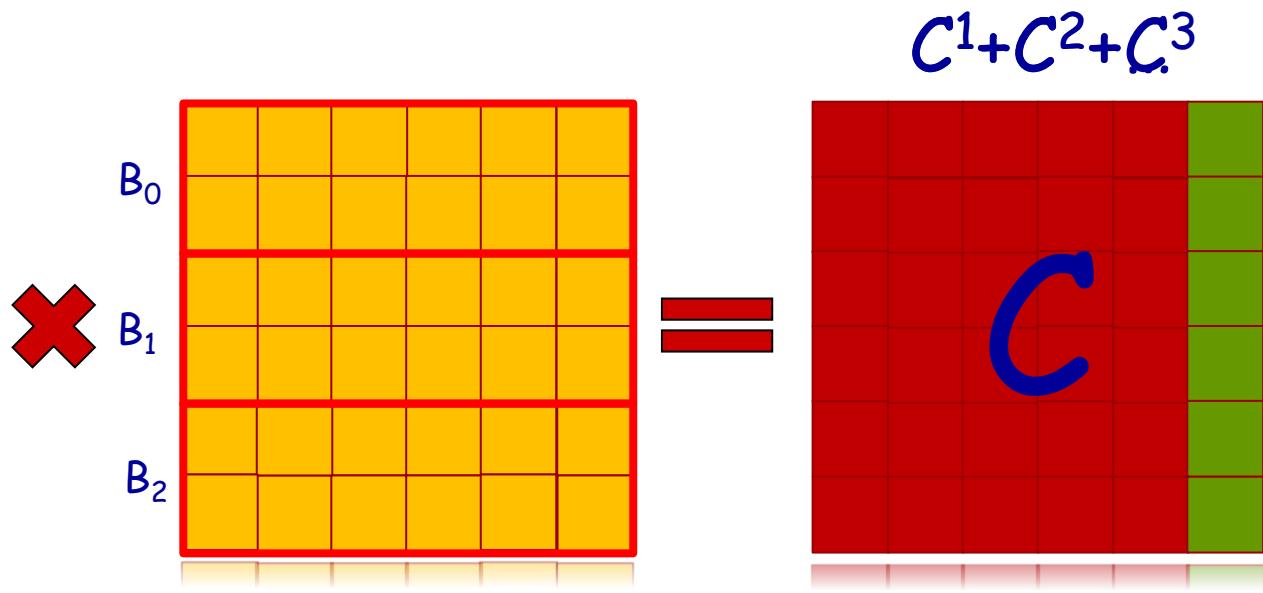
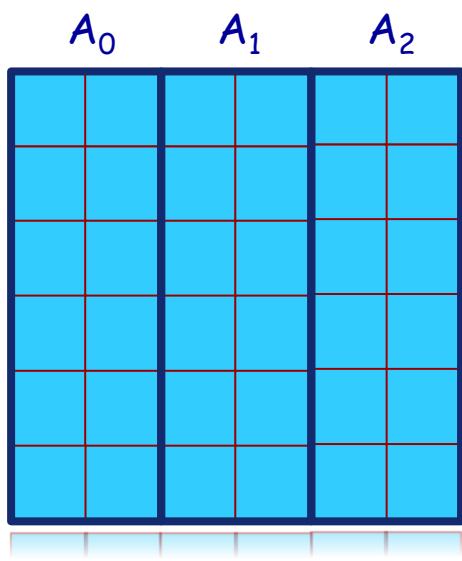
# Algoritmo a blocchi - II strategia



# Algoritmo a blocchi - II strategia



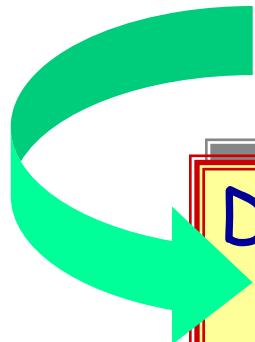
# Algoritmo a blocchi - II strategia



# Introduzione del parallelismo

---

Distribuiamo tra i processori il calcolo dei diversi prodotti matrice-matrice



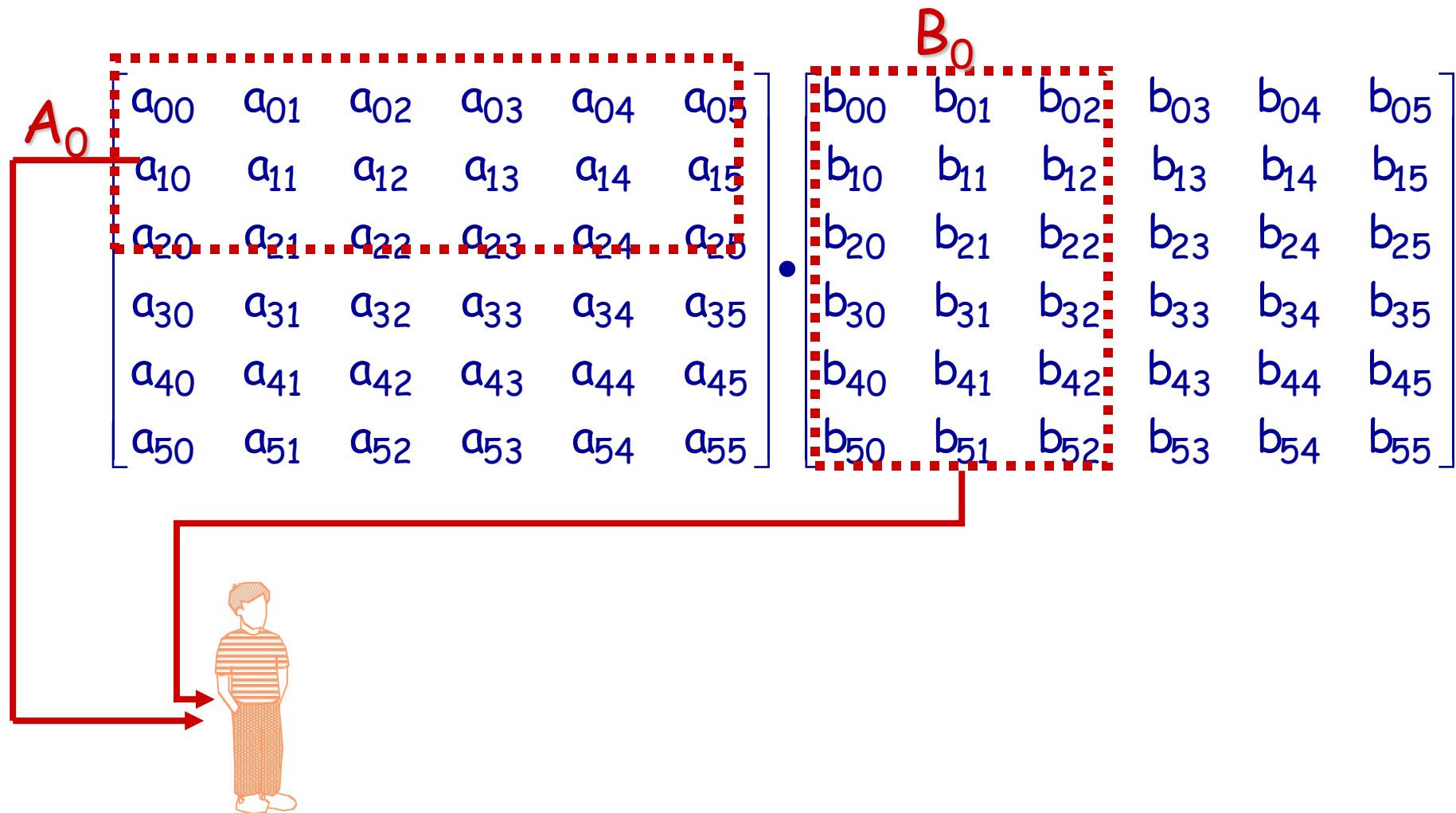
Distribuiamo tra i processori gli elementi delle 2 matrici

# I STRATEGIA

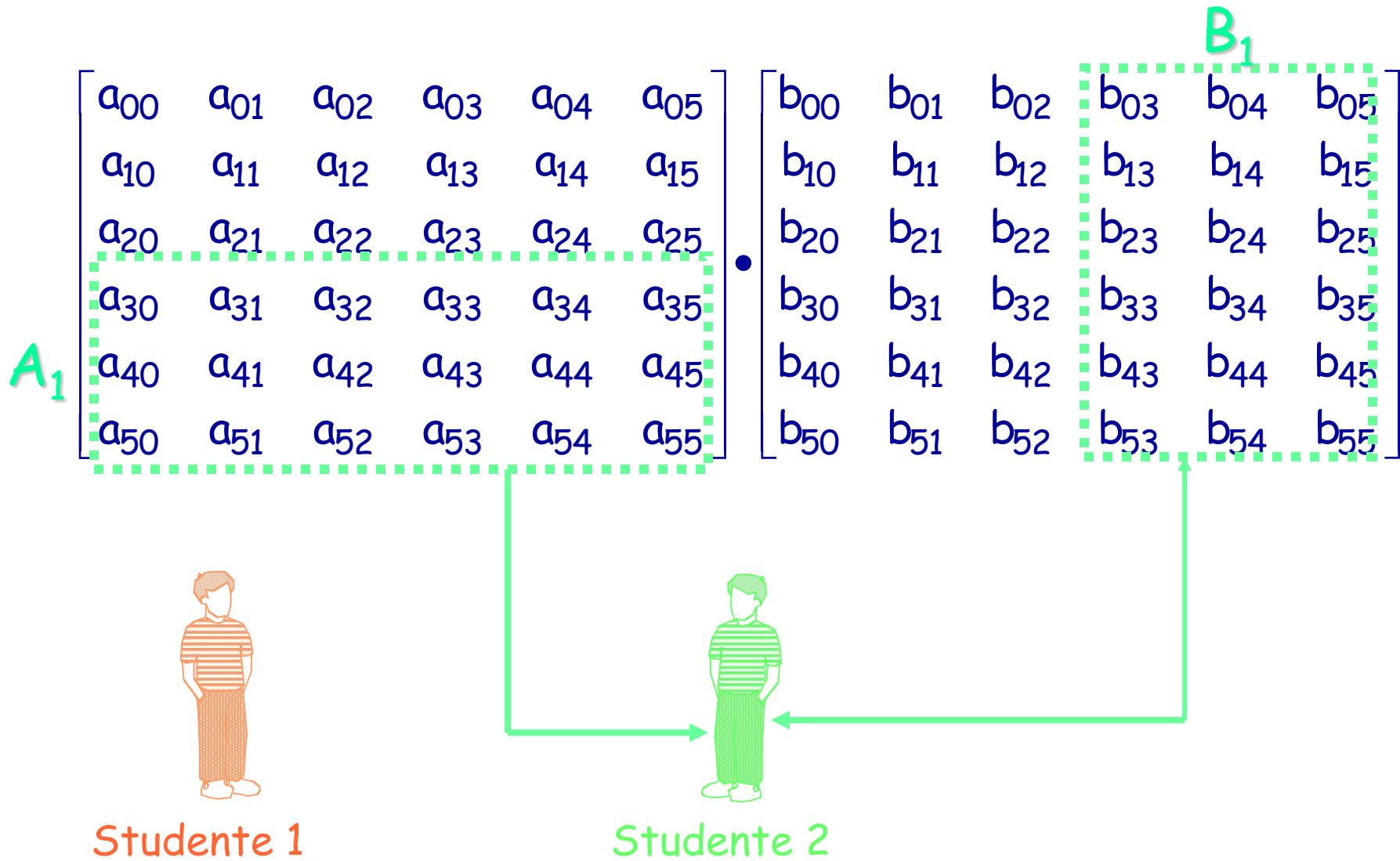
---

suddividiamo  
la matrice A in  
blocchi di RIGHE  
e la matrice B in  
blocchi di COLONNE

# Distribuzione dei dati: Esempio n=6



# Distribuzione dei dati: Esempio n=6



# Domanda

---

Con i dati così distribuiti  
cosa può calcolare  
ciascuno studente

?

# I Strategia: Esempio n=6

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \end{bmatrix} \bullet \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \\ b_{30} & b_{31} & b_{32} \\ b_{40} & b_{41} & b_{42} \\ b_{50} & b_{51} & b_{50} \end{bmatrix} = \begin{bmatrix} c_{00} & c_{01} & c_{02} & ? & ? & ? \\ c_{10} & c_{11} & c_{12} & ? & ? & ? \\ c_{20} & c_{21} & c_{22} & ? & ? & ? \\ ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & ? \end{bmatrix}$$

$A_0$        $B_0$        $C_{00}$



Studente 1

Lo studente 1  
può calcolare  
solo alcune componenti  
della matrice C!

# I Strategia: Esempio n=6

$$\begin{bmatrix} a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix} \bullet \begin{bmatrix} b_{03} & b_{04} & b_{05} \\ b_{13} & b_{14} & b_{15} \\ b_{23} & b_{24} & b_{25} \\ b_{33} & b_{34} & b_{35} \\ b_{43} & b_{44} & b_{45} \\ b_{53} & b_{54} & b_{55} \end{bmatrix} = \begin{bmatrix} ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & ? \\ ? & ? & ? & c_{33} & c_{34} & c_{35} \\ ? & ? & ? & c_{43} & c_{44} & c_{45} \\ ? & ? & ? & c_{53} & c_{54} & c_{55} \end{bmatrix}$$

**A<sub>1</sub>**

**B<sub>1</sub>**

**C<sub>11</sub>**

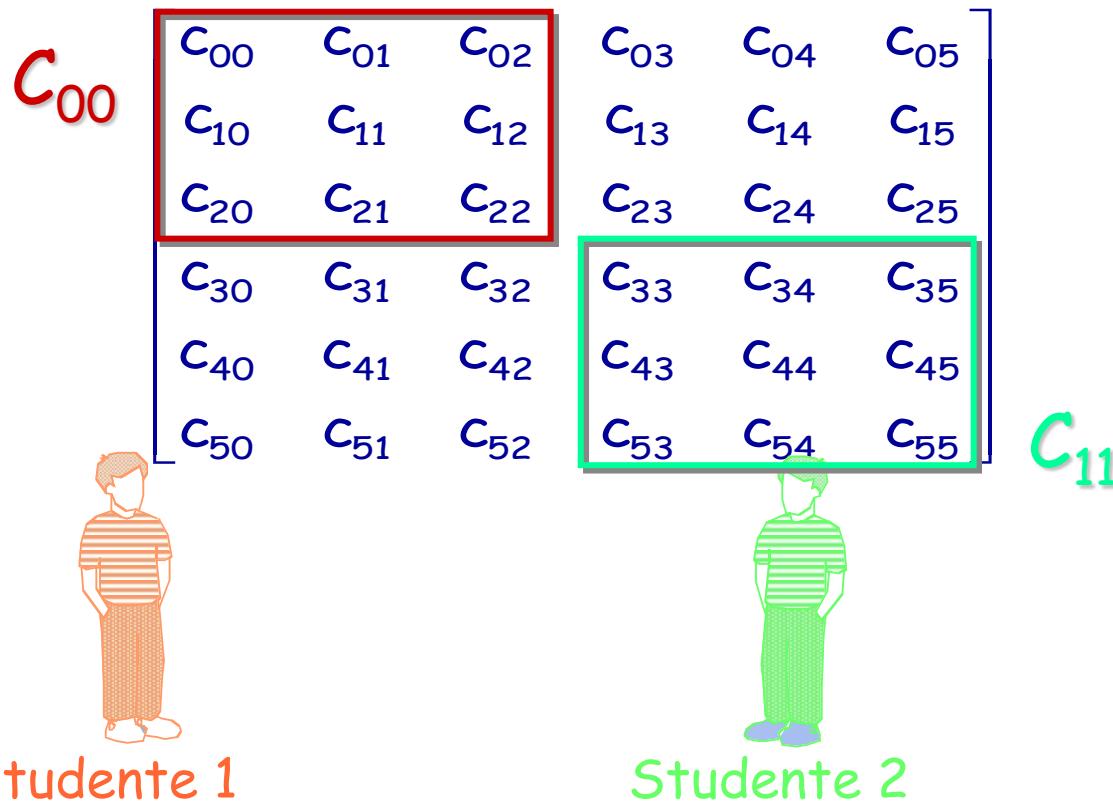
Lo studente 2  
può calcolare  
solo alcune componenti  
della matrice C!

Studente 2



# Osservazione

Con la distribuzione effettuata  
dei blocchi di A e B  
gli studenti hanno calcolato  
solo alcune componenti di C!



# Domanda

Come calcolare le altre componenti di  $C$ ?

$C_{00}$	$C_{01}$	$C_{02}$	$C_{03}$	$C_{04}$	$C_{05}$	$C_{01}$
$C_{10}$	$C_{11}$	$C_{12}$	$C_{13}$	$C_{14}$	$C_{15}$	
$C_{20}$	$C_{21}$	$C_{22}$	$C_{23}$	$C_{24}$	$C_{25}$	
$C_{30}$	$C_{31}$	$C_{32}$	$C_{33}$	$C_{34}$	$C_{35}$	
$C_{40}$	$C_{41}$	$C_{42}$	$C_{43}$	$C_{44}$	$C_{45}$	
$C_{50}$	$C_{51}$	$C_{52}$	$C_{53}$	$C_{54}$	$C_{55}$	



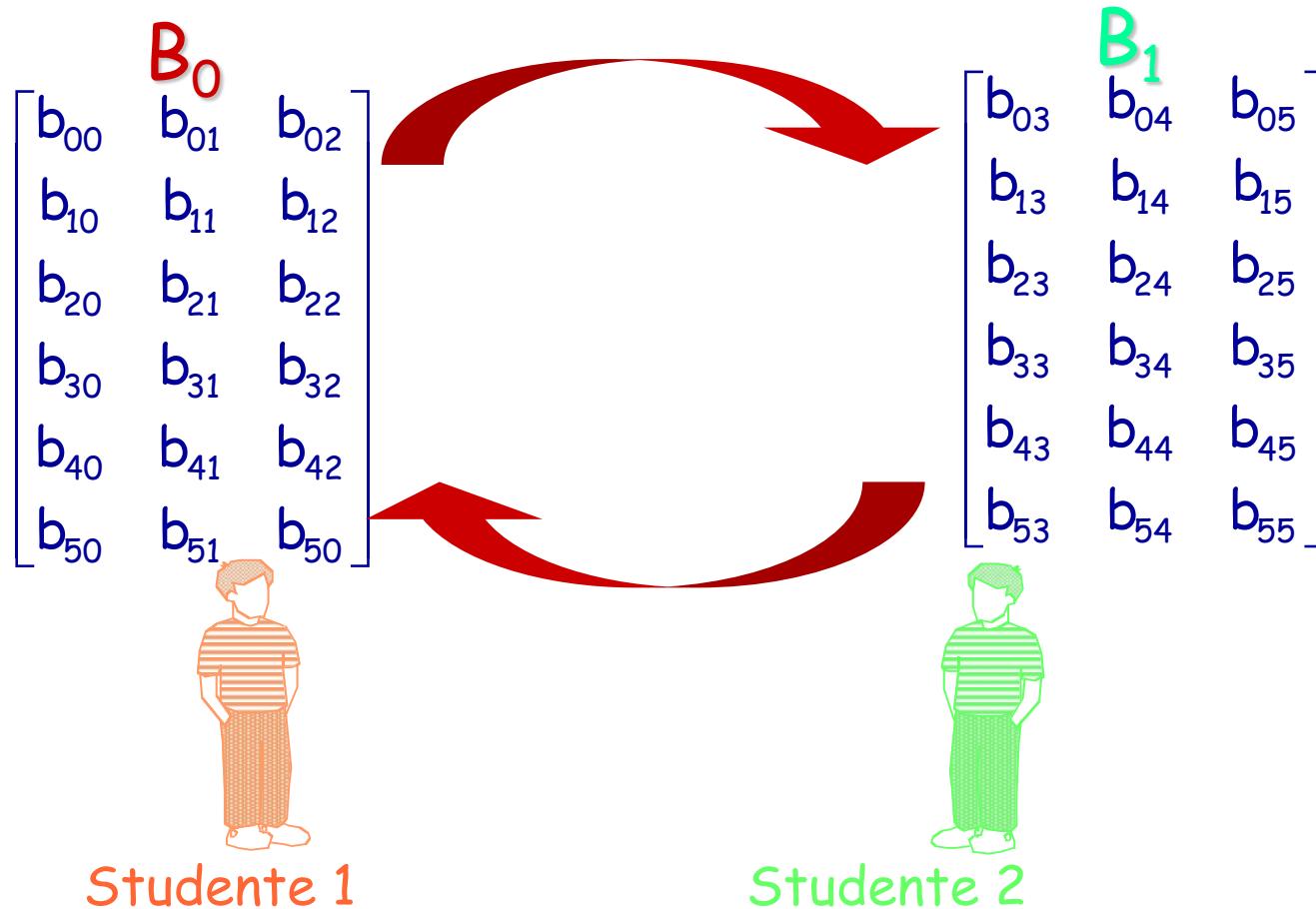
Studente 1



Studente 2

# IDEA!

Gli studenti possono "interagire"  
scambiandosi  
i blocchi di COLONNE della matrice B!



# IDEA!

Gli studenti possono "interagire"  
scambiandosi  
i blocchi di COLONNE della matrice B!

$B_0$

$$\begin{bmatrix} b_{03} & b_{04} & b_{05} \\ b_{13} & b_{14} & b_{15} \\ b_{23} & b_{24} & b_{25} \\ b_{33} & b_{34} & b_{35} \\ b_{43} & b_{44} & b_{45} \\ b_{53} & b_{54} & b_{55} \end{bmatrix}$$



Studente 1

$B_1$

$$\begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \\ b_{30} & b_{31} & b_{32} \\ b_{40} & b_{41} & b_{42} \\ b_{50} & b_{51} & b_{50} \end{bmatrix}$$



Studente 2

# Domanda

---

Dopo lo scambio  
dei blocchi di B  
cosa può calcolare  
ciascuno studente

?

# I Strategia: Esempio n=6

Lo studente 1 ha ora anche l'altro blocco di B!

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \end{bmatrix} \bullet \begin{array}{c} B_0 \\ B_1 \end{array} = \begin{array}{c} C_{00} \\ C_{01} \end{array}$$

The diagram shows a 3x6 matrix  $A_0$  multiplied by a 2x6 matrix  $B$ , resulting in a 3x6 matrix  $C$ . Matrix  $B$  is partitioned into two 3x3 blocks:  $B_0$  (top-left) and  $B_1$  (bottom-right). Matrix  $C$  is partitioned into two 3x3 blocks:  $C_{00}$  (top-left) and  $C_{01}$  (bottom-right). The first two columns of  $A_0$  are multiplied by  $B_0$ , and the last four columns are multiplied by  $B_1$ . The result is a matrix where the first two columns are  $C_{00}$  and the last four are  $C_{01}$ . Red boxes highlight the relevant parts of  $A_0$ ,  $B_0$ ,  $B_1$ ,  $C_{00}$ , and  $C_{01}$ .



Studente 1

Lo studente 1

può calcolare

ALTRÉ componenti

della matrice  $C$ : il blocco  $C_{01}$  !

# I Strategia: Esempio n=6

Lo studente 2 ha ora anche l'altro blocco di B!

$$\begin{bmatrix} a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix} \bullet \begin{array}{c} B_0 \\ B_1 \end{array} = \begin{bmatrix} ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & ? \end{bmatrix} = \begin{array}{|c|c|} \hline C_{30} & C_{31} & C_{32} & C_{33} & C_{34} & C_{35} \\ \hline C_{40} & C_{41} & C_{42} & C_{43} & C_{44} & C_{45} \\ \hline C_{50} & C_{51} & C_{52} & C_{53} & C_{54} & C_{55} \\ \hline \end{array} \quad C_{11}$$

$A_1$

$B_0$

$B_1$

$C_{10}$

Lo studente 2 può calcolare ALTRE componenti

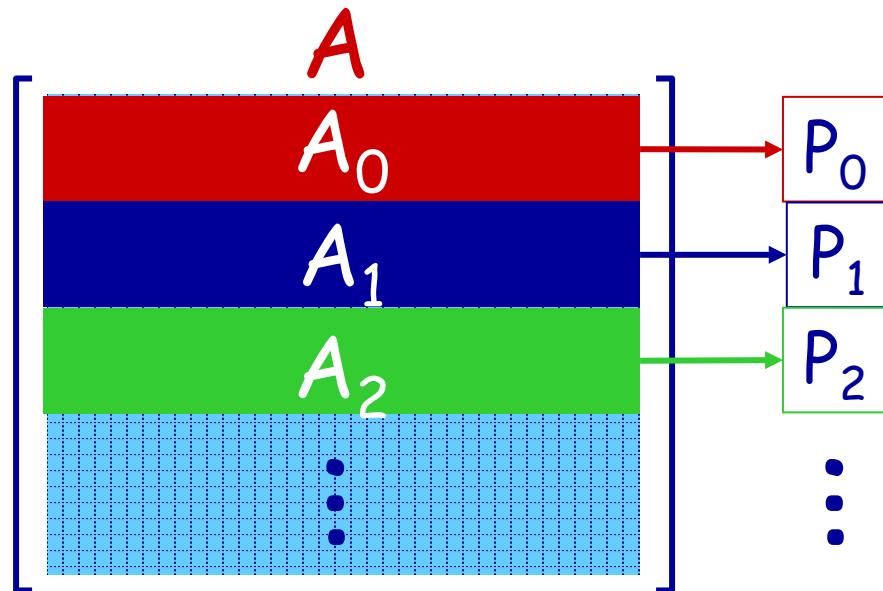
Studente 2 della matrice C: il blocco  $C_{10}$ !



# I STRATEGIA: In generale

## I passo: decomposizione del problema

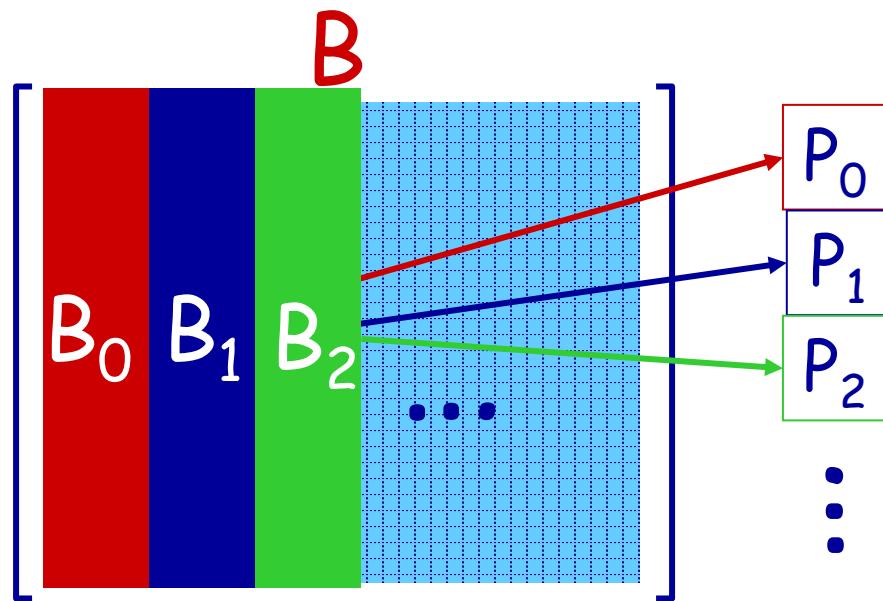
La matrice  $A$  viene distribuita  
in BLOCCHI di RIGHE  
fra p processori



# I STRATEGIA: In generale

## I passo: decomposizione del problema

La matrice  $B$  viene distribuita  
in BLOCCHI di COLONNE  
fra p processori



# I STRATEGIA: In generale

---

## II passo: risoluzione dei sottoproblemi

Il prodotto  $A \cdot B = C$  viene decomposto  
in  $p$  prodotti del tipo

$$A_i \cdot B_j = C_{ij}$$

Ciascun processore calcola  
 $p$  prodotti matrice-matrice  
(di dimensione più piccola di quello assegnato!).

# I strategia: caratteristiche

---

- ◆ Analogico localmente alla I strategia del prodotto matrice-vettore

MA

- ◆ sono necessari degli scambi di colonne della matrice B tra i processori

# Domanda

---

Qual è l'algoritmo parallelo  
della I Strategia  
di decomposizione

?

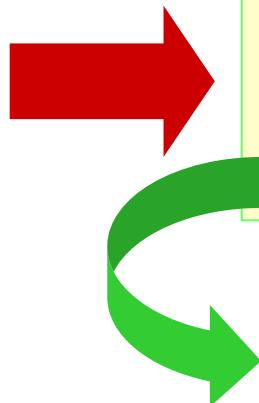
# Risposta

Partizionamento di  
A in blocchi di righe  
B in blocchi di colonne



Algoritmo a blocchi

```
begin
  for i=0 to p-1 do
    for j=0 to p-1 do
       $C_{ij} = A_i \cdot B_j$ 
    endfor
  endfor
end
```



Distribuzione dei  
blocchi fra i  
processori

Algoritmo parallelo

Parallelizzazione dell'algoritmo a  
blocchi!

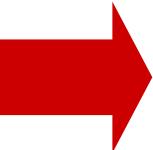
# Risposta

Partizionamento di  
A in blocchi di righe  
B in blocchi di colonne



## Algoritmo a blocchi

```
begin
  for i=0 to p-1 do
    for j=0 to p-1 do
       $C_{ij} = A_i \cdot B_j$ 
    endfor
  endfor
end
```



## Algoritmo parallelo

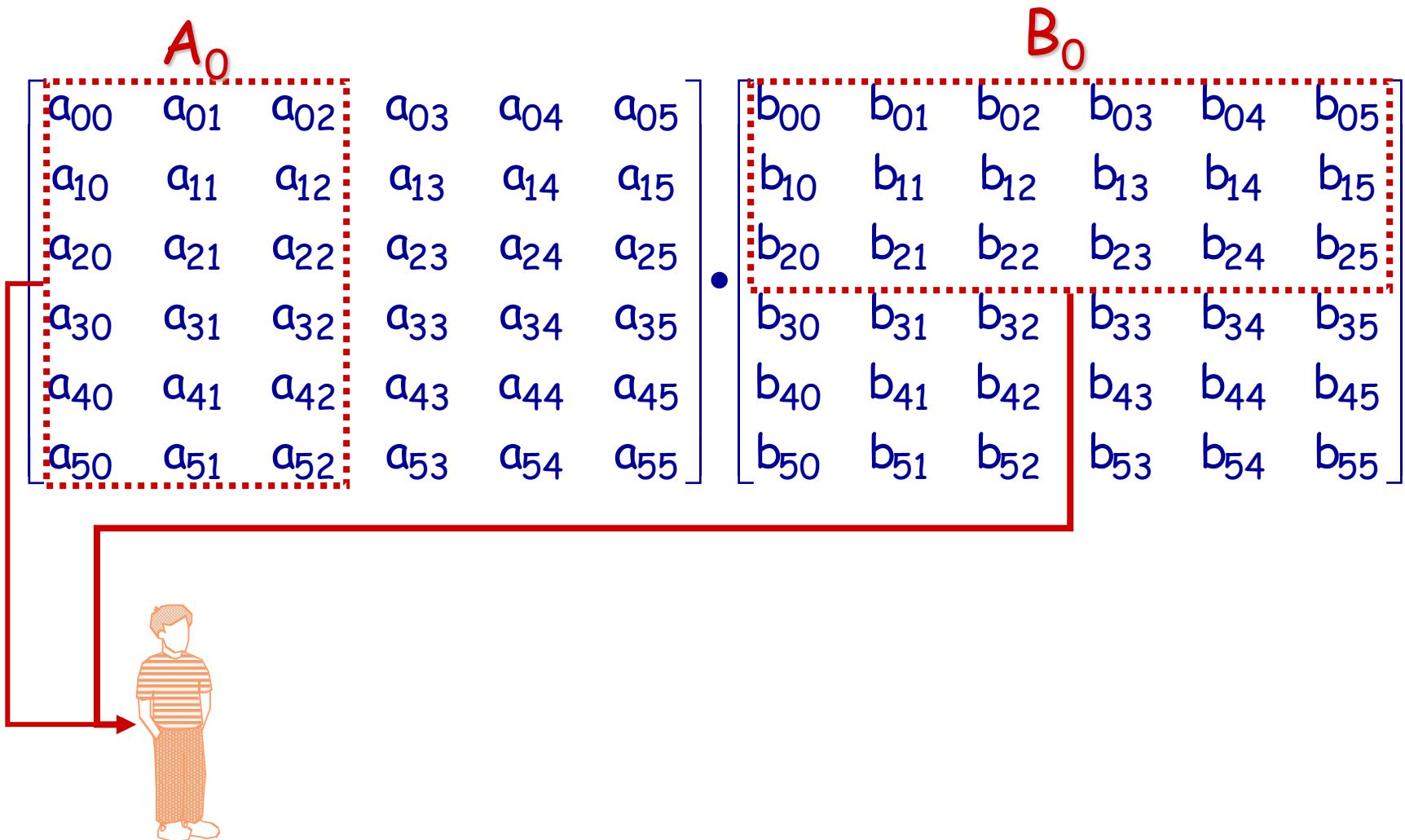
```
Begin
  for k=0 to p-1 do
    forall  $P_i$  ( $i=0, \dots, p-1$ )
       $j = \text{mod}(i+k, p)$ ;
      {  $P_i$  calcola  $C_{ij} = A_i \cdot B_j$  }
      send( $B_j$ ,  $P_{i+1}$ ) { $p=0$ }
      recv( $B_{j-1}$ ,  $P_{i-1}$ ) {-1=p-1}
    endfor
  end
```

## II strategia

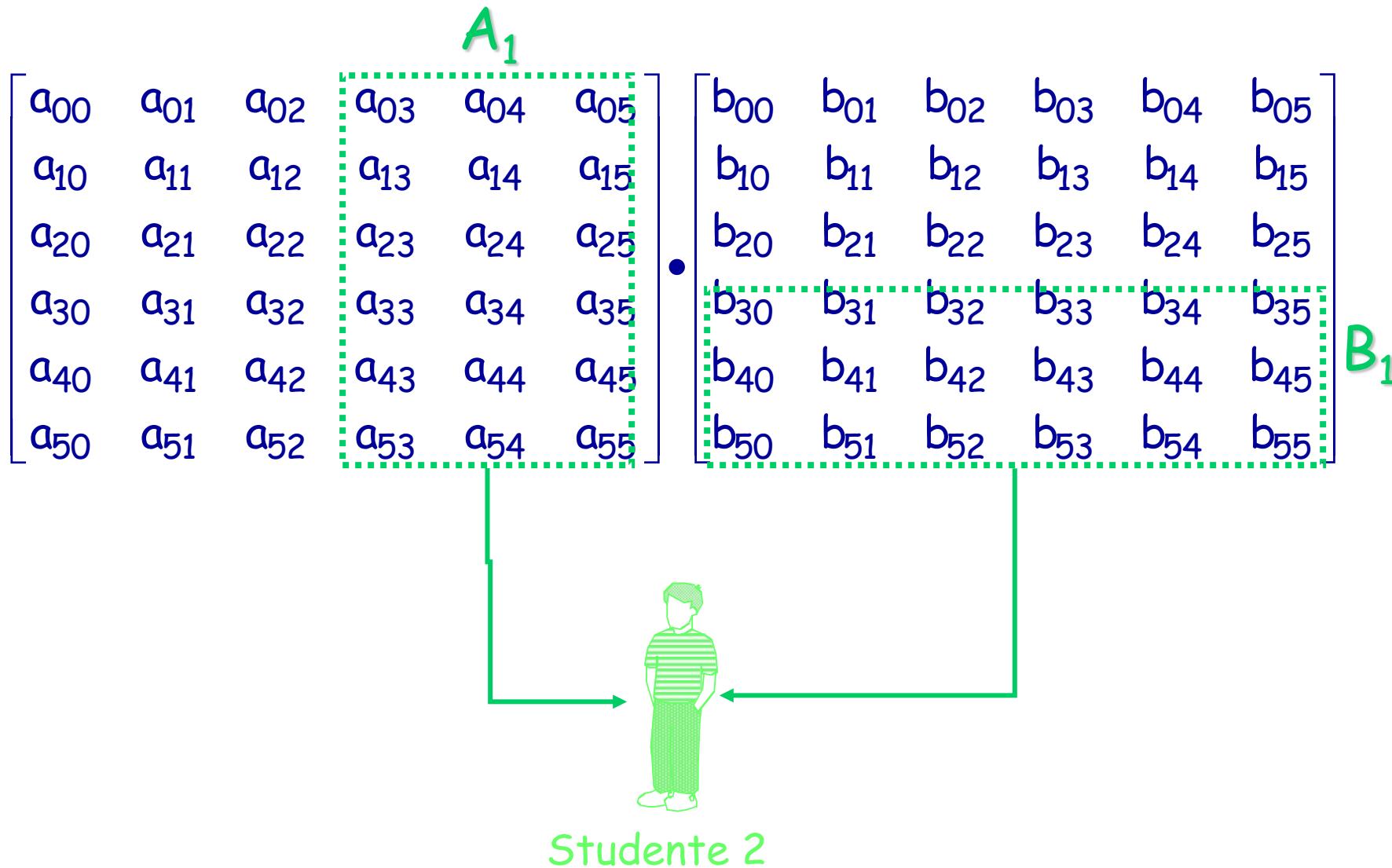
---

suddividiamo  
la matrice A in  
blocchi di **COLONNE**  
e la matrice B in  
blocchi di **RIGHE**

# Distribuzione dei dati: Esempio n=6



# Distribuzione dei dati: Esempio n=6



# Domanda

---

Con i dati così distribuiti  
cosa può calcolare  
ciascuno studente

?

## II Strategia: Esempio n=6

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \\ a_{30} & a_{31} & a_{32} \\ a_{40} & a_{41} & a_{42} \\ a_{50} & a_{51} & a_{50} \end{bmatrix} \bullet \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} & b_{04} & b_{05} \\ b_{10} & b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ b_{20} & b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \end{bmatrix} = [?]$$



Studente 1

*quali componenti  
della matrice C calcola lo  
studente 1 ?*

## II Strategia: Esempio n=6

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \\ a_{30} & a_{31} & a_{32} \\ a_{40} & a_{41} & a_{42} \\ a_{50} & a_{51} & a_{50} \end{bmatrix} \bullet \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} & b_{04} & b_{05} \\ b_{10} & b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ b_{20} & b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \end{bmatrix} = \begin{bmatrix} c_{00}^1 & c_{01}^1 & c_{02}^1 & c_{03}^1 & c_{04}^1 & c_{05}^1 \\ c_{10}^1 & c_{11}^1 & c_{12}^1 & c_{13}^1 & c_{14}^1 & c_{15}^1 \\ c_{20}^1 & c_{21}^1 & c_{22}^1 & c_{23}^1 & c_{24}^1 & c_{25}^1 \\ c_{30}^1 & c_{31}^1 & c_{32}^1 & c_{33}^1 & c_{34}^1 & c_{35}^1 \\ c_{40}^1 & c_{41}^1 & c_{42}^1 & c_{43}^1 & c_{44}^1 & c_{45}^1 \\ c_{50}^1 & c_{51}^1 & c_{52}^1 & c_{53}^1 & c_{54}^1 & c_{55}^1 \end{bmatrix}$$

 Studente 1

Lo studente 1 può calcolare  
un contributo per OGNI  
elemento della matrice C

## II Strategia: Esempio n=6

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \\ a_{30} & a_{31} & a_{32} \\ a_{40} & a_{41} & a_{42} \\ a_{50} & a_{51} & a_{50} \end{bmatrix} \bullet \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} & b_{04} & b_{05} \\ b_{10} & b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ b_{20} & b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \end{bmatrix} = \boxed{\begin{matrix} C_1 \end{matrix}}$$

 Studente 1

Lo studente 1 può calcolare  
un contributo per OGNI  
elemento della matrice C

# Cioè...

---

i,j=0,5

$$c_{ij}^1 = \sum_{k=0}^2 a_{ik} b_{kj}$$

Lo studente 1 può calcolare  
un contributo per OGNI  
elemento della matrice C



Studente 1

k = Indice delle colonne di A  
e Indice di riga di B

## II Strategia: Esempio n=6

$$\begin{matrix} A_1 \\ \left[ \begin{matrix} a_{03} & a_{04} & a_{05} \\ a_{13} & a_{14} & a_{15} \\ a_{23} & a_{24} & a_{25} \\ a_{33} & a_{34} & a_{35} \\ a_{43} & a_{44} & a_{45} \\ a_{53} & a_{54} & a_{55} \end{matrix} \right] \cdot \left[ \begin{matrix} b_{30} & b_{31} & b_{32} & b_{33} & b_{34} & b_{35} \\ b_{40} & b_{41} & b_{42} & b_{43} & b_{44} & b_{45} \\ b_{50} & b_{51} & b_{52} & b_{53} & b_{54} & b_{55} \end{matrix} \right] = [?] \end{matrix}$$

*quali componenti  
della matrice C calcola lo  
studente 1 ?*



Studente 2

## II Strategia: Esempio n=6

$$\begin{matrix} A_1 \\ \left[ \begin{matrix} a_{03} & a_{04} & a_{05} \\ a_{13} & a_{14} & a_{15} \\ a_{23} & a_{24} & a_{25} \\ a_{33} & a_{34} & a_{35} \\ a_{43} & a_{44} & a_{45} \\ a_{53} & a_{54} & a_{55} \end{matrix} \right] \bullet \left[ \begin{matrix} b_{30} & b_{31} & b_{32} & b_{33} & b_{34} & b_{35} \\ b_{40} & b_{41} & b_{42} & b_{43} & b_{44} & b_{45} \\ b_{50} & b_{51} & b_{52} & b_{53} & b_{54} & b_{55} \end{matrix} \right] = \left[ \begin{matrix} c^2_{00} & c^2_{01} & c^2_{02} & c^2_{03} & c^2_{04} & c^2_{05} \\ c^2_{10} & c^2_{11} & c^2_{12} & c^2_{13} & c^2_{14} & c^2_{15} \\ c^2_{20} & c^2_{21} & c^2_{22} & c^2_{23} & c^2_{24} & c^2_{25} \\ c^2_{30} & c^2_{31} & c^2_{32} & c^2_{33} & c^2_{34} & c^2_{35} \\ c^2_{40} & c^2_{41} & c^2_{42} & c^2_{43} & c^2_{44} & c^2_{45} \\ c^2_{50} & c^2_{51} & c^2_{52} & c^2_{53} & c^2_{54} & c^2_{55} \end{matrix} \right] \end{matrix}$$

Studente 2

Lo studente 2 può calcolare un contributo per OGNI elemento della matrice C

## II Strategia: Esempio n=6

$$\begin{matrix} A_1 \\ \left[ \begin{matrix} a_{03} & a_{04} & a_{05} \\ a_{13} & a_{14} & a_{15} \\ a_{23} & a_{24} & a_{25} \\ a_{33} & a_{34} & a_{35} \\ a_{43} & a_{44} & a_{45} \\ a_{53} & a_{54} & a_{55} \end{matrix} \right] \bullet \left[ \begin{matrix} b_{30} & b_{31} & b_{32} & b_{33} & b_{34} & b_{35} \\ b_{40} & b_{41} & b_{42} & b_{43} & b_{44} & b_{45} \\ b_{50} & b_{51} & b_{52} & b_{53} & b_{54} & b_{55} \end{matrix} \right] = C^2 \end{matrix}$$

 Lo studente 2 può calcolare un contributo per OGNI elemento della matrice C

Studente 2

Cioè...

---

i,j=0,5

$$c_{ij}^2 = \sum_{k=3}^5 a_{ik} b_{kj}$$

Lo studente 2 può calcolare

un contributo per OGNI  
elemento della matrice C

k = Indice delle colonne di A  
e Indice di riga di B



Studente 2

# Pertanto...

---

$$c_{ij} = \sum_{k=0}^5 a_{ik} b_{kj}$$

i,j=0,5

I due contributi  
devono essere  
sommati

$$\sum_{k=0}^2 a_{ik} b_{kj}$$



Studente 1

$$\sum_{k=3}^5 a_{ik} b_{kj}$$



Studente 2

# Ovvero, in forma matriciale

$$\begin{bmatrix} C_{00} & C_{01} & C_{02} & C_{03} & C_{04} & C_{05} \\ C_{10} & C_{11} & C_{12} & C_{13} & C_{14} & C_{15} \\ C_{20} & C_{21} & C_{22} & C_{23} & C_{24} & C_{25} \\ C_{30} & C_{31} & C_{32} & C_{33} & C_{34} & C_{35} \\ C_{40} & C_{41} & C_{42} & C_{43} & C_{44} & C_{45} \\ C_{50} & C_{51} & C_{52} & C_{53} & C_{54} & C_{55} \end{bmatrix} = \begin{bmatrix} C_{00}^1 & C_{01}^1 & C_{02}^1 & C_{03}^1 & C_{04}^1 & C_{05}^1 \\ C_{10}^1 & C_{11}^1 & C_{12}^1 & C_{13}^1 & C_{14}^1 & C_{15}^1 \\ C_{20}^1 & C_{21}^1 & C_{22}^1 & C_{23}^1 & C_{24}^1 & C_{25}^1 \\ C_{30}^1 & C_{31}^1 & C_{32}^1 & C_{33}^1 & C_{34}^1 & C_{35}^1 \\ C_{40}^1 & C_{41}^1 & C_{42}^1 & C_{43}^1 & C_{44}^1 & C_{45}^1 \\ C_{50}^1 & C_{51}^1 & C_{52}^1 & C_{53}^1 & C_{54}^1 & C_{55}^1 \end{bmatrix} + \begin{bmatrix} C_{00}^2 & C_{01}^2 & C_{02}^2 & C_{03}^2 & C_{04}^2 & C_{05}^2 \\ C_{10}^2 & C_{11}^2 & C_{12}^2 & C_{13}^2 & C_{14}^2 & C_{15}^2 \\ C_{20}^2 & C_{21}^2 & C_{22}^2 & C_{23}^2 & C_{24}^2 & C_{25}^2 \\ C_{30}^2 & C_{31}^2 & C_{32}^2 & C_{33}^2 & C_{34}^2 & C_{35}^2 \\ C_{40}^2 & C_{41}^2 & C_{42}^2 & C_{43}^2 & C_{44}^2 & C_{45}^2 \\ C_{50}^2 & C_{51}^2 & C_{52}^2 & C_{53}^2 & C_{54}^2 & C_{55}^2 \end{bmatrix}$$

I due contributi  
devono essere  
sommati



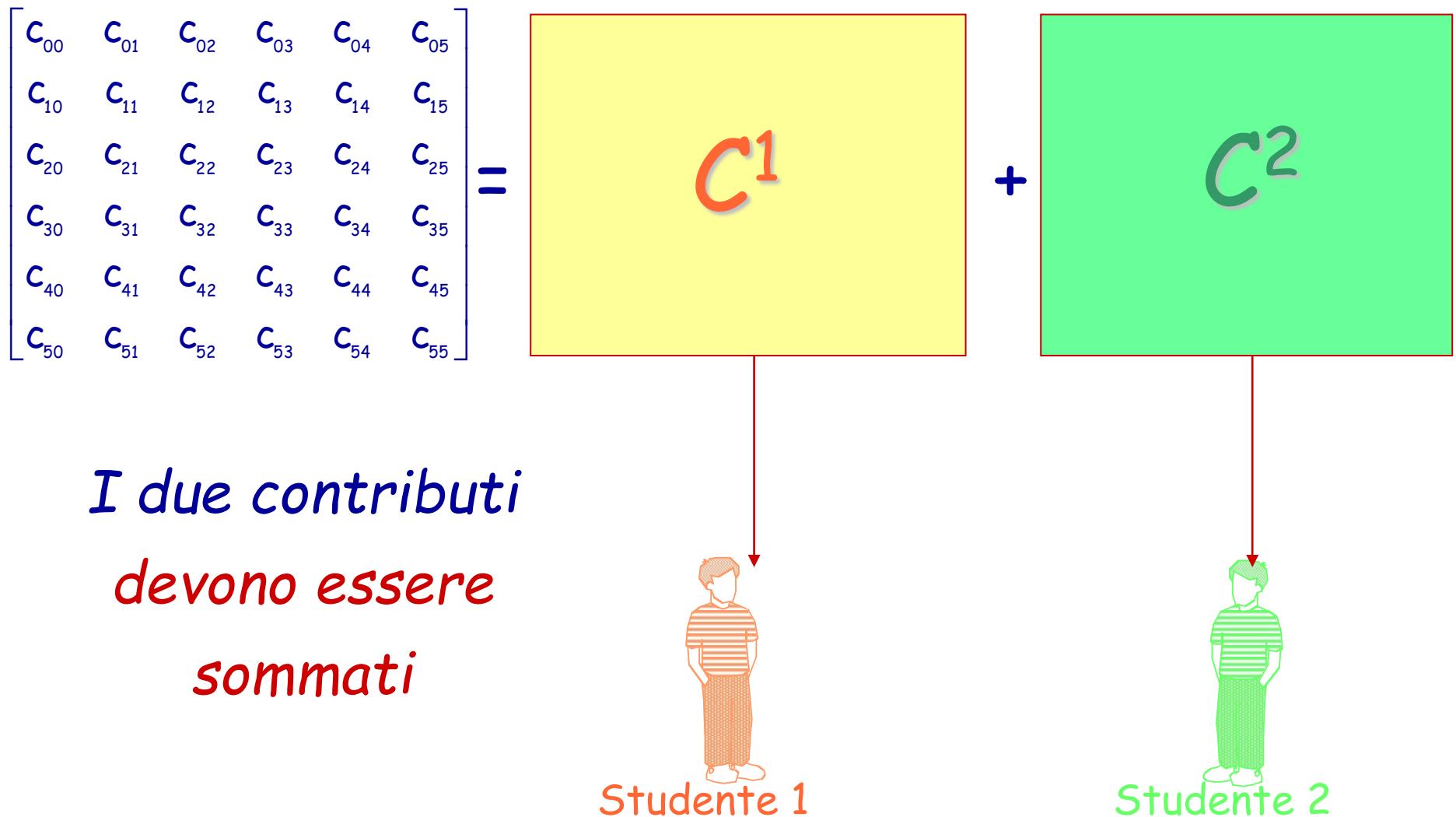
Studente 1

+



Studente 2

# Ovvero, in forma matriciale



# Domanda

---

Come calcolare  
la matrice  $C$

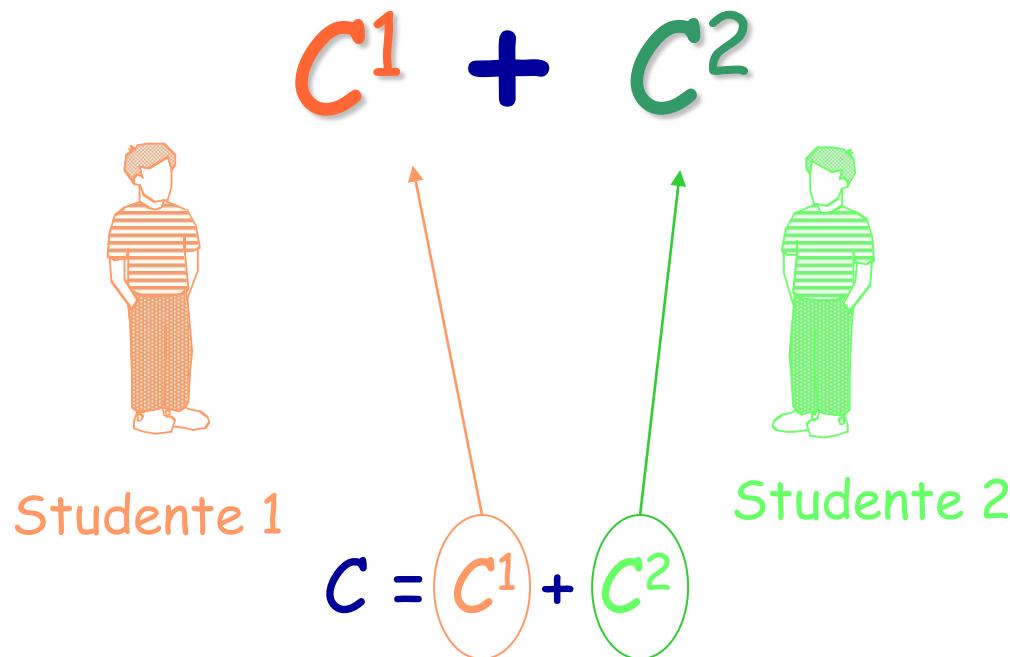
$$C = C^1 + C^2$$

?

## II STRATEGIA: Esempio n= 6

---

Per ottenere la matrice C  
gli studenti devono “interagire”  
sommendo i loro risultati parziali



## II STRATEGIA: In generale

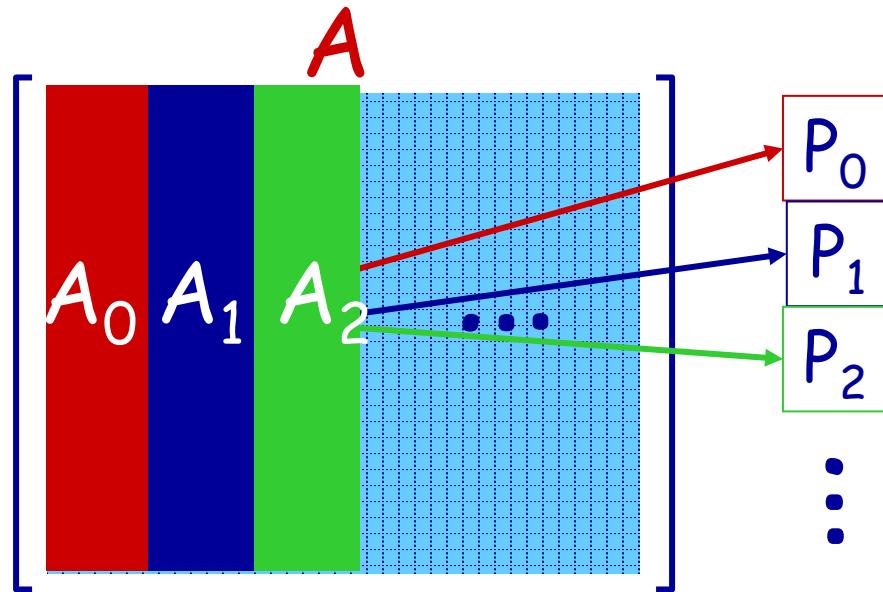
---

- ◆ **Calcolo:**  
Inizialmente tutti i processori calcolano, indipendentemente, un contributo parziale di tutte le componenti di  $C$
- ◆ **Comunicazione/calcolo**  
Successivamente in parallelo tutti i processori concorrono alla somma dei contributi parziali

## II STRATEGIA: In generale

### I passo: decomposizione del problema

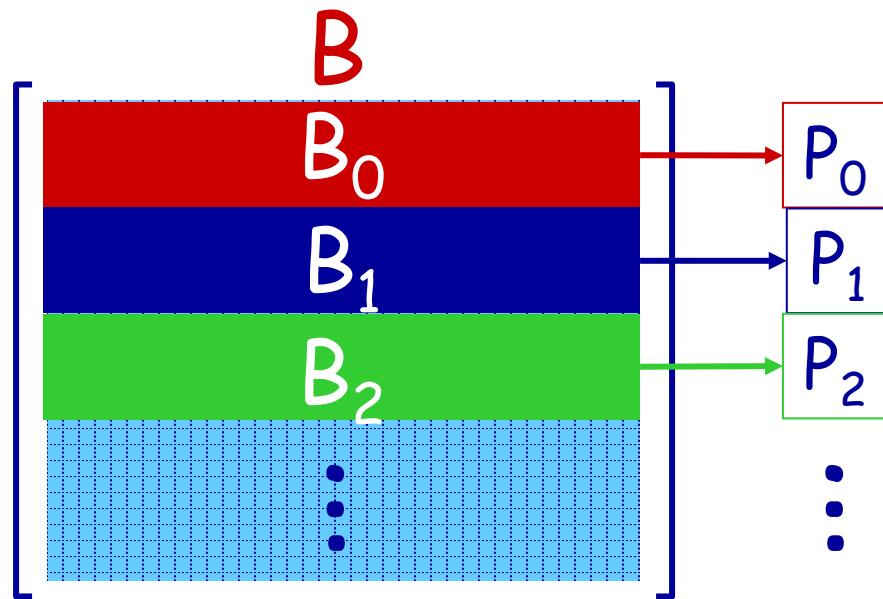
La matrice  $A$  viene distribuita  
in BLOCCHI di COLONNE  
fra p processori



## II STRATEGIA: In generale

### I passo: decomposizione del problema

La matrice  $B$  viene distribuita  
in BLOCCHI di RIGHE  
fra  $p$  processori



## II STRATEGIA: In generale

---

### II passo: risoluzione dei sottoproblemi

Il prodotto  $A \cdot B = C$  viene decomposto  
in  $p$  prodotti del tipo

$$A_i \cdot B_i = C^i \text{ dove } C = \sum_{i=0}^{p-1} C^i$$

Ciascun processore calcola  
un prodotto matrice matrice  
(di dimensione più piccola di quello assegnato).

## II strategia: caratteristiche

---

- ◆ I risultati parziali devono essere sommati tra i processori
- ◆ In questo caso l'algoritmo parallelo è analogo a quello del prodotto matrice-vettore (II strategia)

# Domanda

---

Qual è  
l'algoritmo parallelo  
della II Strategia  
di decomposizione

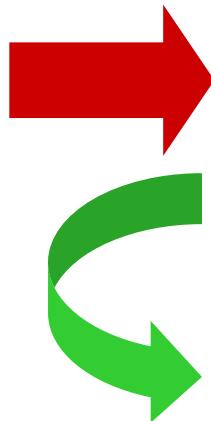
?

# Risposta

Partizionamento di  
A in blocchi di colonne  
B in blocchi di righe



```
begin
  C = 0
  for i=0 to p-1 do
    Ci = Ai · Bi
    C = C + Ci
  endfor
End
```



Distribuzione dei  
blocchi fra i  
processori

Algoritmo parallelo

Parallelizzazione dell'algoritmo a  
blocchi!

# Risposta

Partizionamento di  
A in blocchi di colonne  
B in blocchi di righe



Algoritmo parallelo

```
begin
  C =0
  forall i=0 to p-1 do
    Ci = Ai · Bi
    C =C + Ci
  endfor
End
```



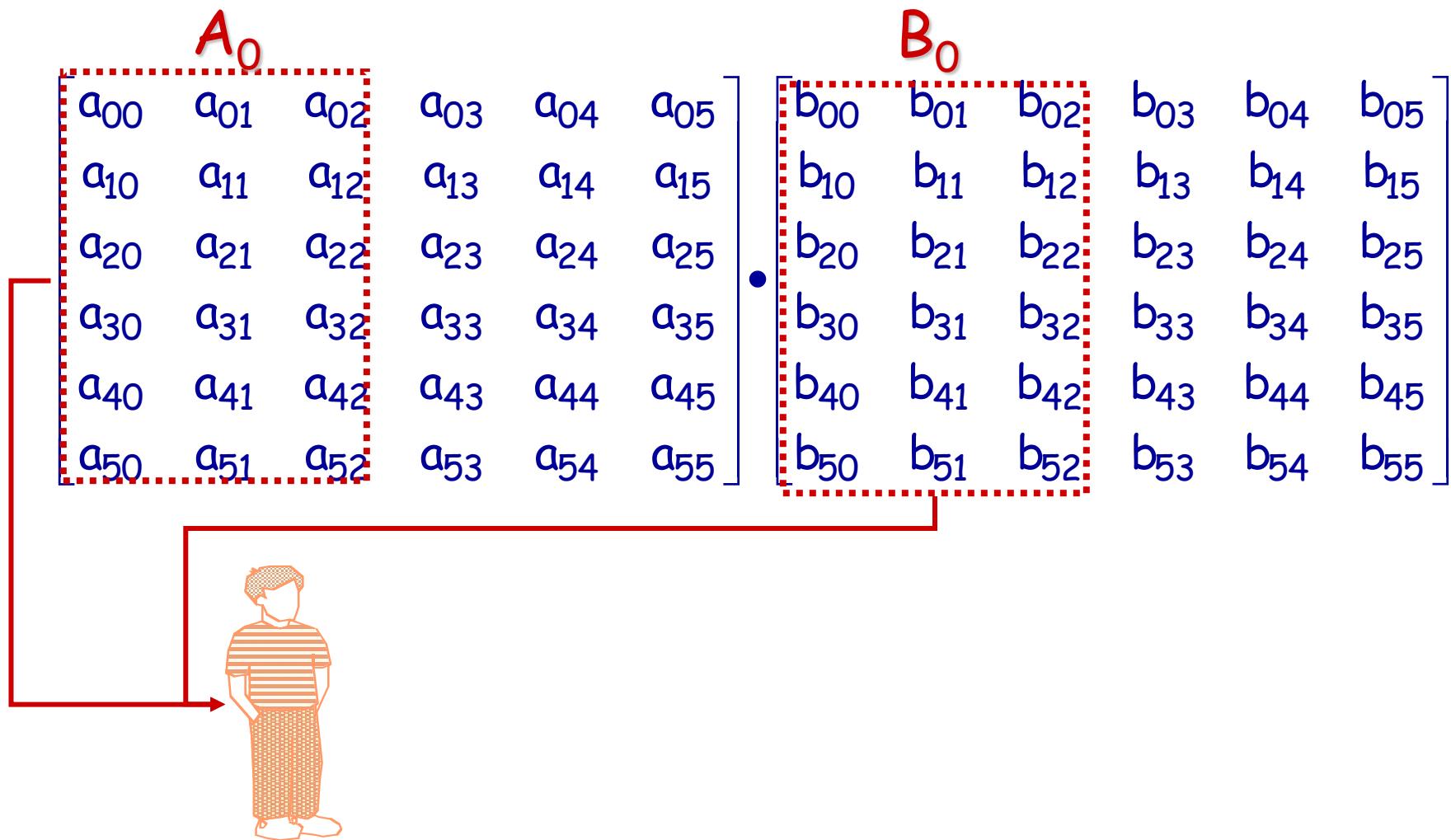
```
begin
  forall Pi (i=0, . . . , p-1)
    { Pi calcola Ci = Ai · Bi }
    { combinazione dei Ci }
    C =C + Ci
  end
```

## III strategia

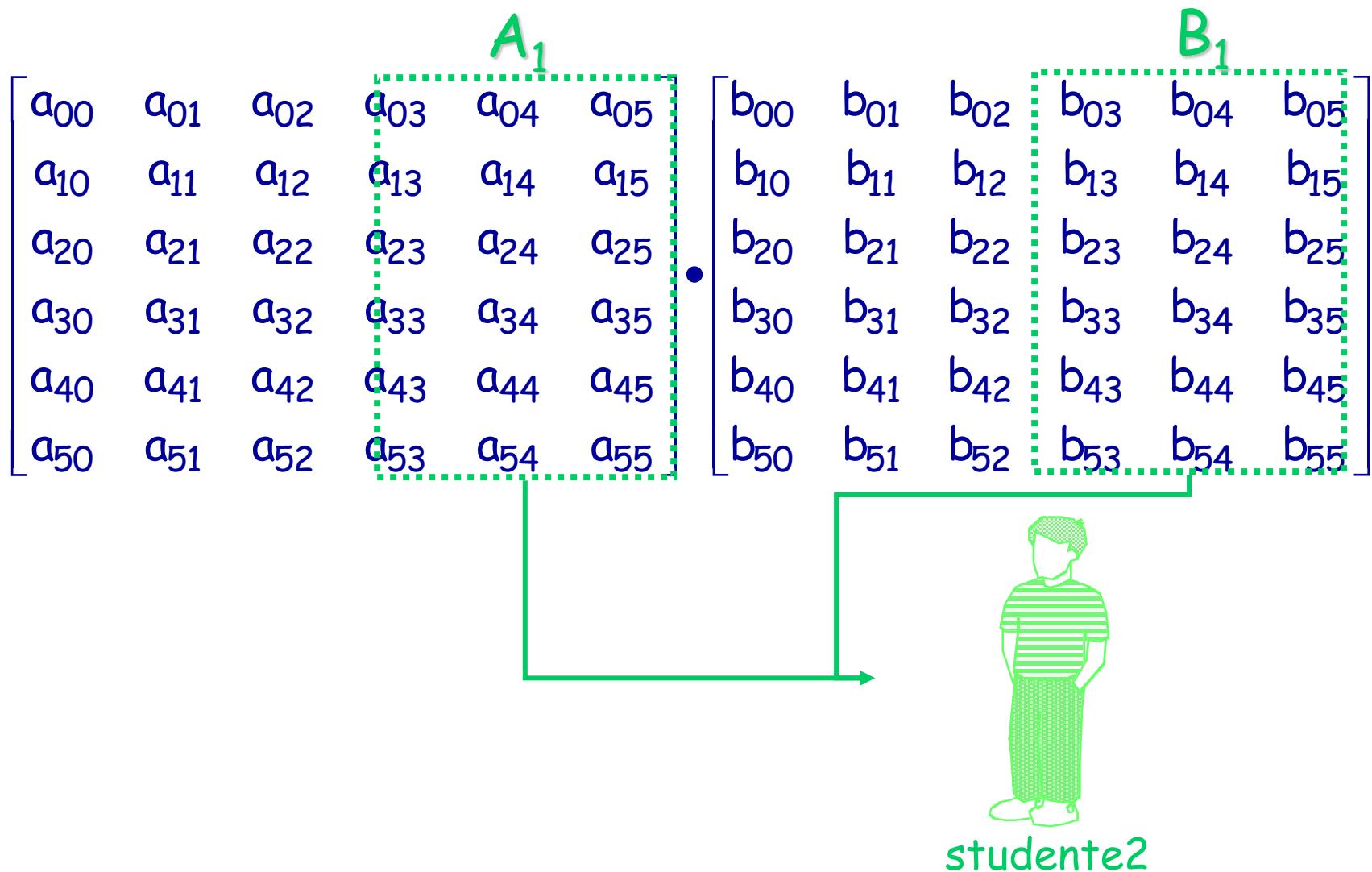
---

Suddividiamo  
ENTRAMBE  
le matrici A e B in  
blocchi di COLONNE

# Distribuzione dei dati: Esempio n=6



# Distribuzione dei dati: Esempio n=6



# Domanda

---

Con i dati così distribuiti  
cosa può calcolare  
ciascuno studente

?

# III Strategia: Esempio n=6

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \\ a_{30} & a_{31} & a_{32} \\ a_{40} & a_{41} & a_{42} \\ a_{50} & a_{51} & a_{50} \end{bmatrix} \cdot \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \\ b_{30} & b_{31} & b_{32} \\ b_{40} & b_{41} & b_{42} \\ b_{50} & b_{51} & b_{52} \end{bmatrix} = [?]$$

*quali componenti*

*della matrice C*

*calcola lo studente 1*

?



Studente 1

# III Strategia: Esempio n=6

$$\begin{bmatrix} a_{03} & a_{04} & a_{05} \\ a_{13} & a_{14} & a_{15} \\ a_{23} & a_{24} & a_{25} \\ a_{33} & a_{34} & b_{35} \\ a_{43} & a_{44} & a_{45} \\ a_{53} & a_{54} & a_{55} \end{bmatrix} \cdot \begin{bmatrix} b_{03} & b_{04} & b_{05} \\ b_{13} & b_{14} & b_{15} \\ b_{23} & b_{24} & b_{25} \\ b_{33} & b_{34} & b_{35} \\ b_{43} & b_{44} & b_{45} \\ b_{53} & b_{54} & b_{55} \end{bmatrix} = [?]$$

*quali componenti  
della matrice C  
calcola lo studente 2*

Studente 2 ?

# Premessa...

$$\begin{matrix} C \\ \begin{bmatrix} C_{00} & C_{01} & C_{02} & C_{03} & C_{04} & C_{05} \\ C_{10} & C_{11} & C_{12} & C_{13} & C_{14} & C_{15} \\ C_{20} & C_{21} & C_{22} & C_{23} & C_{24} & C_{25} \\ C_{30} & C_{31} & C_{32} & C_{33} & C_{34} & C_{35} \\ C_{40} & C_{41} & C_{42} & C_{43} & C_{44} & C_{45} \\ C_{50} & C_{51} & C_{52} & C_{53} & C_{54} & C_{55} \end{bmatrix} = \begin{matrix} A \\ \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix} \bullet \begin{matrix} B \\ \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} & b_{04} & b_{05} \\ b_{10} & b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ b_{20} & b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \\ b_{30} & b_{31} & b_{32} & b_{33} & b_{34} & b_{35} \\ b_{40} & b_{41} & b_{42} & b_{43} & b_{44} & b_{45} \\ b_{50} & b_{51} & b_{52} & b_{53} & b_{54} & b_{55} \end{bmatrix} \end{matrix} \end{matrix}$$

$A_0$        $A_1$        $B_0$        $B_1$

Riorganizziamo la matrice  $C$  in blocchi quadrati

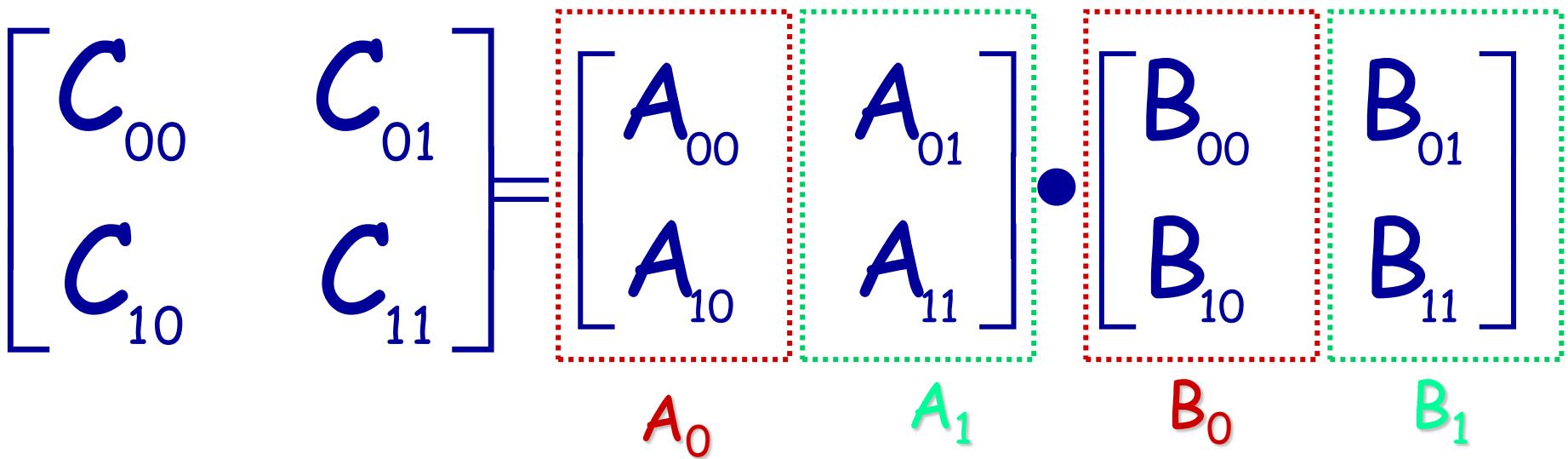
# Premessa...

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix} \bullet \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} & b_{04} & b_{05} \\ b_{10} & b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ b_{20} & b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \\ b_{30} & b_{31} & b_{32} & b_{33} & b_{34} & b_{35} \\ b_{40} & b_{41} & b_{42} & b_{43} & b_{44} & b_{45} \\ b_{50} & b_{51} & b_{52} & b_{53} & b_{54} & b_{55} \end{bmatrix}$$

$A_0$        $A_1$        $B_0$        $B_1$

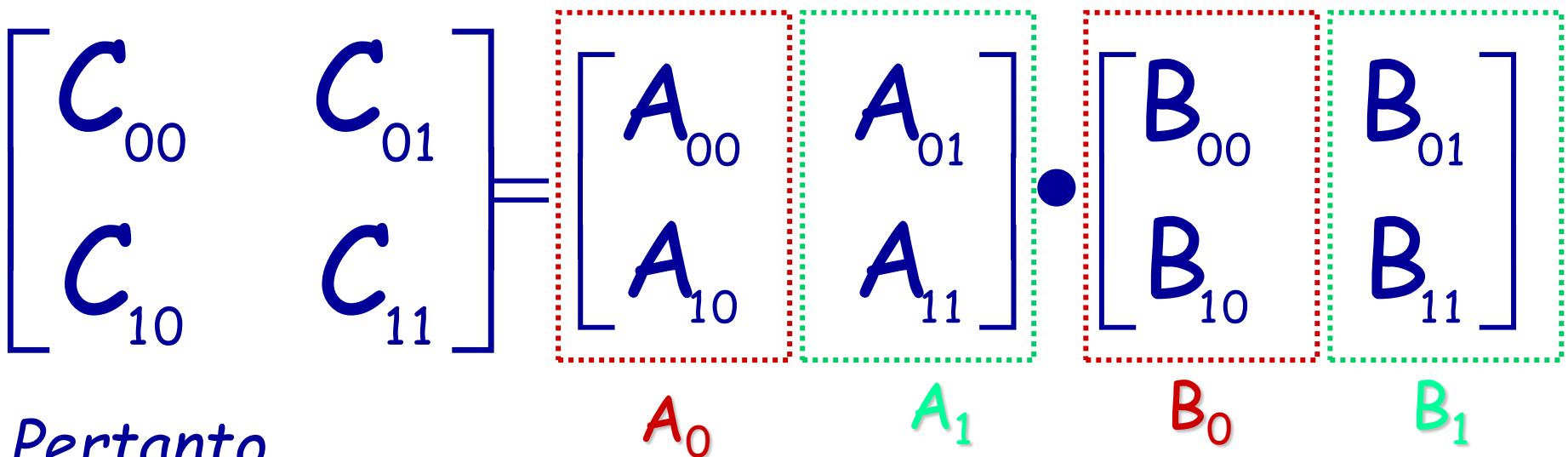
Riorganizziamo la matrice  $C$  in blocchi quadrati  
Allo stesso modo riorganizziamo  
i blocchi di colonne di  $A$  e  $B$ !

# Osservazione...



Riorganizziamo la matrice  $C$  in blocchi quadrati  
Allo stesso modo riorganizziamo  
i blocchi di colonne di  $A$  e  $B$ !

# Osservazione...



Pertanto...

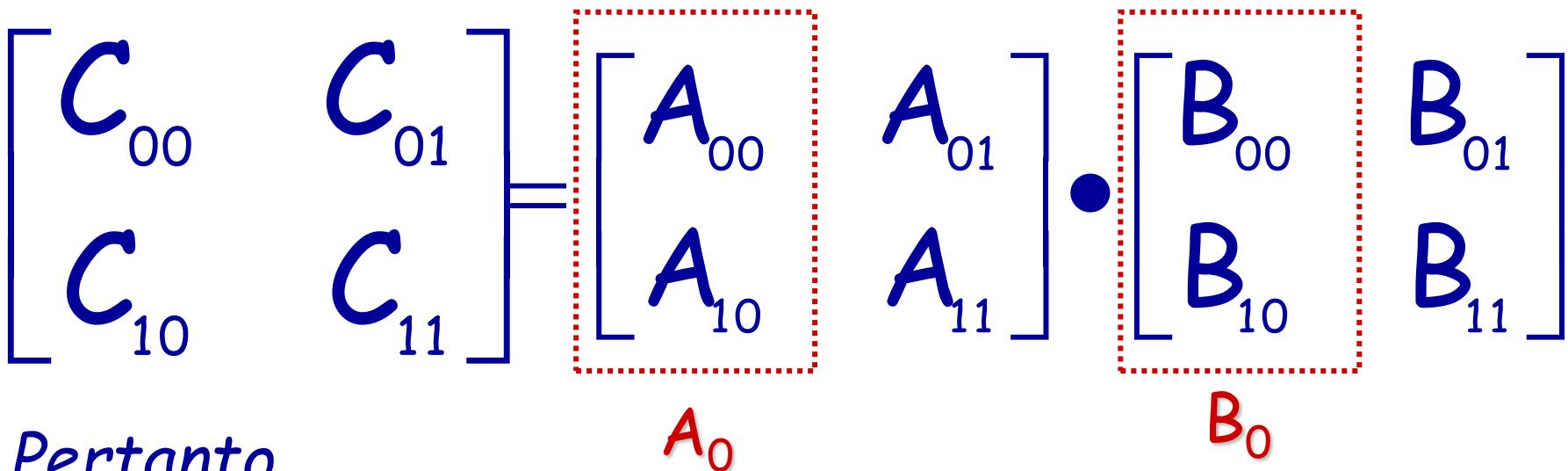
$$C_{00} = A_{00}B_{00} + A_{01}B_{10}$$

$$C_{10} = A_{10}B_{00} + A_{11}B_{10}$$

$$C_{01} = A_{00}B_{01} + A_{01}B_{11}$$

$$C_{11} = A_{10}B_{01} + A_{11}B_{11}$$

# Fase di calcolo



Pertanto...

$$C_{00} = A_{00} B_{00} + A_{01} B_{10}$$

$$C_{10} = A_{10} B_{00} + A_{11} B_{10}$$

$$C_{01} = A_{00} B_{01} + A_{01} B_{11}$$

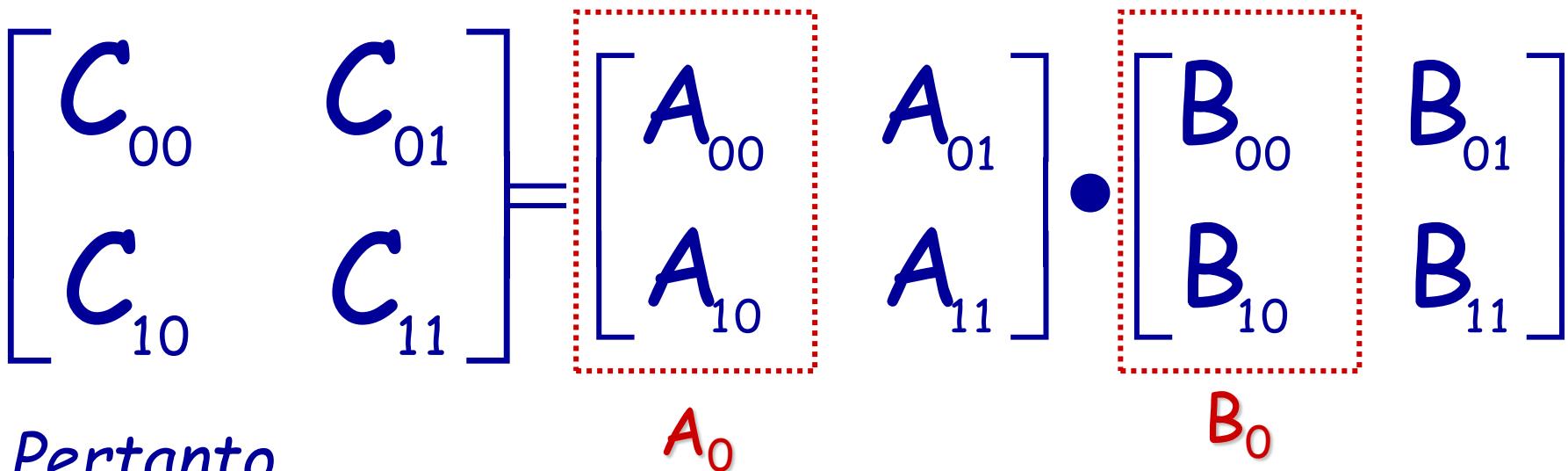
$$C_{11} = A_{10} B_{01} + A_{11} B_{11}$$



Studente 1

Lo studente 1  
calcola "un contributo"  
di "una parte" della  
matrice C!

# Fase di calcolo



$$C_{00} = \boxed{A_{00} B_{00}} + \boxed{A_{01} B_{10}}$$
$$C_{10} = \boxed{A_{10} B_{00}} + \boxed{A_{11} B_{10}}$$

?

$$C_{01} = A_{00} B_{01} + A_{01} B_{11}$$

$$C_{11} = A_{10} B_{01} + A_{11} B_{11}$$



Lo studente 1  
NON può completare  
i calcoli  
perché non possiede  
gli altri blocchi di A!

# Fase di calcolo

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{10} \\ A_{01} & A_{11} \end{bmatrix} \cdot \begin{bmatrix} B_{00} & B_{10} \\ B_{01} & B_{11} \end{bmatrix}$$

$A_1$        $B_1$

Pertanto...

$$C_{00} = A_{00} B_{00} + A_{01} B_{10}$$

$$C_{10} = A_{10} B_{00} + A_{11} B_{10}$$

$$C_{01} = A_{00} B_{01} + A_{01} B_{11}$$

$$C_{11} = A_{10} B_{01} + A_{11} B_{11}$$



Lo studente 2  
calcola "un contributo"  
di "una parte" della  
matrice C!

# Fase di calcolo

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{10} \\ A_{01} & A_{11} \end{bmatrix}$$

$$\begin{bmatrix} A_{01} \\ A_{11} \end{bmatrix} \bullet \begin{bmatrix} B_{00} & B_{10} \\ B_{10} & B_{11} \end{bmatrix}$$

$A_1$        $B_1$

Pertanto...

$$C_{00} = A_{00} B_{00} + A_{01} B_{10}$$

$$C_{10} = A_{10} B_{00} + A_{11} B_{10}$$

$$C_{01} = \boxed{A_{00} B_{01}} + \boxed{A_{01} B_{11}}$$
$$C_{11} = \boxed{A_{10} B_{01}} + \boxed{A_{11} B_{11}}$$



Student

Lo studente 2  
NON può completare  
i calcoli  
perché non possiede  
gli altri blocchi di A!

# Domanda

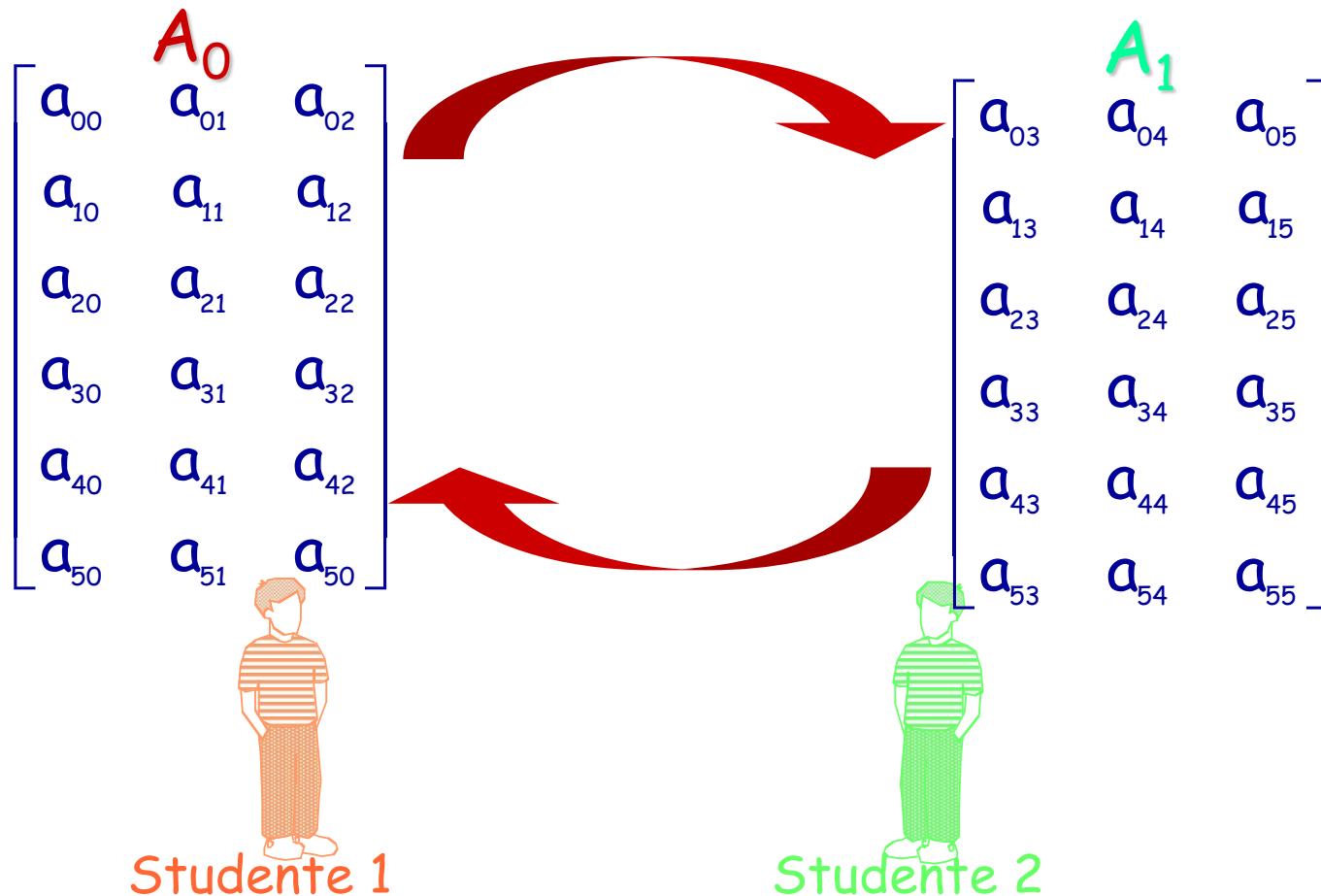
---

Come "completare"  
i contributi calcolati

?

# IDEA!

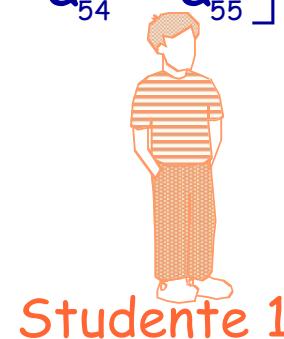
Gli studenti possono scambiarsi  
i blocchi di COLONNE della matrice A!



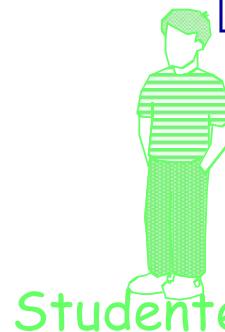
# IDEA!

Gli studenti possono scambiarsi  
i blocchi di COLONNE della matrice A!

$$\begin{bmatrix} a_{03} & \color{red}{a_0} & a_{05} \\ a_{13} & a_{14} & a_{15} \\ a_{23} & a_{24} & a_{25} \\ a_{33} & a_{34} & a_{35} \\ a_{43} & a_{44} & a_{45} \\ a_{53} & a_{54} & a_{55} \end{bmatrix}$$

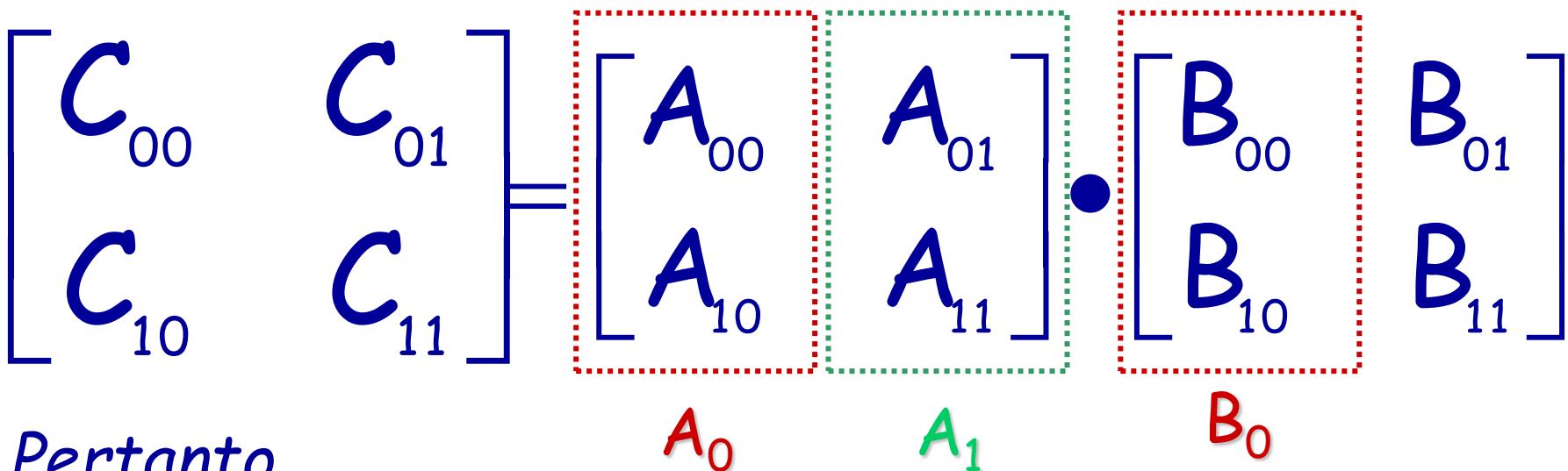


$$\begin{bmatrix} a_{00} & \color{green}{a_1} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \\ a_{30} & a_{31} & a_{32} \\ a_{40} & a_{41} & a_{42} \\ a_{50} & a_{51} & a_{50} \end{bmatrix}$$



Dopo lo scambio...

# Dopo lo scambio... Fase di calcolo 2



Pertanto...

$$C_{00} = \boxed{A_{00} B_{00}} + \boxed{A_{01} B_{10}}$$

$$C_{10} = \boxed{A_{10} B_{00}} + \boxed{A_{11} B_{10}}$$

$$C_{01} = A_{00} B_{01} + A_{01} B_{11}$$

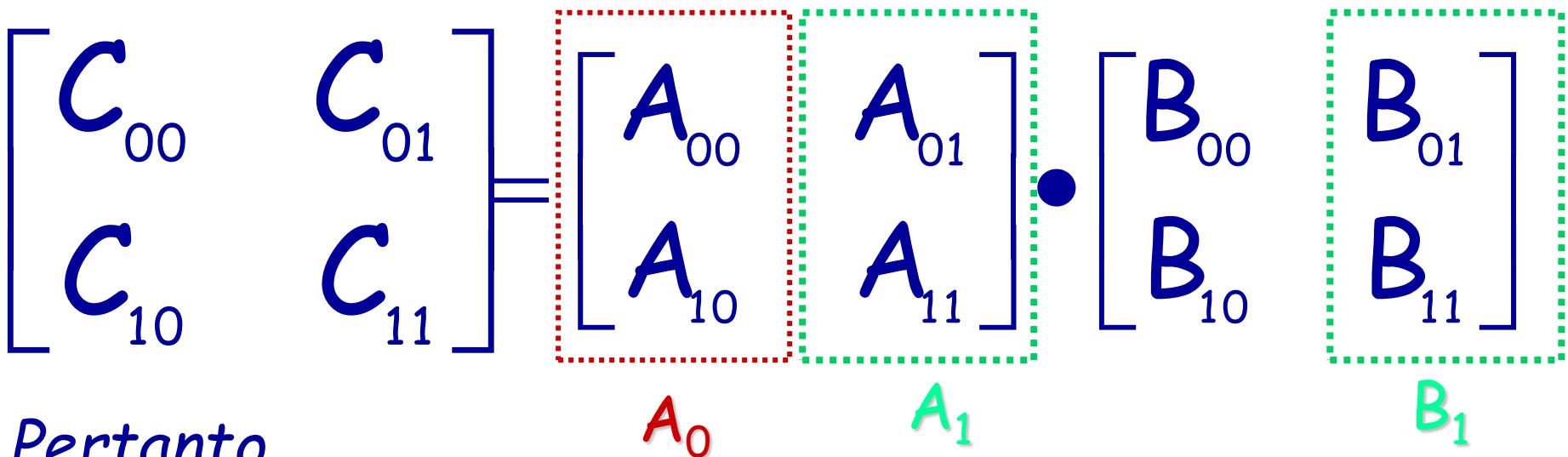
$$C_{11} = A_{10} B_{01} + A_{11} B_{11}$$



Studente 1

Lo studente 1  
ha ricevuto  
l'altro blocco di  $A$   
e può completare il  
Calcolo!

# Dopo lo scambio... Fase di calcolo 2



$$C_{00} = A_{00} B_{00} + A_{01} B_{10}$$

$$C_{10} = A_{10} B_{00} + A_{11} B_{10}$$

$$C_{01} = A_{00} B_{01} + A_{01} B_{11}$$

$$C_{11} = A_{10} B_{01} + A_{11} B_{11}$$

Lo studente 2  
ha ricevuto  
l'altro blocco di A  
e può completare il  
Calcolo!

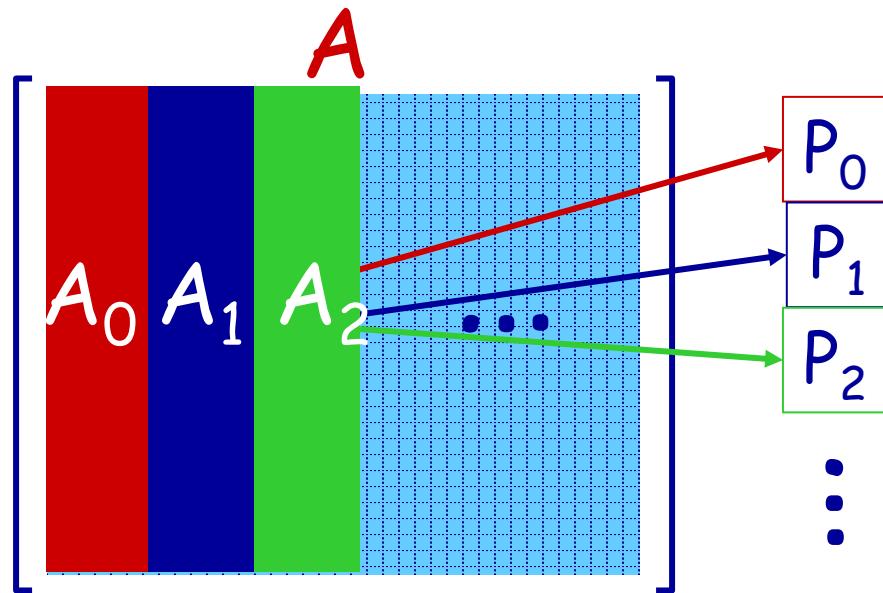


Studente 2

# III STRATEGIA: In generale

## I passo: decomposizione del problema

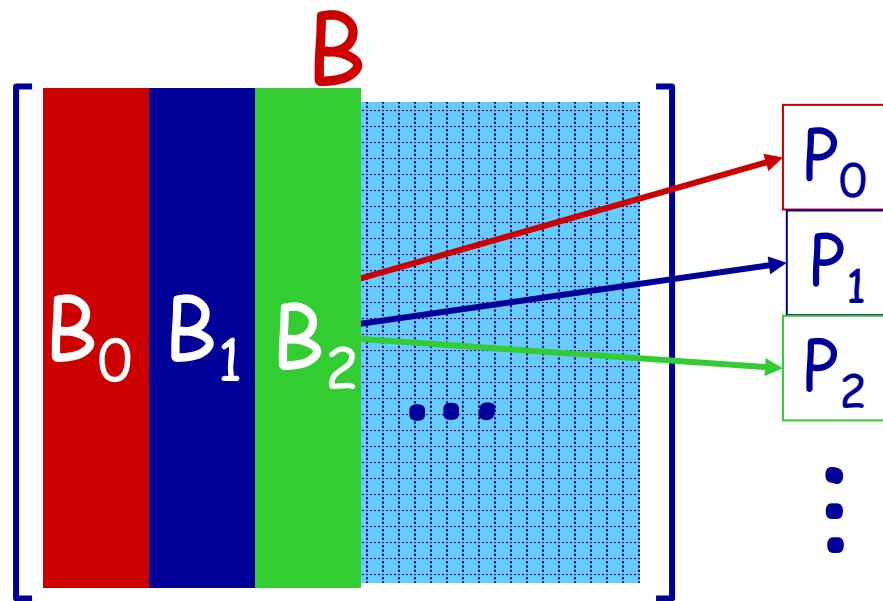
La matrice  $A$  viene distribuita  
in BLOCCHI di COLONNE  
fra p processori



# III STRATEGIA: In generale

## I passo: decomposizione del problema

La matrice  $B$  viene distribuita  
in BLOCCHI di COLONNE  
fra p processori



### III STRATEGIA: In generale

---

#### II passo: risoluzione dei sottoproblemi

Il prodotto  $A \cdot B = C$  viene decomposto  
in  $p \times p$  prodotti del tipo

$$C_i = \sum_{k=0}^{p-1} A_k B_{ki}$$

Ciascun processore calcola  
 $p$  prodotti matrice matrice  
(di dimensione più piccola di quello assegnato).

# Domanda

---

Qual è l'algoritmo parallelo  
della III Strategia  
di decomposizione

?

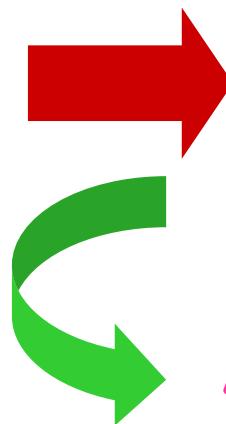
# Risposta

Partizionamento delle matrici

A e B in blocchi di colonne



```
begin
  for i=0 to p-1 do
    Ci = 0
    for j=0 to p-1 do
      Ci = Ci + Aj · Bji
    endfor
  endfor
end
```



Distribuzione dei  
blocchi fra i  
processori

Algoritmo parallelo

Parallelizzazione dell'algoritmo a  
blocchi!

# Risposta

Partizionamento delle matrici

A e B in blocchi di colonne



```
begin
  for i=0 to p-1 do
    Ci = 0
    for j=0 to p-1 do
      Ci = Ci + Aj · Bji
    endfor
  endfor
end
```

Algoritmo parallelo

```
Begin
  for k=0 to p-1 do
    forall Pi (i=0, ..., p-1)
    j=mod(i+k, p);
    {Pi calcola
      Ci = Ci + Aj · Bji}
    send(Aj, Pi-1) {p = 0}
    recv(Aj+1, Pi+1) {-1=p-1}
  endfor
```

---

# Fine Lezione

# Parallel and Distributed Computing

Prodotto Matrice-Vettore

---

*Approfondimenti...*

# PROBLEMA: Prodotto Matrice-Vettore

---

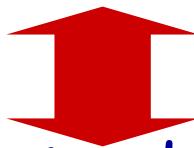
Progettazione  
di un algoritmo parallelo  
per architettura MIMD  
a memoria distribuita  
per il calcolo del prodotto  
di una matrice  $A$  per un vettore  $x$ :

$$Ax = y, \quad A \in \mathbb{R}^{n \times n}, \quad x, y \in \mathbb{R}^n$$

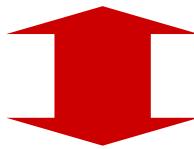
# IDEA!

---

Decomposizione del problema  
Matrice-Vettore



Partizionamento della matrice A  
**IN BLOCCHI**



Riformulazione dell'algoritmo sequenziale  
**"A BLOCCHI"**



Parallelismo dell'algoritmo  
**"A BLOCCHI"**

# I STRATEGIA

---

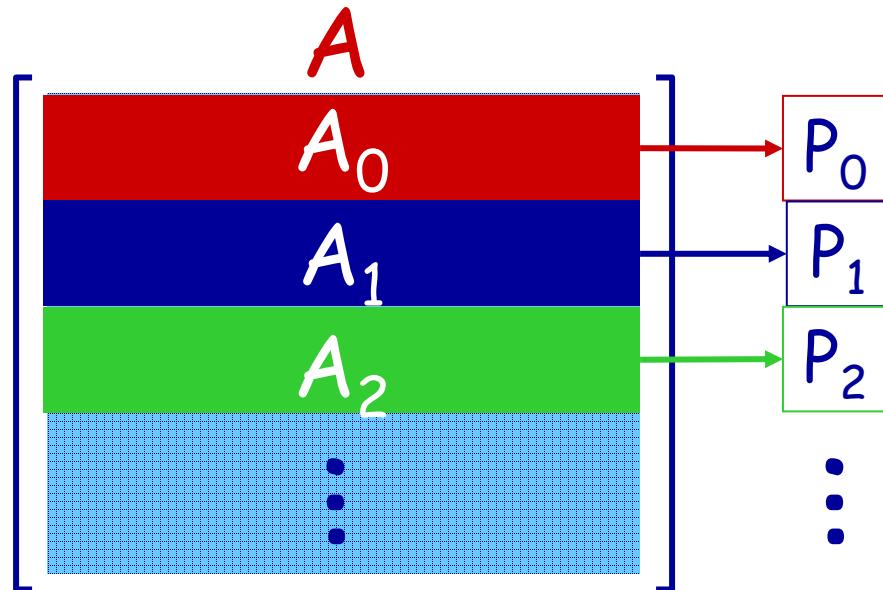
**Decomposizione 1**  
suddividiamo la  
matrice  $A$  in  
**BLOCCHI di RIGHE**

# I STRATEGIA: In generale

---

## I passo: decomposizione del problema

La matrice  $A$  viene distribuita  
in BLOCCHI di RIGHE  
fra p processori

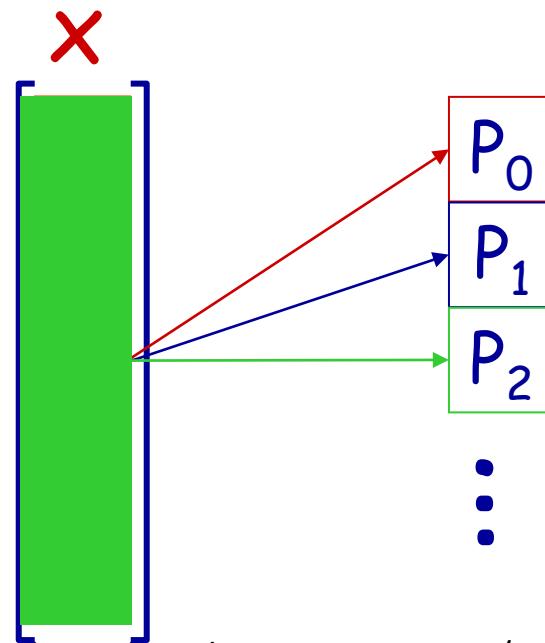


# I STRATEGIA: In generale

---

## I passo: decomposizione del problema

Il vettore  $x$  viene assegnato INTERAMENTE  
ai  $p$  processori



# I STRATEGIA: In generale

---

## II passo: risoluzione dei sottoproblemi

Il prodotto  $Ax=y$  viene decomposto  
in  $p$  prodotti del tipo

$$A_i \cdot x = y_i$$

Ciascun processore calcola  
un prodotto matrice vettore  
(di dimensione più piccola di quello assegnato).

# Domanda

---

Qual è l'algoritmo parallelo  
con la I Strategia  
di decomposizione

?

# Risposta

---

Partizionamento della matrice  
in blocchi di righe



Algoritmo a blocchi

```
begin
    for i=0 to p-1 do
         $y_i = A_i x$ 
    endfor
end
```



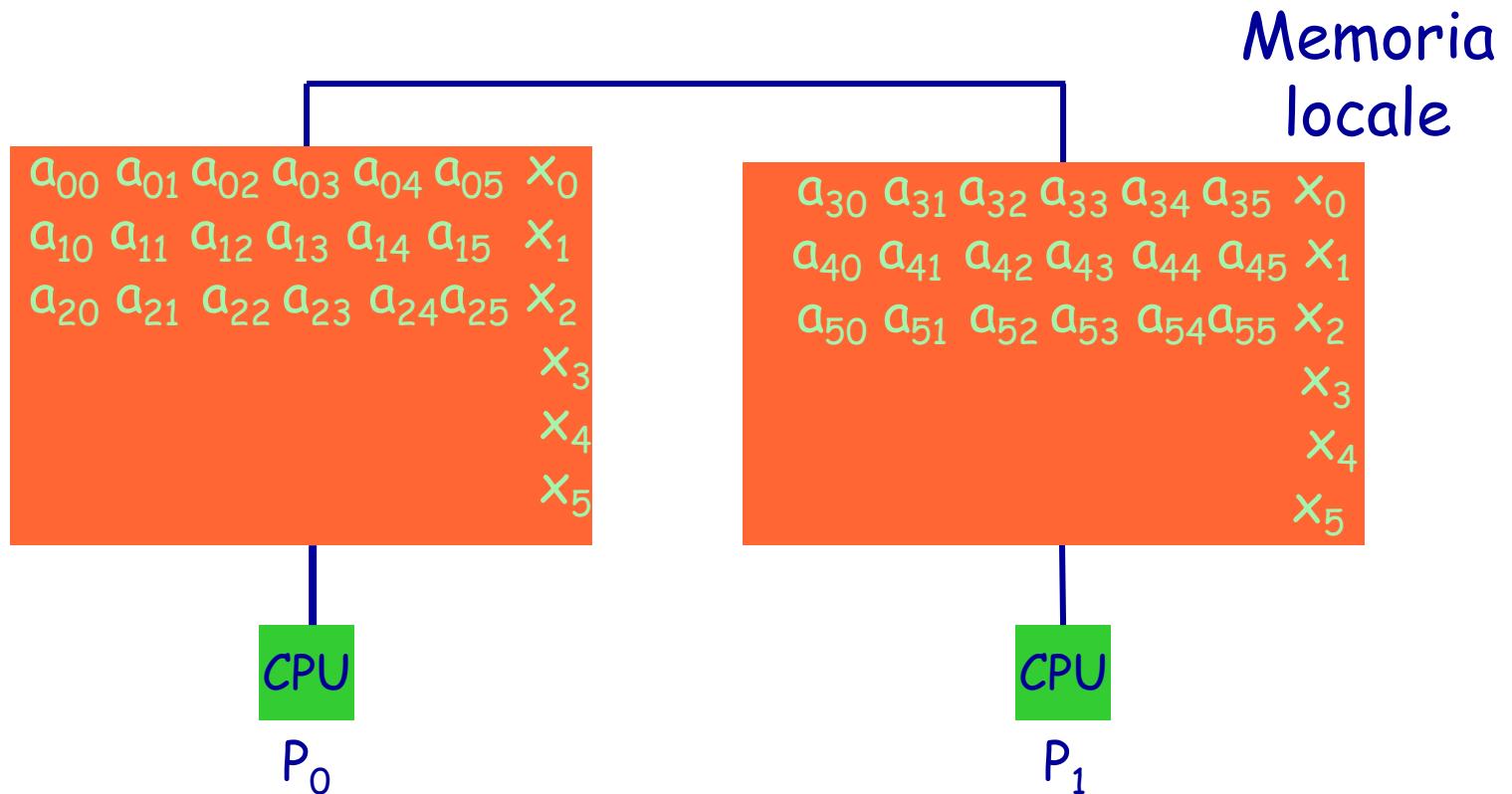
Algoritmo parallelo

```
begin
    forall  $P_i, i=0, p-1$ 
         $\{P_i \text{ calcola } y_i = A_i x\}$ 
    endfor
end
```

Parallelizzazione dell'algoritmo a blocchi!

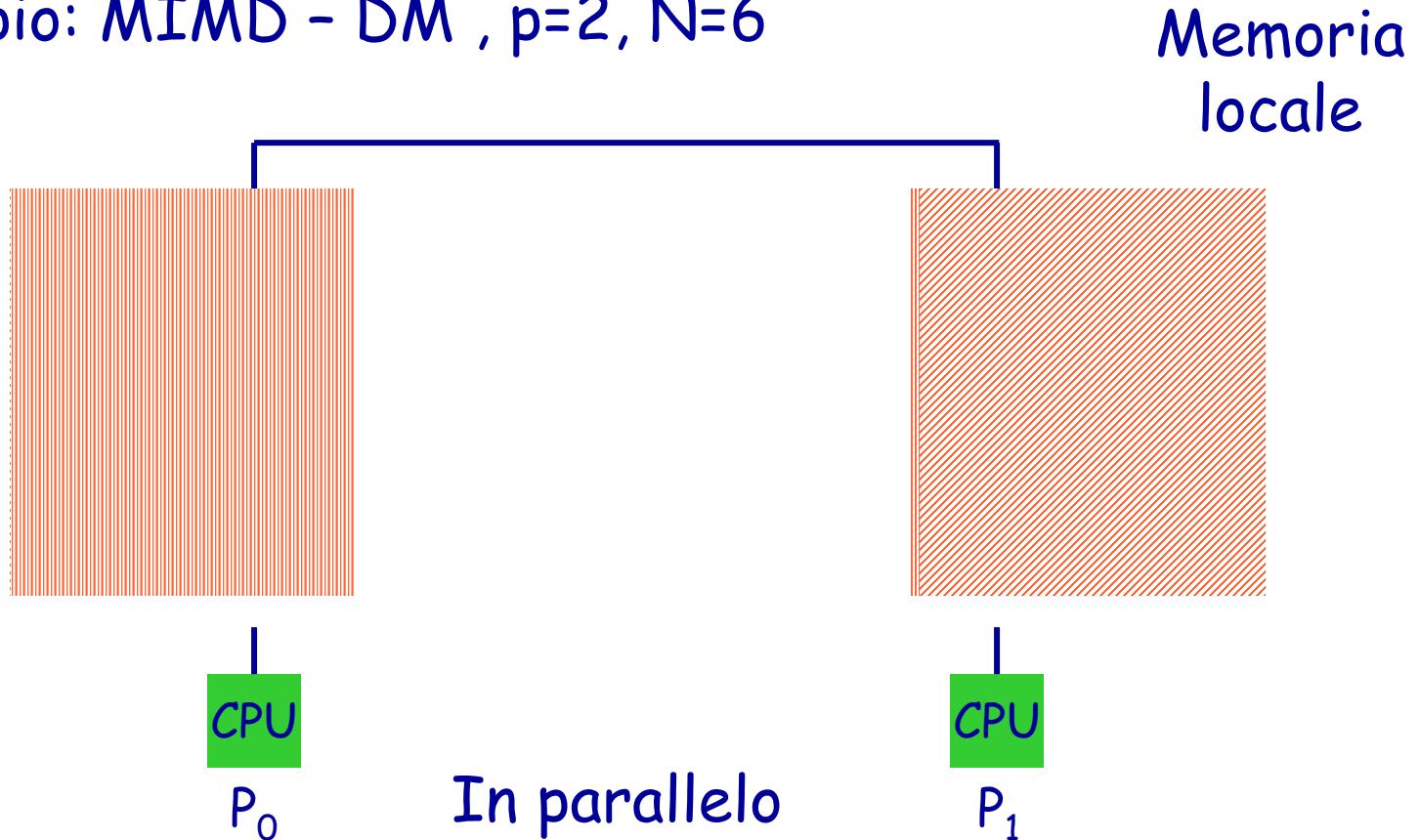
# I Strategia: Distribuzione dei dati

Esempio: MIMD - DM , p=2, N=6



# I Strategia: Fase di calcolo

Esempio: MIMD - DM , p=2, N=6



$$a_{00} x_0 + a_{01} x_1 + a_{02} x_2 + a_{03} x_3 + a_{04} x_4 + a_{05} x_5$$

$$a_{10} x_0 + a_{11} x_1 + a_{12} x_2 + a_{13} x_3 + a_{14} x_4 + a_{15} x_5$$

$$a_{20} x_0 + a_{21} x_1 + a_{22} x_2 + a_{23} x_3 + a_{24} x_4 + a_{25} x_5$$

$$a_{30} x_0 + a_{31} x_1 + a_{32} x_2 + a_{33} x_3 + a_{34} x_4 + a_{35} x_5$$

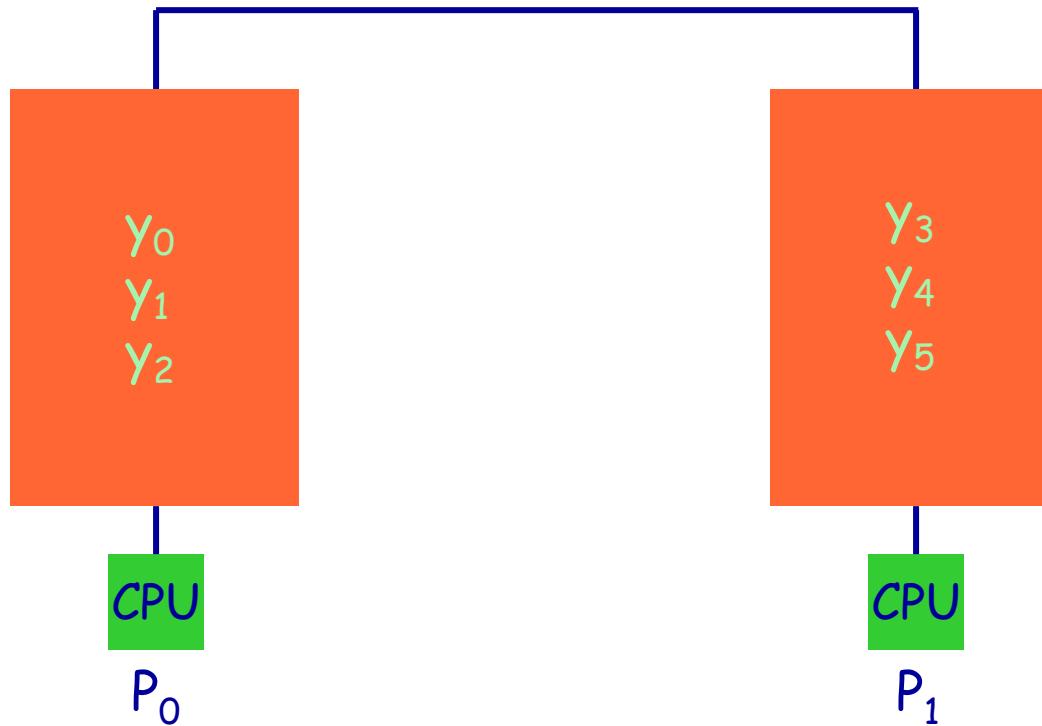
$$a_{40} x_0 + a_{41} x_1 + a_{42} x_2 + a_{43} x_3 + a_{44} x_4 + a_{45} x_5$$

$$a_{50} x_0 + a_{51} x_1 + a_{52} x_2 + a_{53} x_3 + a_{54} x_4 + a_{55} x_5$$

# I Strategia: Risultato finale

---

Esempio: MIMD - DM , p=2, N=6

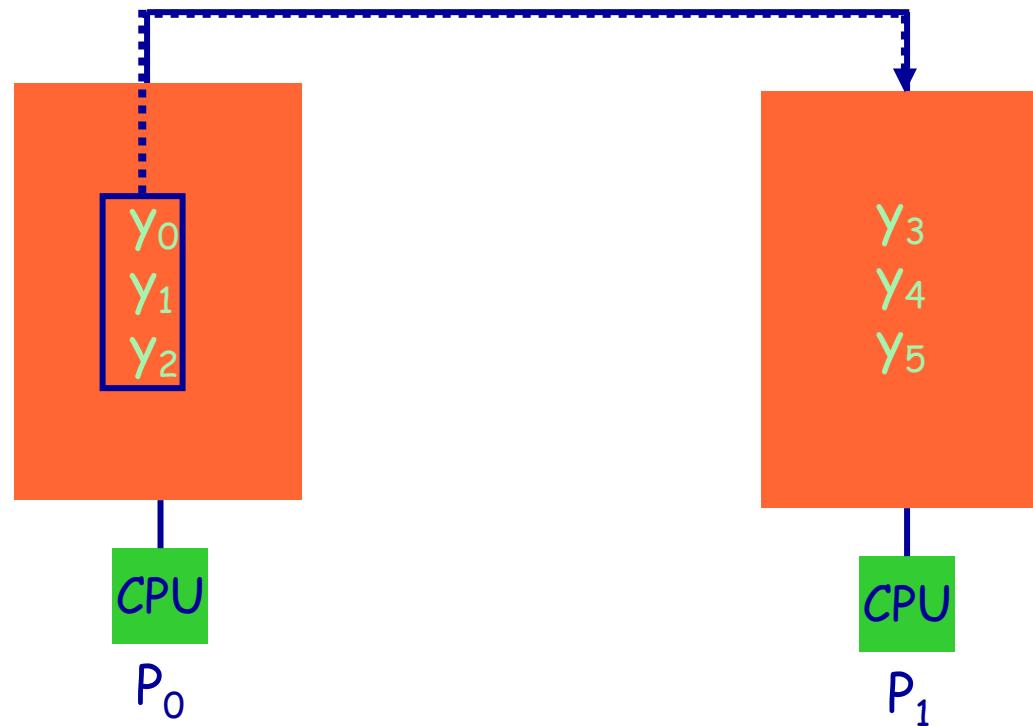


Osservazione

i processori  $P_0$  e  $P_1$  possono effettuare un'operazione collettiva di tipo "gather" riunendo i risultati parziali

# I Strategia: operazione "gather"

Esempio: MIMD - DM , p=2, N=6

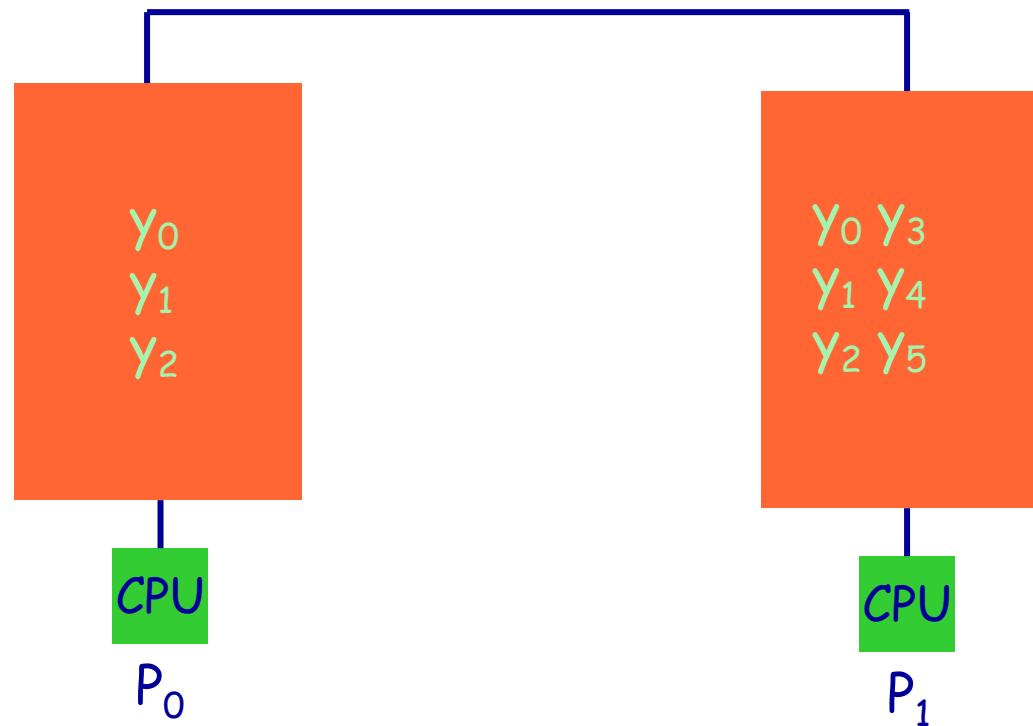


$P_0$  spedisce a  $P_1$  le componenti calcolate di  $y$

# I Strategia: operazione “gather”

---

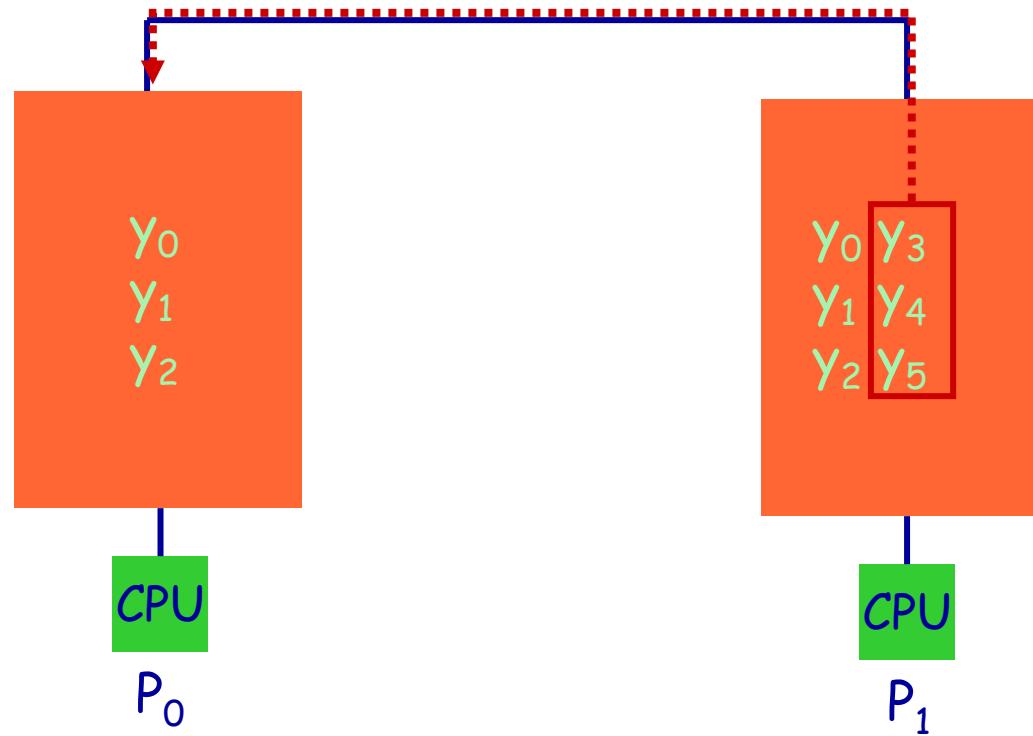
Esempio: MIMD - DM , p=2, N=6



$P_1$  le riunisce in maniera opportuna

# I Strategia: operazione “gather”

Esempio: MIMD - DM , p=2, N=6

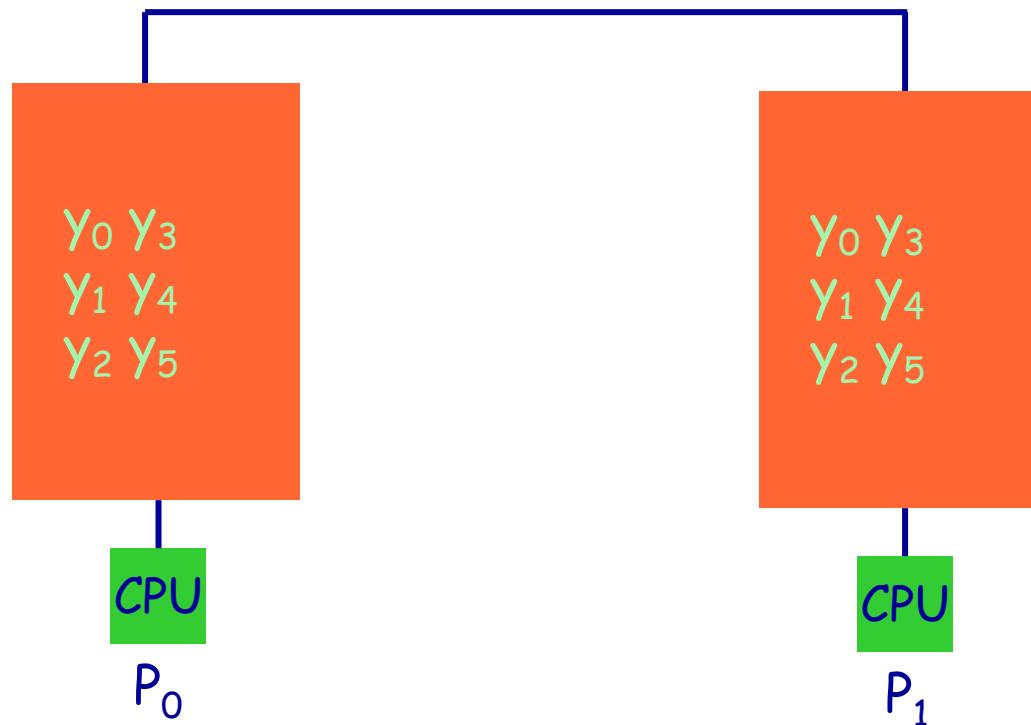


$P_1$  spedisce a  $P_0$  le componenti calcolate di  $y$

# I Strategia: operazione “gather”

---

Esempio: MIMD - DM , p=2, N=6



$P_0$  le riunisce in maniera opportuna

***Entrambi i processori hanno il risultato finale!***

## **II STRATEGIA**

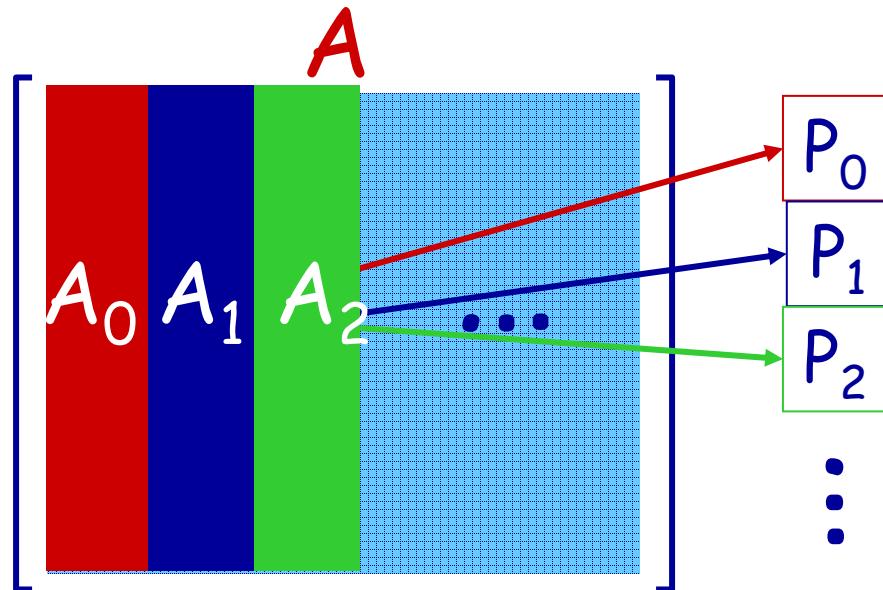
---

**Decomposizione 2**  
suddividiamo  
la matrice A in  
**BLOCCHI di COLONNE**

## II STRATEGIA: In generale

### I passo: decomposizione del problema

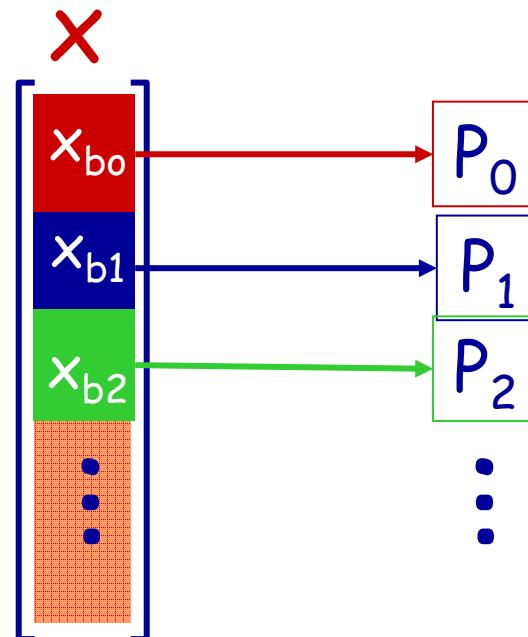
La matrice  $A$  viene distribuita  
in BLOCCHI di COLONNE  
fra p processori



## II STRATEGIA: In generale

### I passo: decomposizione del problema

Il vettore  $x$  viene distribuito  
fra i  $p$  processori



## II STRATEGIA: In generale

---

### II passo: risoluzione dei sottoproblemi

Il prodotto  $Ax=y$  viene decomposto  
in  $p$  prodotti del tipo

$$A_i \cdot x_i = r_i \text{ dove } y = \sum_{i=0}^{p-1} r_i$$

Ciascun processore calcola  
un prodotto matrice vettore  
(di dimensione più piccola di quello assegnato).

# Domanda

---

Qual è l'algoritmo parallelo  
con la II Strategia  
di decomposizione

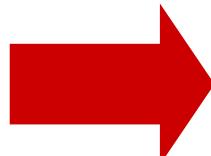
?

# Risposta

## Partizionamento della matrice in blocchi di colonne



```
begin
    y=0
    for i=0 to p-1 do
         $r_i = A_i \cdot x_i$ 
         $y=y+r_i$ 
    endfor
end
```



### Algoritmo parallelo

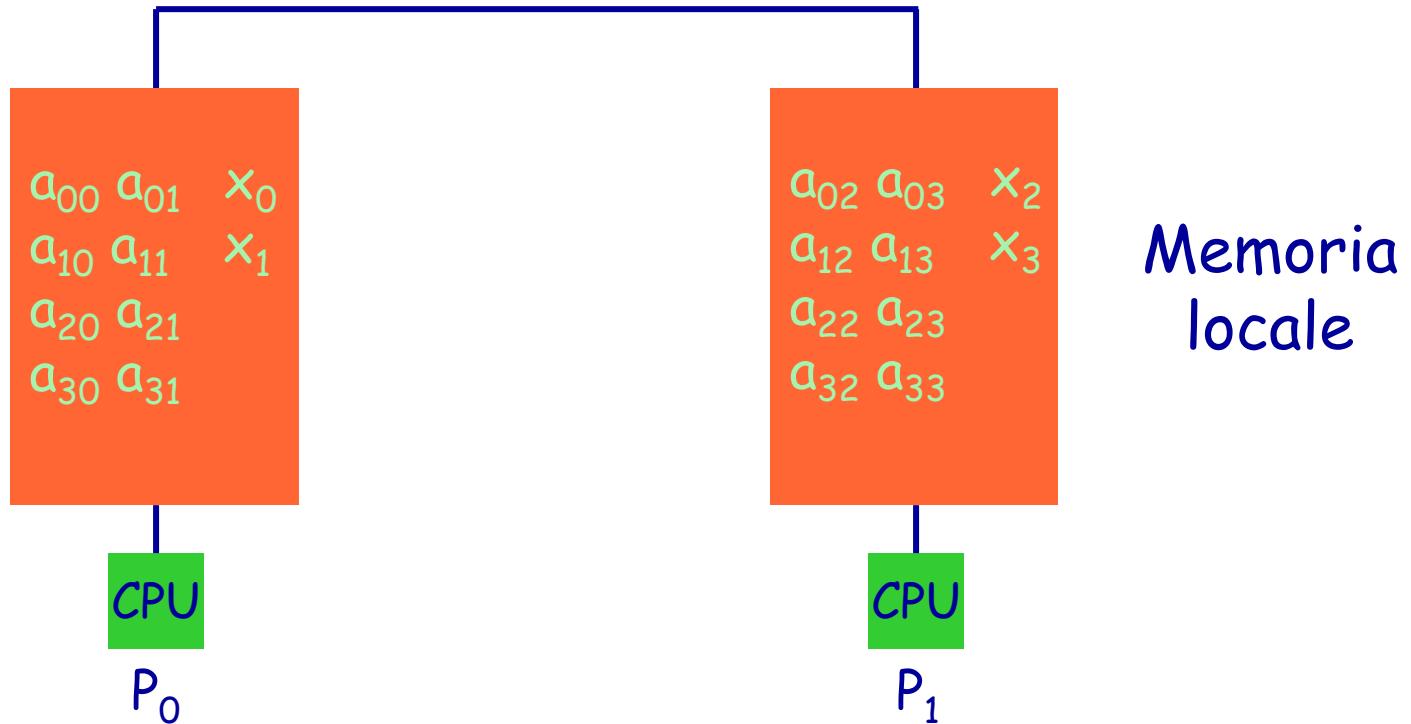
```
begin
    forall  $P_i$ ,  $i=0, p-1$ 
         $\{P_i \text{ calcola } r_i = A_i \cdot x_i\}$ 
        { combinazione degli  $r_i$ }
         $y=y+r_i$ 
    endfor
end
```

Parallelizzazione dell'algoritmo a blocchi!

# II Strategia: Distribuzione dei dati

---

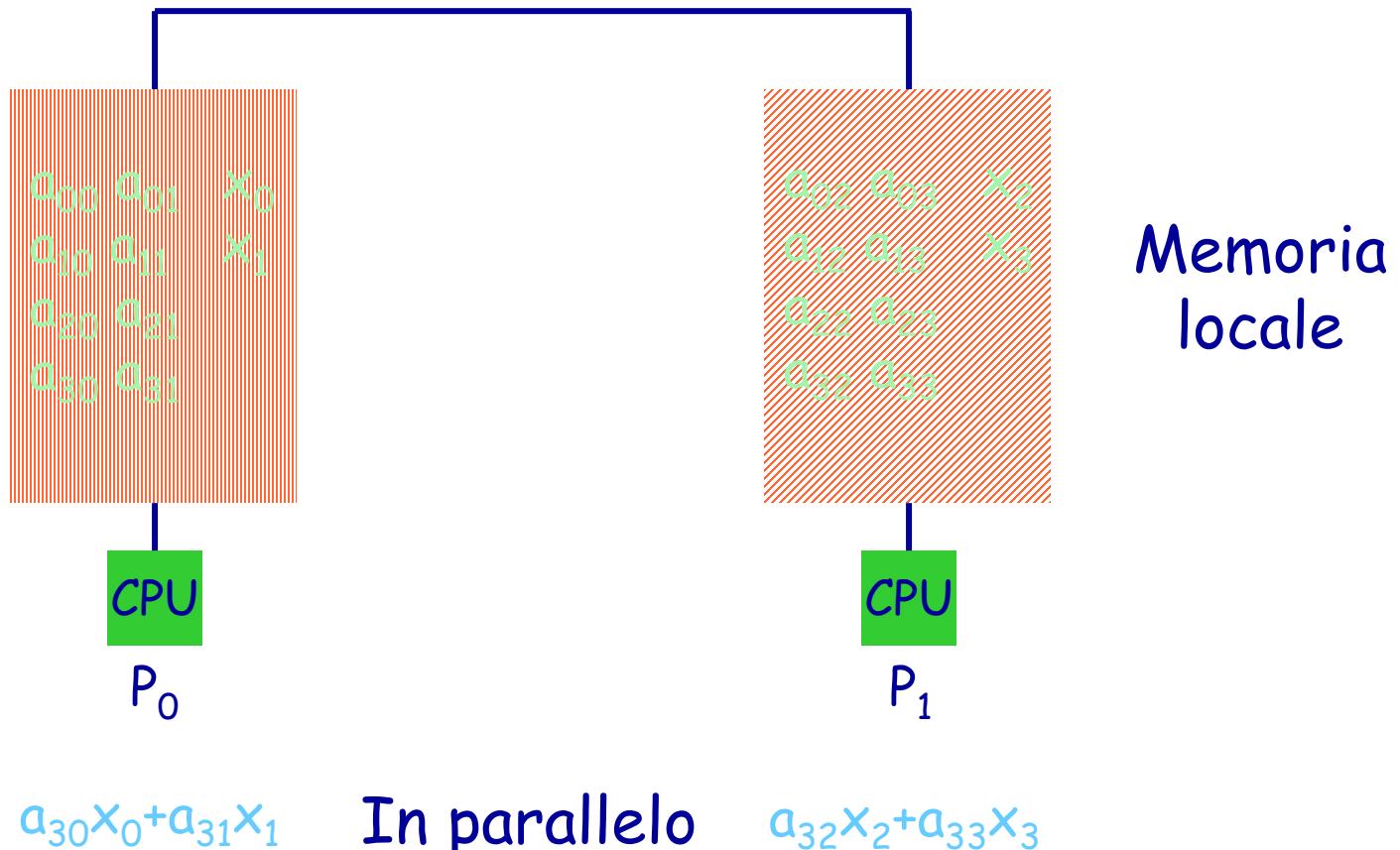
Esempio:  $N=4$ ,  $p=2$



# II Strategia: Fase di calcolo

---

Esempio:  $N=4$ ,  $p=2$



Memoria  
locale

$a_{00}$	$a_{01}$	$x_0$
$a_{10}$	$a_{11}$	$x_1$
$a_{20}$	$a_{21}$	
$a_{30}$	$a_{31}$	

CPU

$P_0$

$$a_{30}x_0 + a_{31}x_1$$

In parallelo

$a_{02}$	$a_{03}$	$x_2$
$a_{12}$	$a_{13}$	$x_3$
$a_{22}$	$a_{23}$	
$a_{32}$	$a_{33}$	

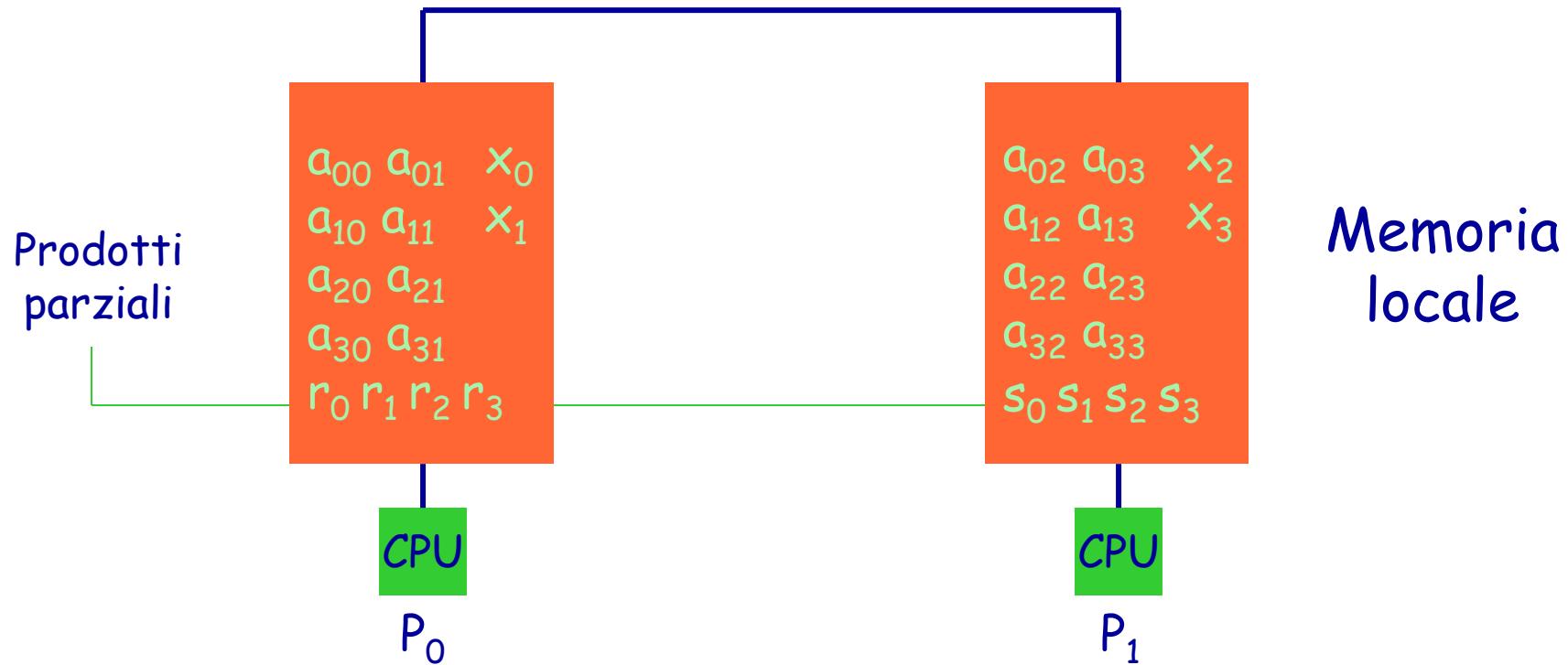
CPU

$P_1$

$$a_{32}x_2 + a_{33}x_3$$

# II Strategia

Esempio:  $N=4$ ,  $p=2$

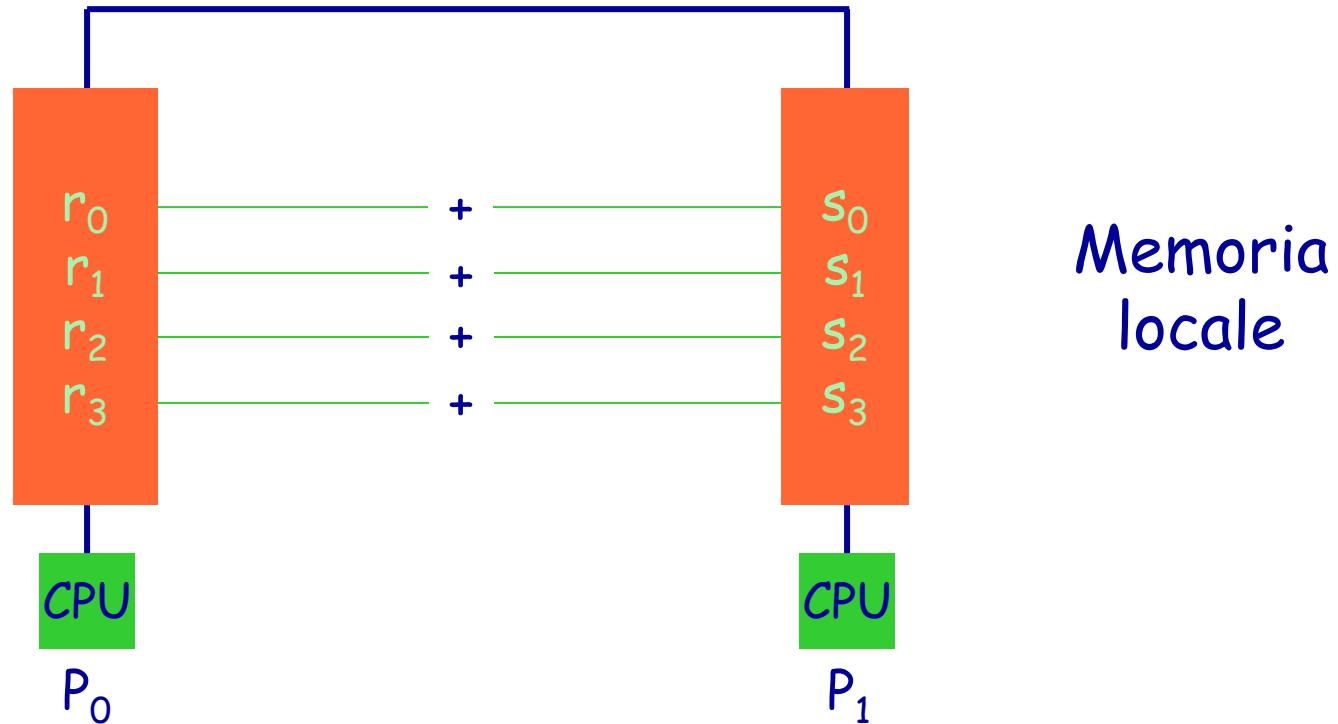


Come calcolare il risultato finale?

# II Strategia

---

Esempio:  $N=4$ ,  $p=2$

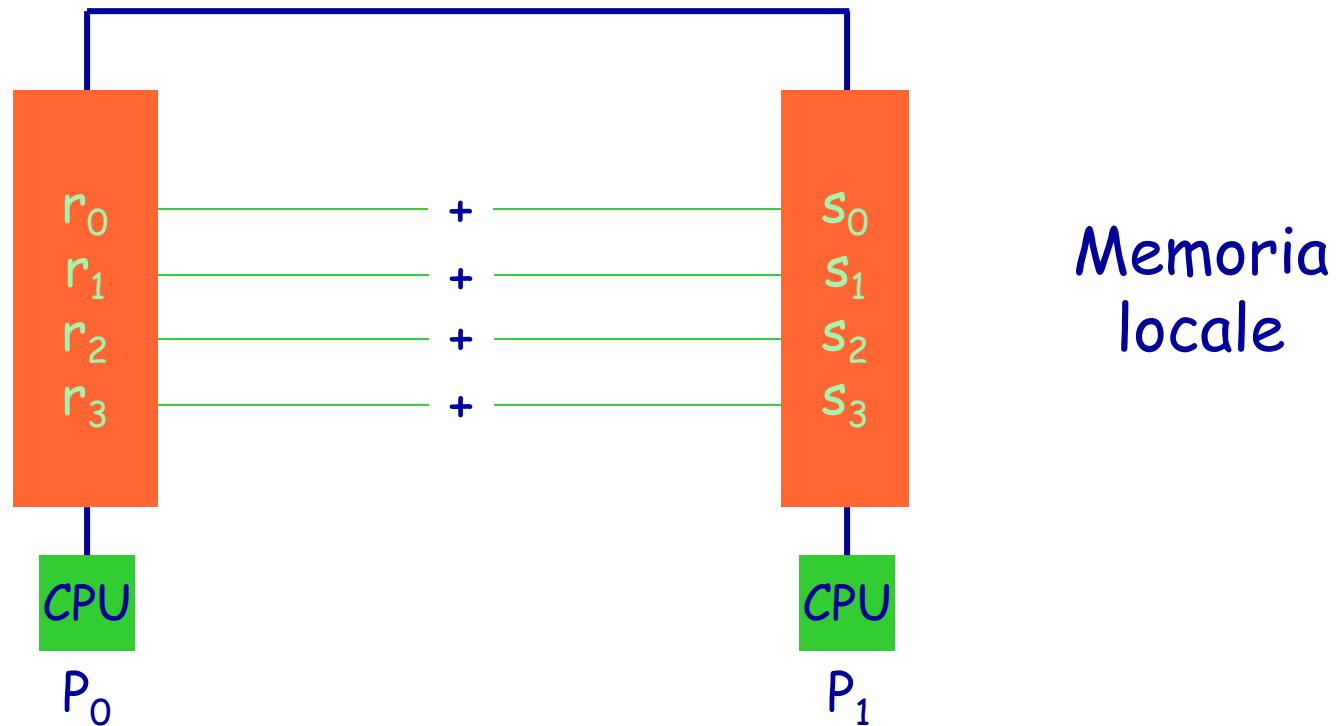


si devono sommare 2 vettori distribuiti tra i 2 processori

# Come sommare i due vettori ?

---

Esempio:  $N=4$ ,  $p=2$



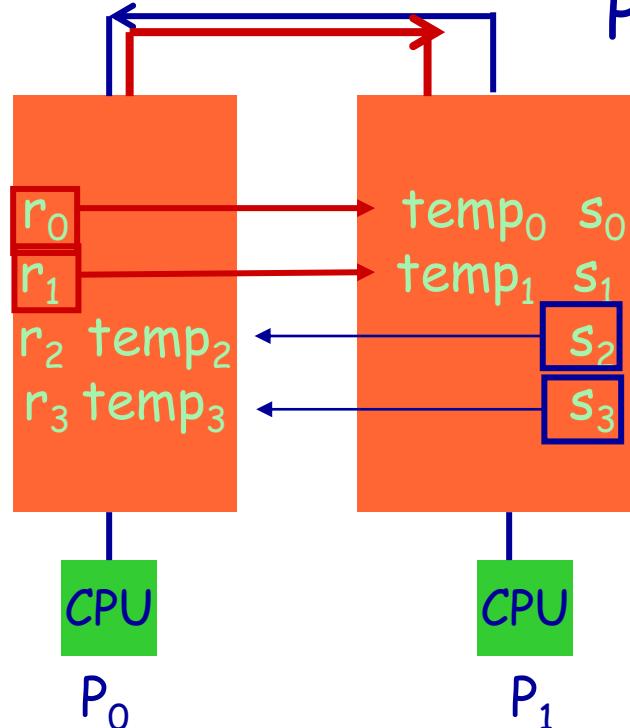
# Esempio: N=4, p=2

I passo

Spedizione

$P_0$  invia a  $P_1$   $r_0$  e  $r_1$

$P_1$  invia a  $P_0$   $s_2$  e  $s_3$



Calcolo in  $P_0$

$$y_2 = s_2 + \text{temp}_2$$

$$y_3 = s_3 + \text{temp}_3$$

Calcolo in  $P_1$

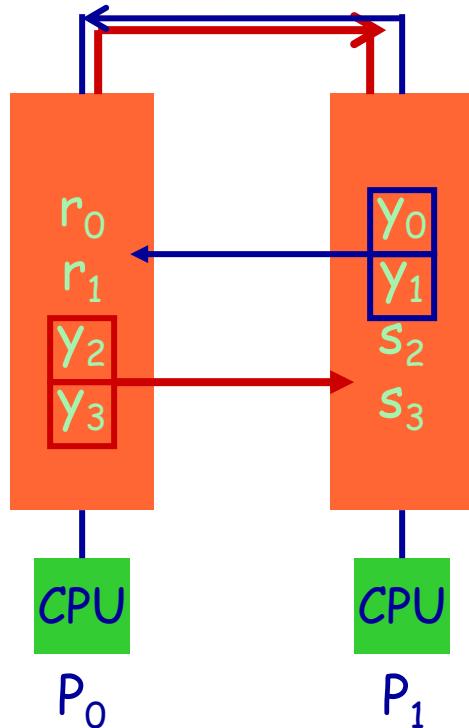
$$y_0 = r_0 + \text{temp}_0$$

$$y_1 = r_1 + \text{temp}_1$$

# Esempio: N=4, p=2

II passo

Spedizione



$P_0$  invia a  $P_1$   $y_2$  e  $y_3$

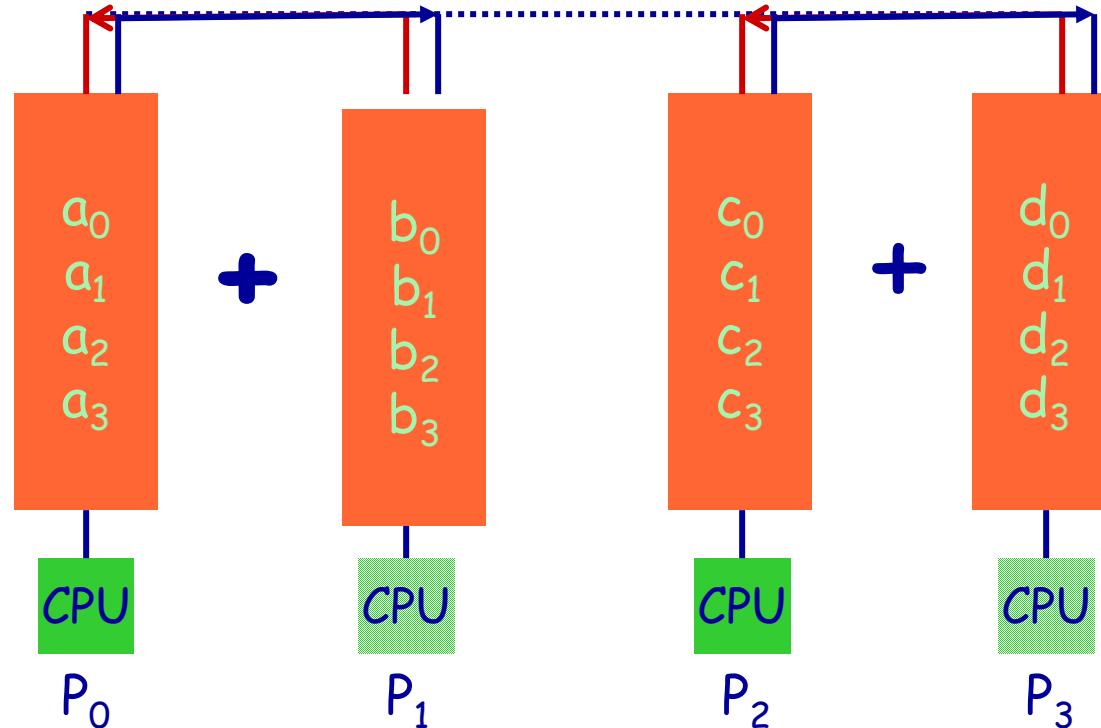
$P_1$  invia a  $P_0$   $y_0$  e  $y_1$

Al termine del II passo  
entrambi i processori  
hanno tutto il vettore  $y$

# Esempio N=4, p=4

---

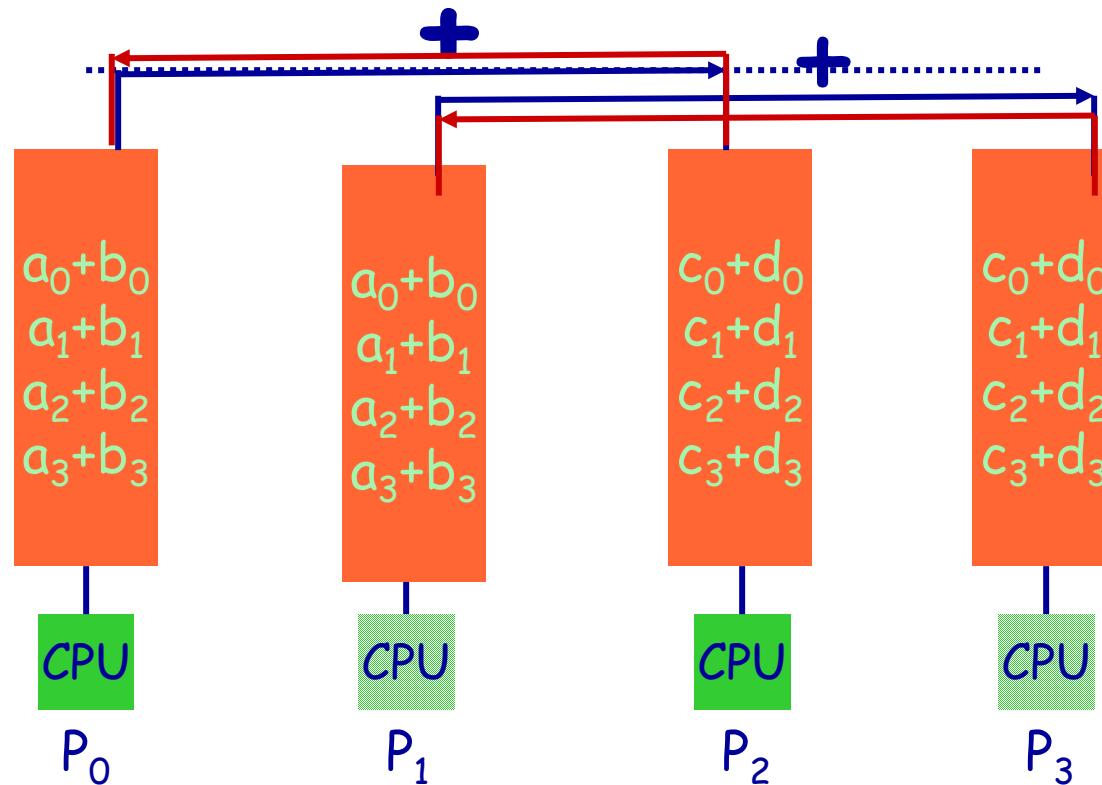
I passo



$P_0$  e  $P_1$  calcolano in parallelo la somma di 2 vettori  
 $P_2$  e  $P_3$  calcolano in parallelo la somma di 2 vettori

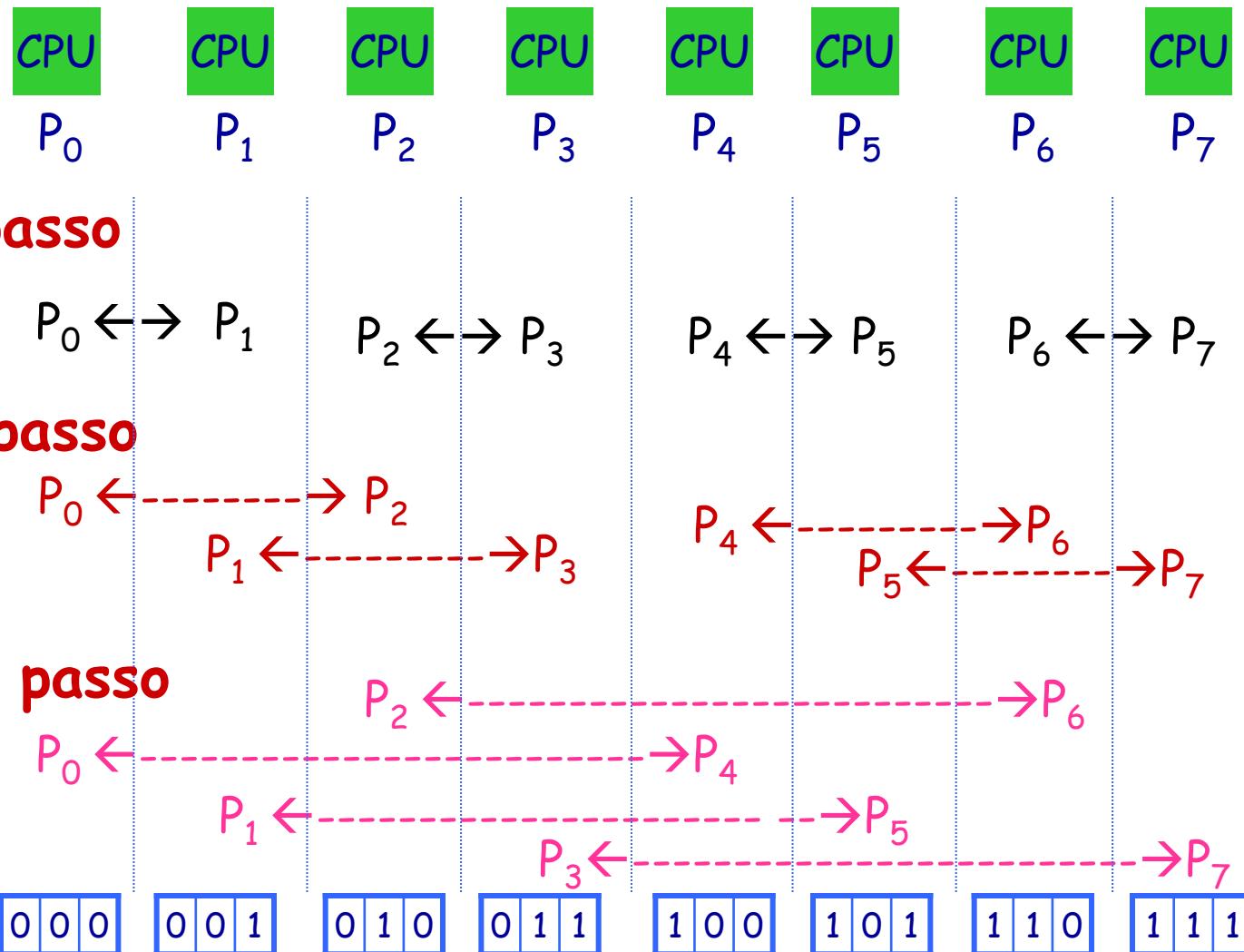
# Esempio N=4, p=4

II passo

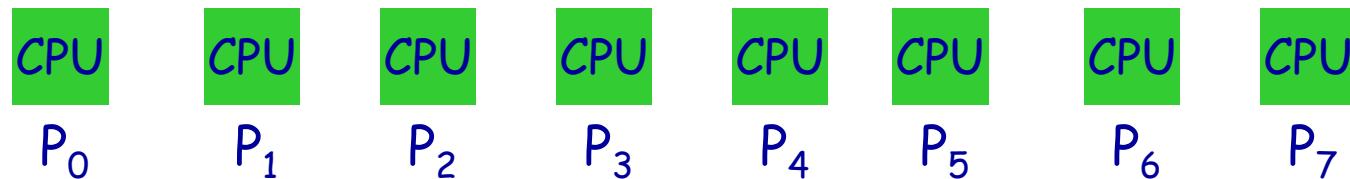


$P_1, P_2, P_3, P_4$  calcolano in parallelo la somma di 2 vettori

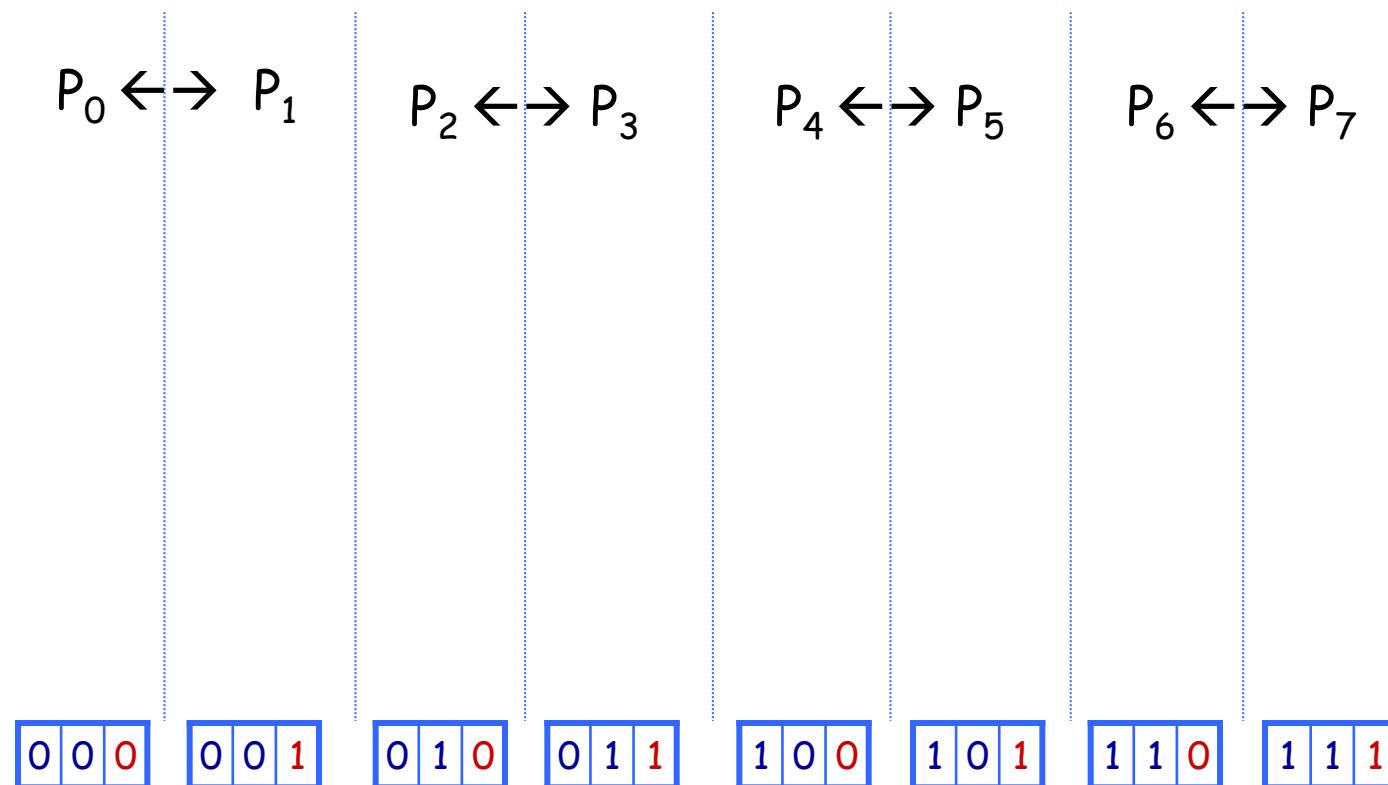
# Esempio N=8, p=8



# Esempio N=8, p=8



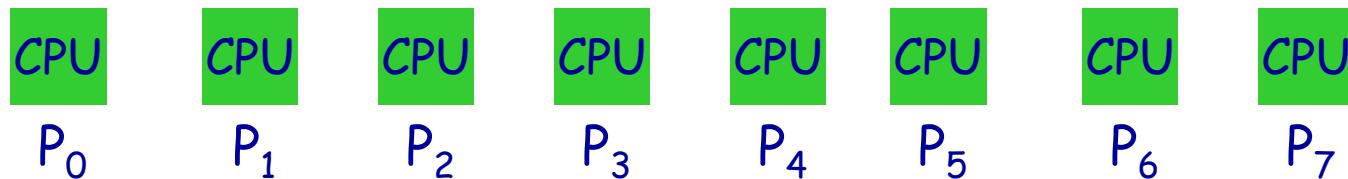
I passo - bit 0



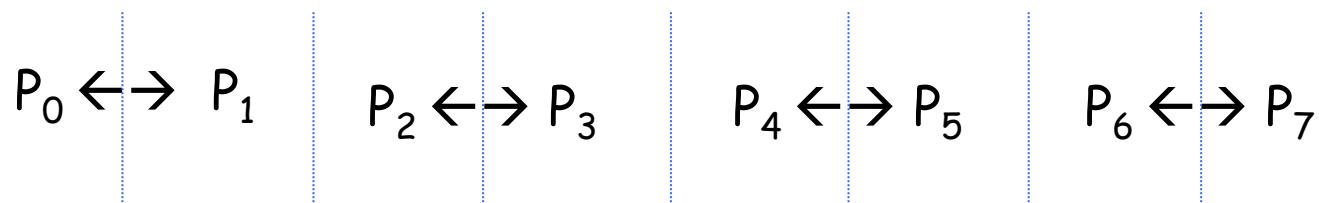
I Processori il cui bit zero è 0 comunicano con  $m_{enum}+2^0$

Prof. G. Lacc I Processori il cui bit zero è 1 comunicano con  $m_{enum}-2^0$

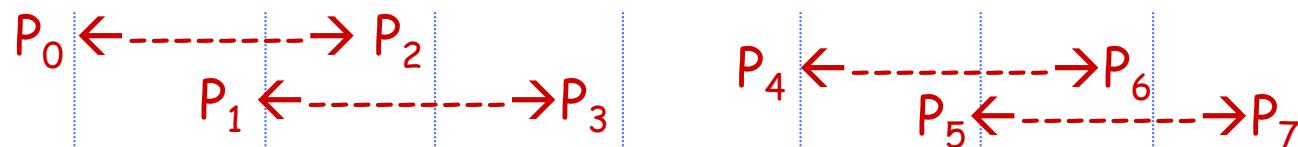
# Esempio N=8, p=8



I passo - bit 0



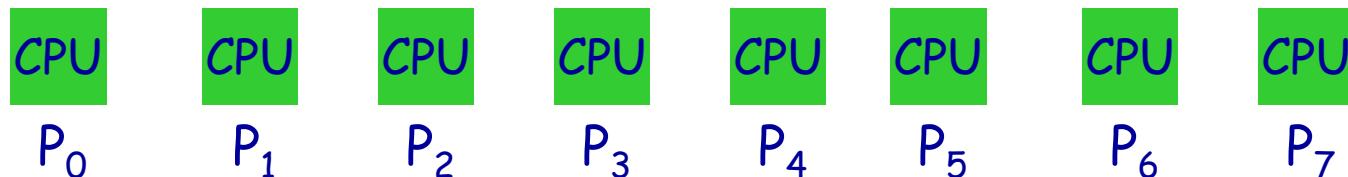
II passo - bit 1



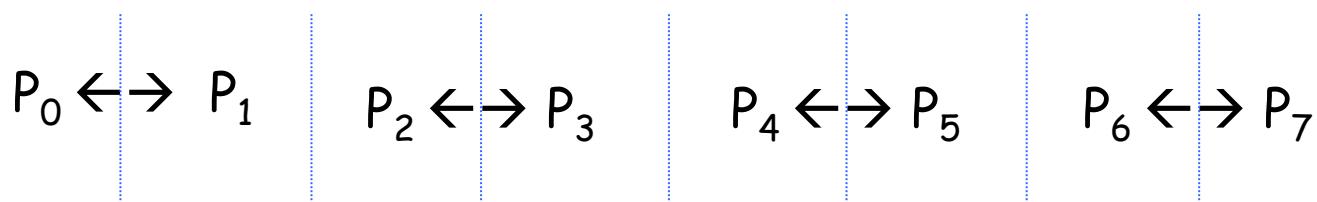
I Processori il cui bit uno è 0 comunicano con  $m_{\text{enum}}+2^1$

I Processori il cui bit uno è 1 comunicano con  $m_{\text{enum}}-2^1$

# Esempio N=8, p=8



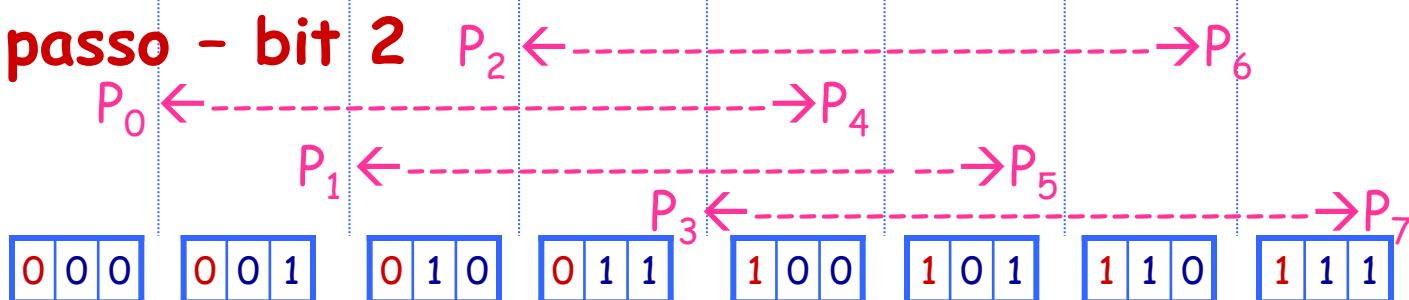
I passo - bit 0



II passo - bit 1



III passo - bit 2



I Processori il cui bit due è 0 comunicano con  $m_{enum+2^2}$

Prof. G. Lacc I Processori il cui bit due è 1 comunicano con  $m_{enum-2^2}$

# Domanda

---

E' possibile realizzare  
un'altra decomposizione  
del problema:  
prodotto  
Matrice-Vettore

?

# Risposta: SI!

---

Decomposizione 1: BLOCCHI di RIGHE

+

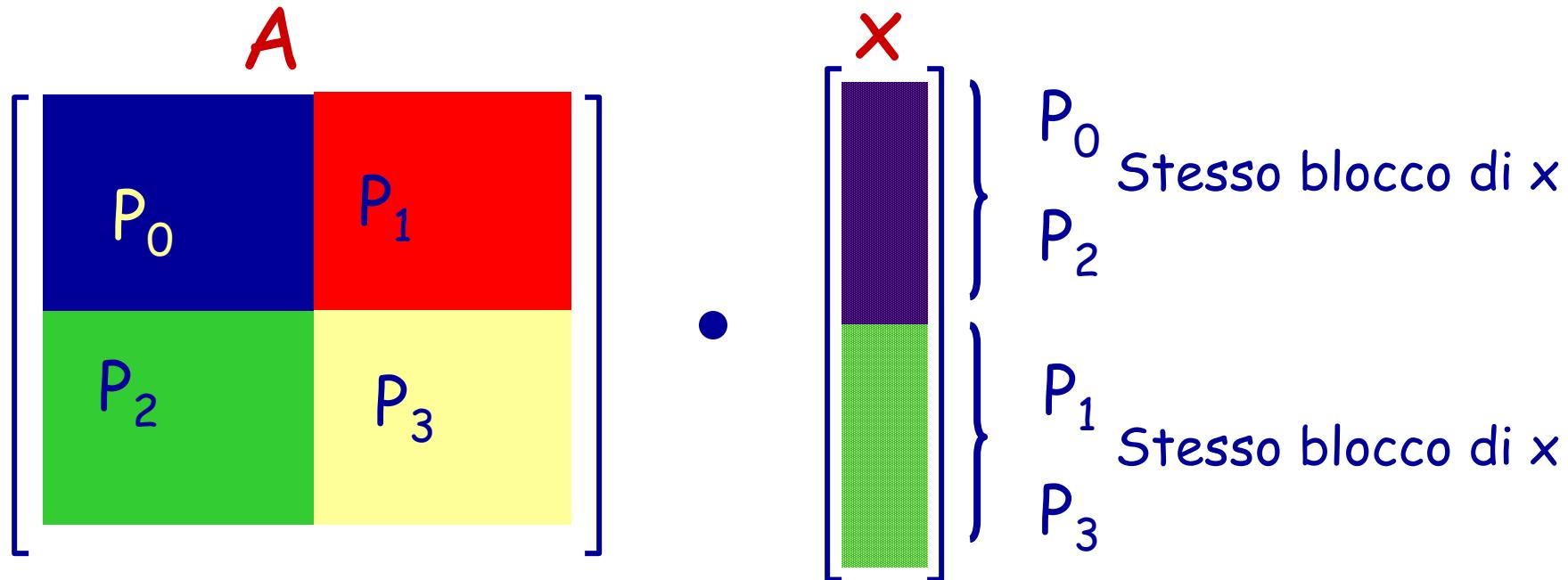
Decomposizione 2: BLOCCHI di COLONNE

=

Decomposizione 3: BLOCCHI QUADRATI

### III Strategia: Esempio (4 processori)

Distribuzione della matrice A per blocchi quadrati



Distribuzione del vettore  $x$  fra i processori

# Domanda

---

Ciascun processore  
quale “parte” di  $y$   
calcola

?

# Esempio N = 10 , Processori=4

P<sub>0</sub>

$$a_{00}x_0 + a_{01}x_1 + \dots + a_{04}x_4$$

$$a_{40}x_0 + a_{41}x_1 + \dots + a_{44}x_4$$

P<sub>1</sub>

$$a_{05}x_5 + a_{06}x_6 + \dots + a_{09}x_9$$

$$a_{45}x_5 + a_{46}x_6 + \dots + a_{49}x_9$$

$$a_{50}x_0 + a_{51}x_1 + \dots + a_{54}x_4$$

$$a_{90}x_0 + a_{91}x_1 + \dots + a_{94}x_4$$

P<sub>2</sub>

$$a_{55}x_5 + a_{56}x_6 + \dots + a_{59}x_9$$

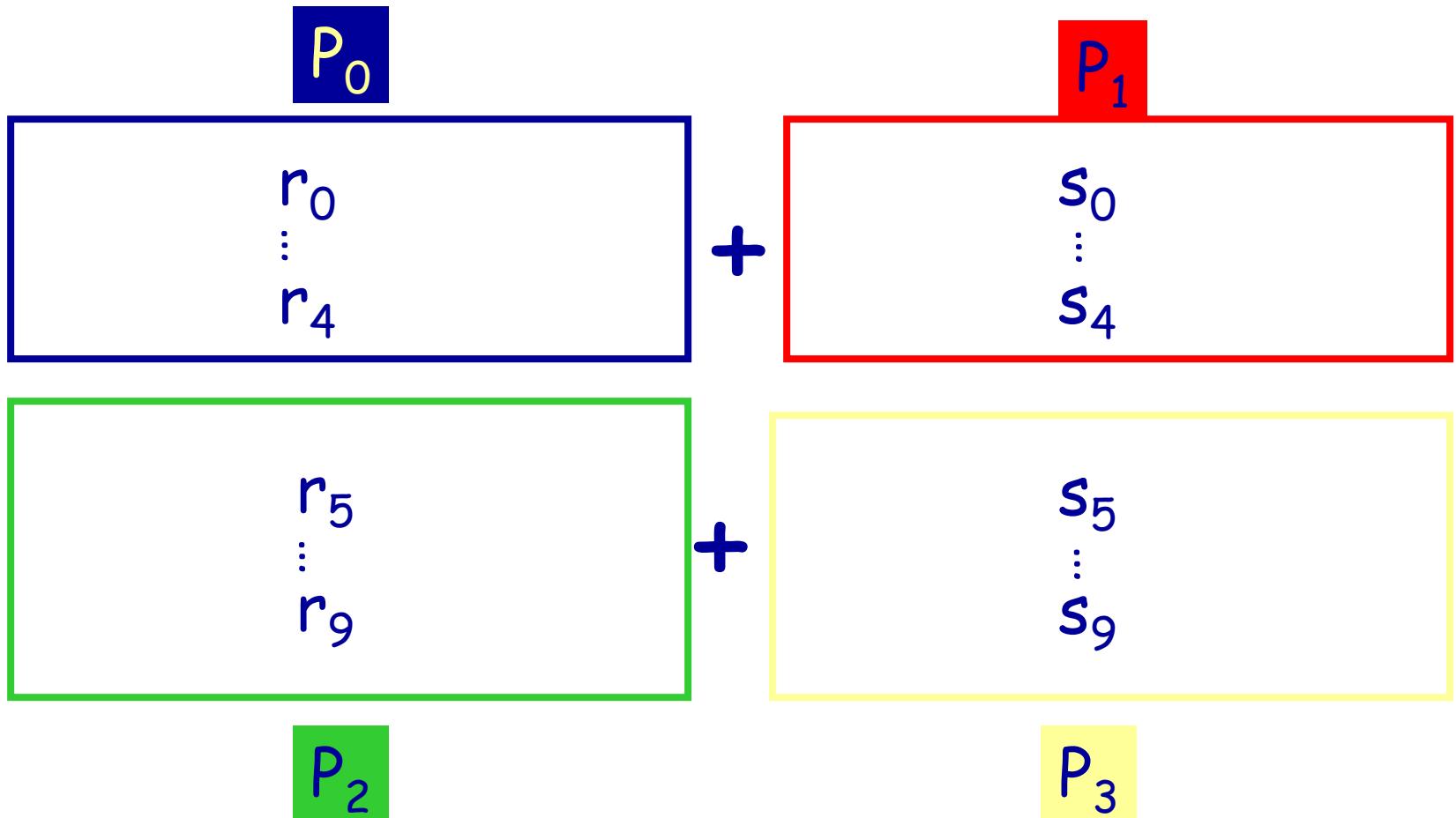
$$a_{95}x_5 + a_{96}x_6 + \dots + a_{99}x_9$$

P<sub>3</sub>

Calcolo dei prodotti parziali

# Esempio $N = 10$ , $P=4$

---

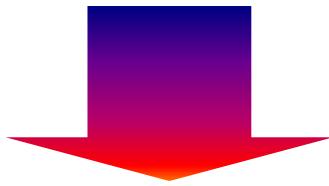


Comunicazione: somma in parallelo

# III strategia: sintesi

---

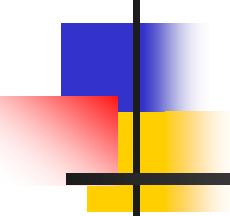
Ciascun processore calcola  
somme parziali  
di alcune componenti del vettore  $y$



I processori devono sommare i risultati parziali  
e scambiarsi le componenti  
per avere il risultato finale,  $y$

---

FINE APPROFONDIMENTO....



# Parallel and Distributed Computing

a.a. 2021-2022

## Somma di N numeri

Su architettura MIMD-DM

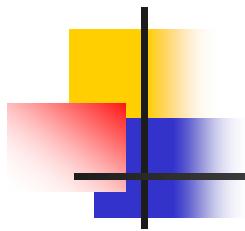
**Prof. Giuliano Laccetti**

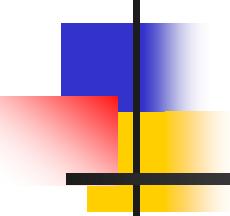
Materiale tratto dal testo

A. Murli – Lezioni di Calcolo Parallelo, Liguori

e

da Appunti e Lezioni tenute dal prof. Murli in vari corsi in a.a. precedenti





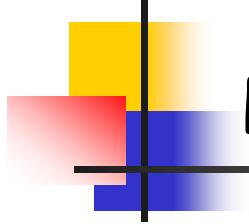
# Parallel and Distributed Computing

a.a. 2021-2022

## Somma di N numeri

Su architettura MIMD-DM

Prof. Giuliano Laccetti

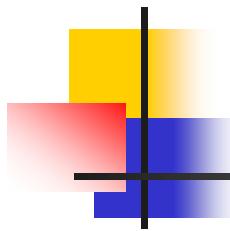


## PROBLEMA:

Calcolo della somma di  $N$  numeri

$$a_0 + a_1 + \dots + a_{N-1}$$

su un calcolatore parallelo  
tipo **MIMD** con  $p$  processori  
**A MEMORIA DISTRIBUITA**



# Somma di N numeri

Su un calcolatore monoprocessoresso la somma è calcolata eseguendo le  $N-1$  addizioni una per volta secondo un ordine prestabilito

$$\text{sumtot} := a_0$$

$$\text{sumtot} := \text{sumtot} + a_1$$

$$\text{sumtot} := \text{sumtot} + a_2$$

⋮

$$\text{sumtot} := \text{sumtot} + a_{N-1}$$

# Somma di N numeri

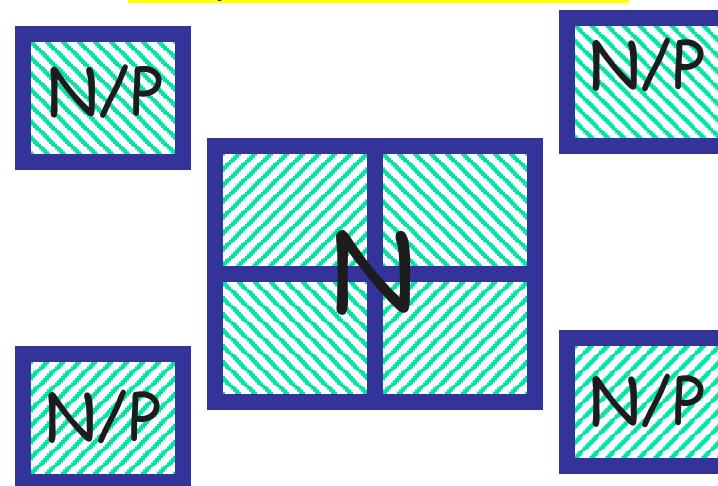
Su un calcolatore monoprocessoresso la somma è calcolata eseguendo le  $N-1$  addizioni una per volta secondo un ordine prestabilito

```
begin
    sumtot:= a0;
    for i=1 to N-1 do
        sumtot:= sumtot+ai ;
    endfor
end
```

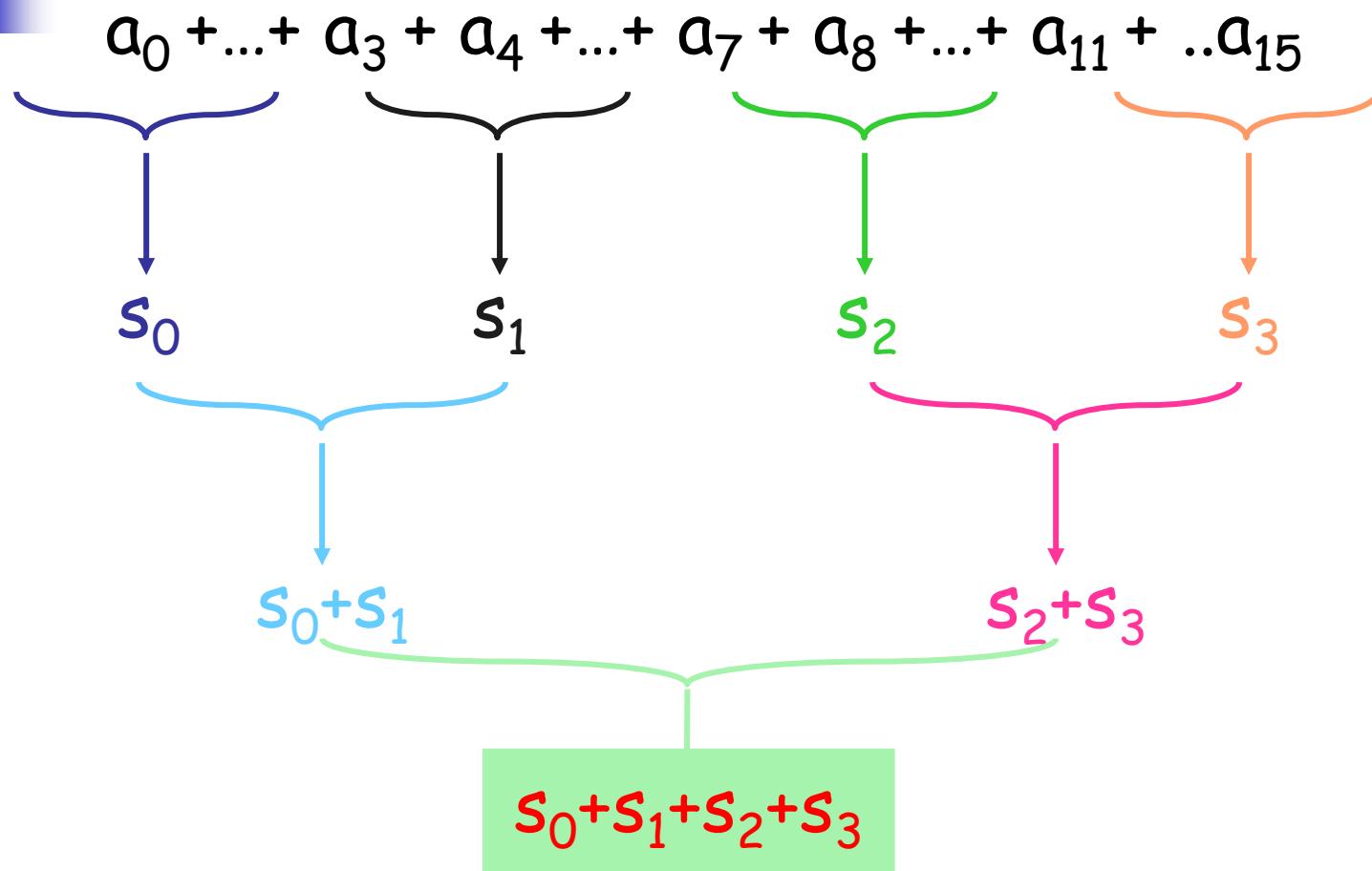
Qual è  
l'ALGORITMO PARALLELO?

# CALCOLO PARALLELO

Decomporre un problema di dimensione  $N$   
in  $P$  sottoproblemi di dimensione  $N/P$   
e risolverli contemporaneamente  
su più calcolatori

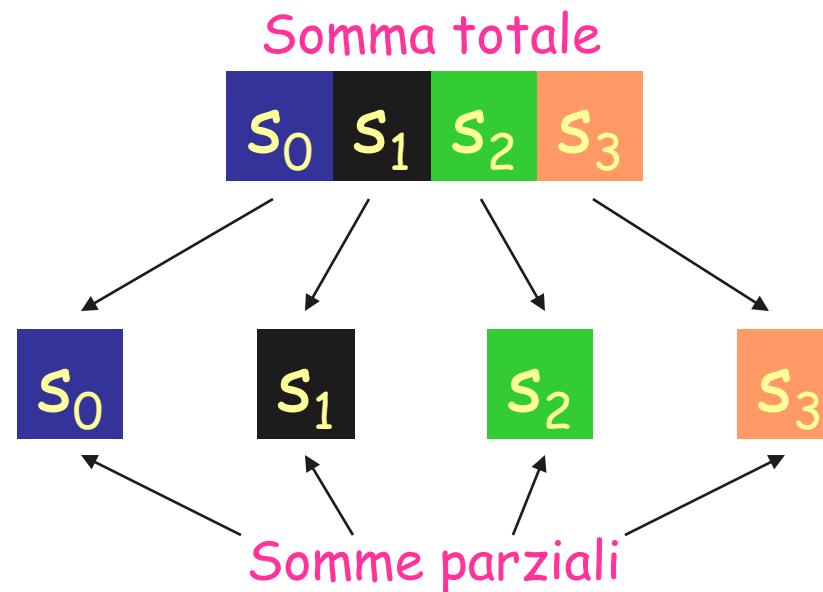


## Esempio: N=16, p=4



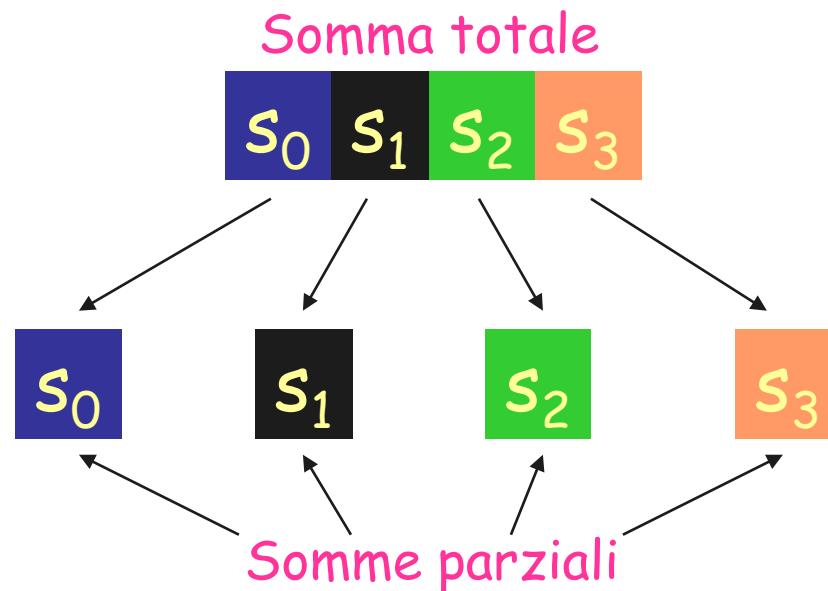
# IDEA

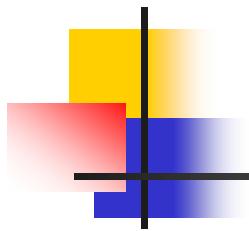
Suddividere la somma in somme parziali  
ed assegnare ciascuna somma parziale  
ad un processore



# IDEA

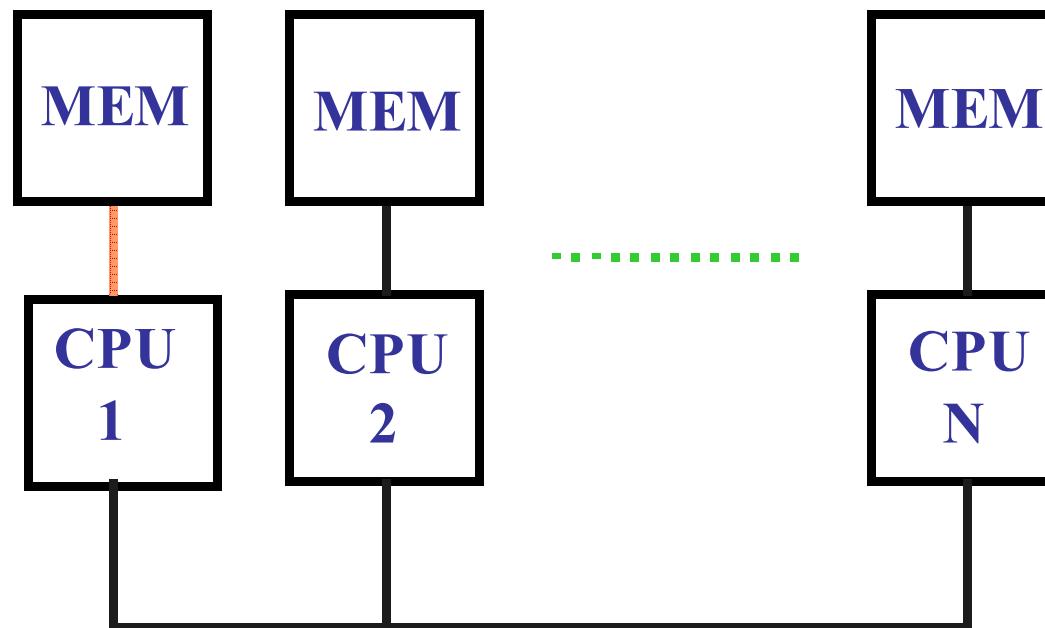
Le somme parziali devono poi essere  
combinate in modo opportuno per  
ottenere la somma totale





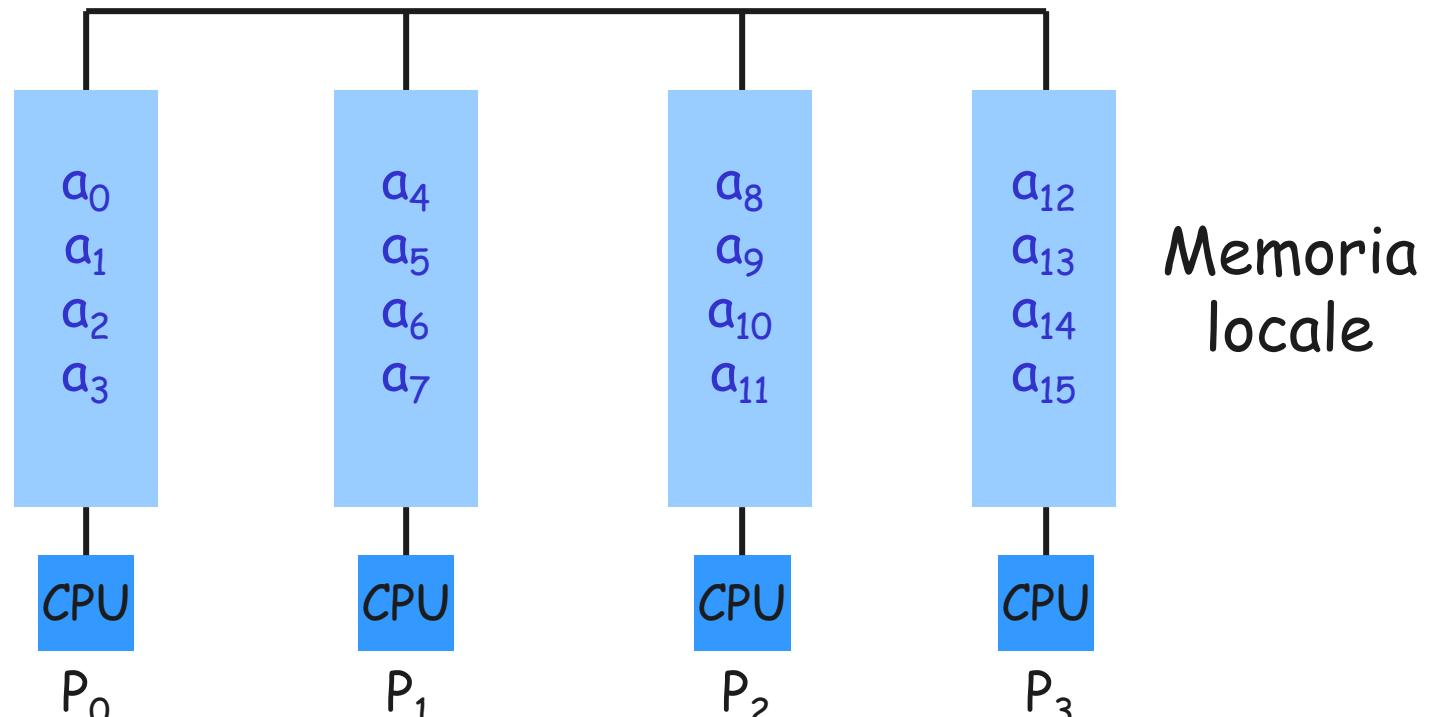
# Schema Calcolatori

## MIMD a memoria distribuita (distributed-memory)



# Somma - MIMD Distributed Memory

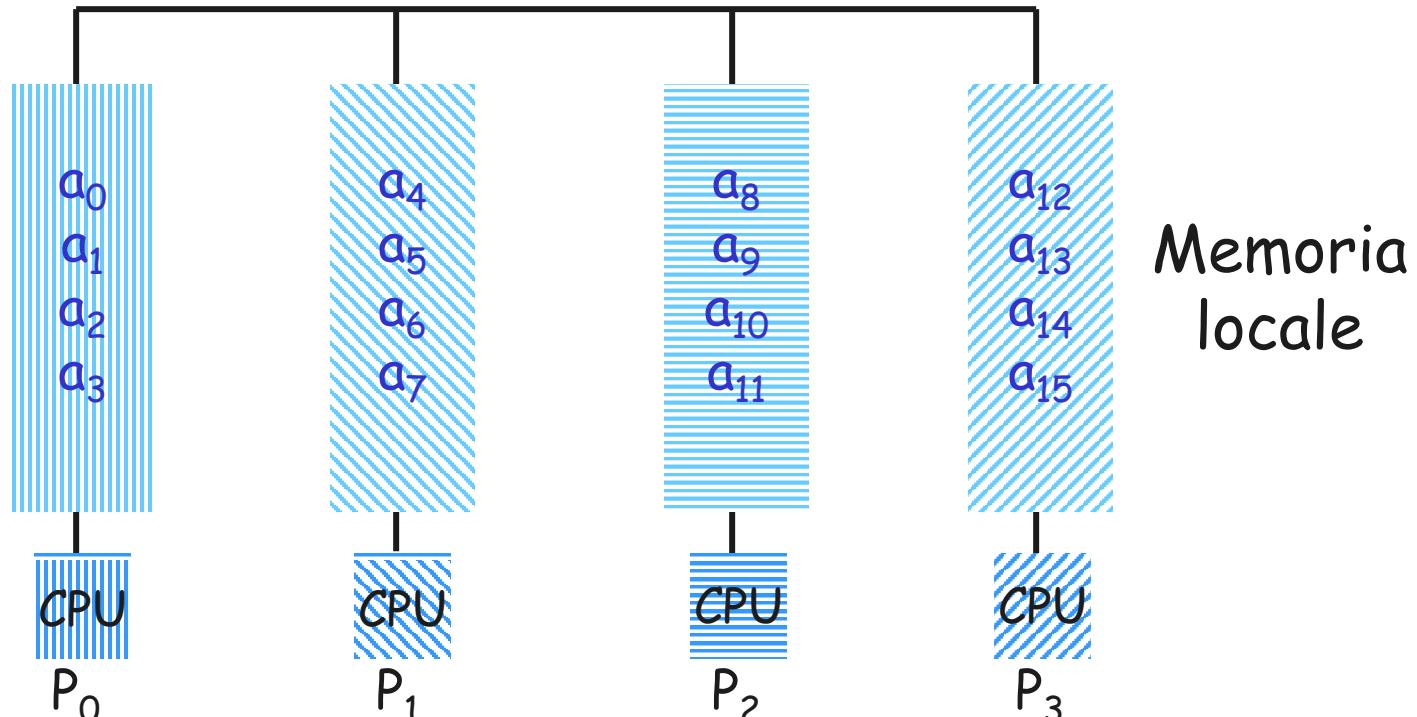
Esempio:  $N=16$ ,  $p=4$



Distribuzione degli addendi fra i processori.

# Somma - MIMD Distributed Memory

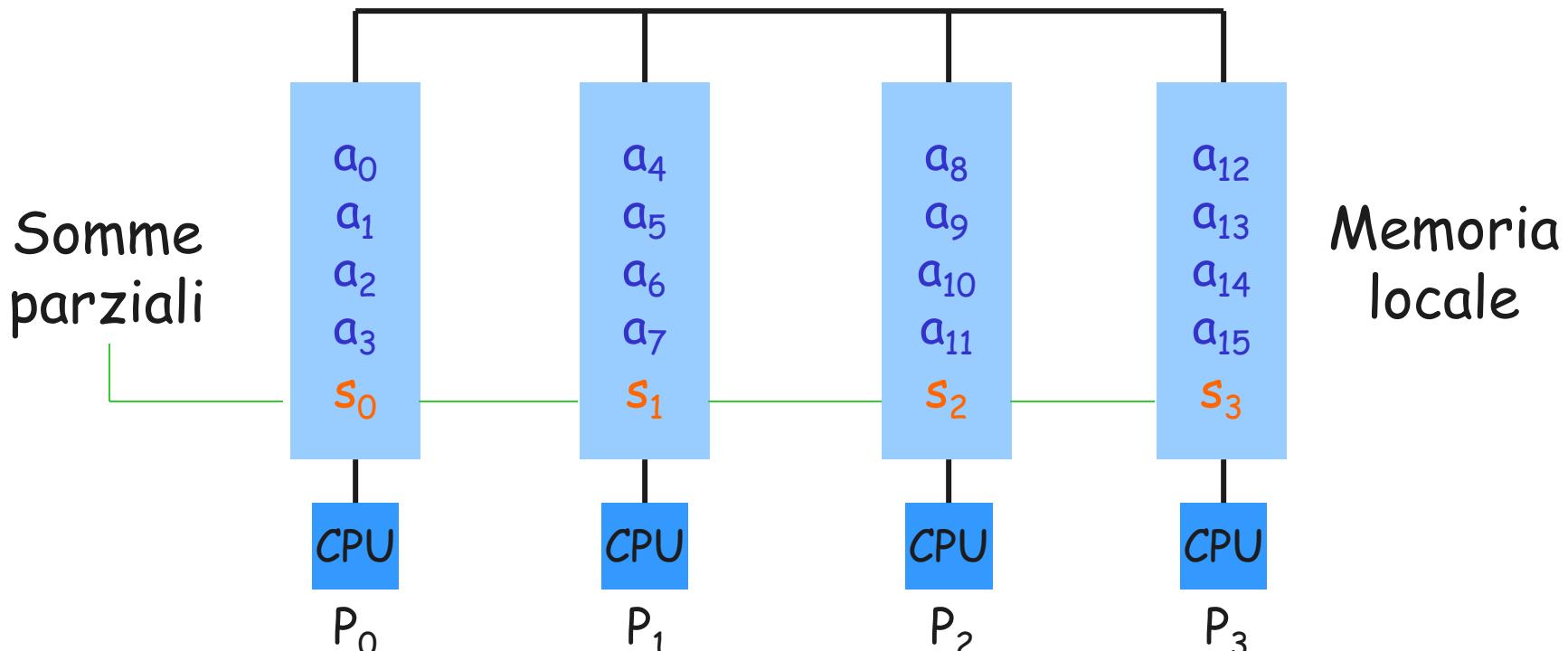
Esempio:  $N=16$ ,  $p=4$



Concorrentemente tutti i processori  
calcolano una somma parziale

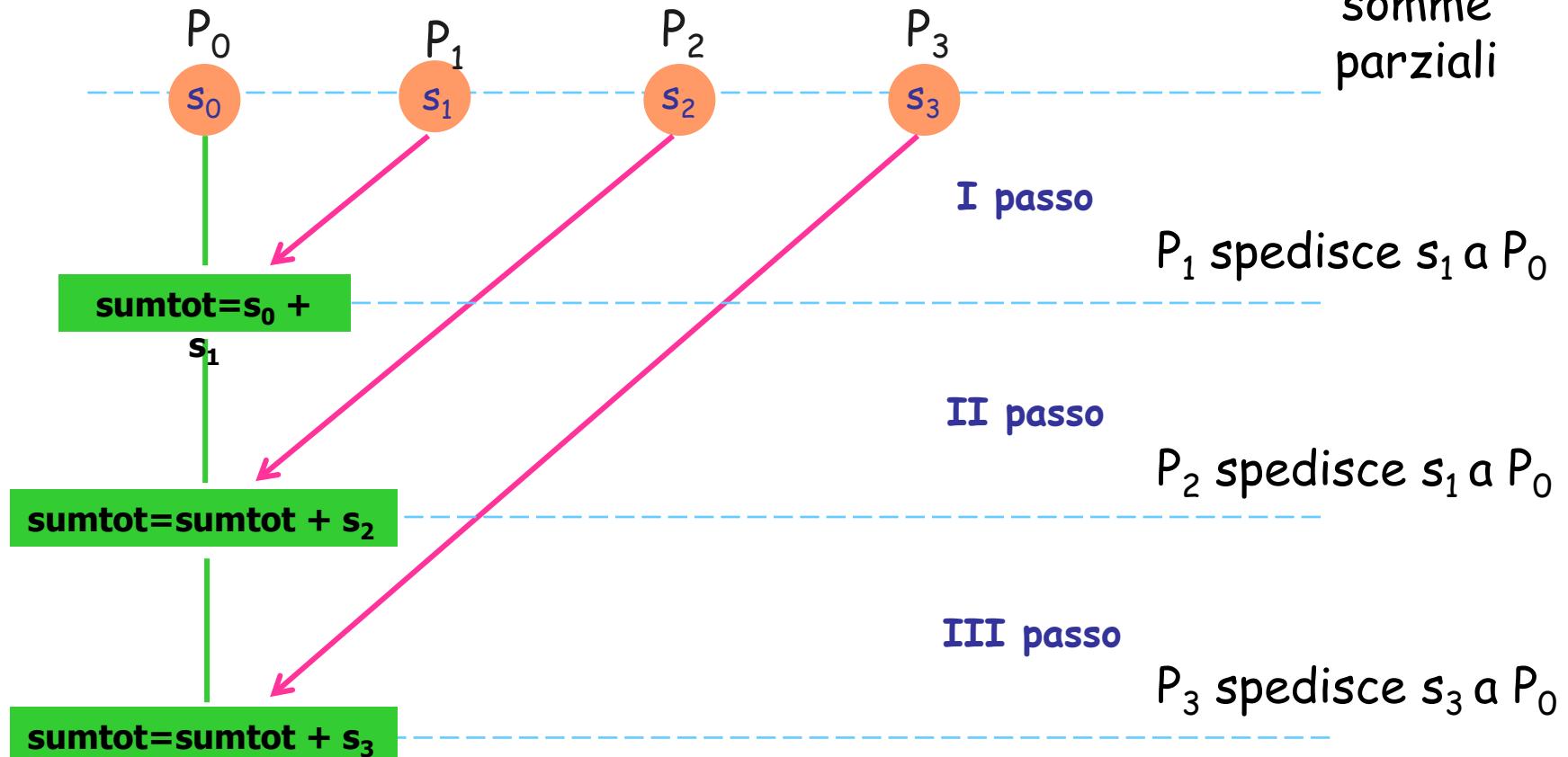
# Somma - MIMD Distributed Memory

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



Come calcolare la somma totale?

## I strategia



# I strategia

Ogni processore

- calcola la propria somma parziale

Ad ogni passo

- ciascun processore invia tale valore ad un unico processore prestabilito

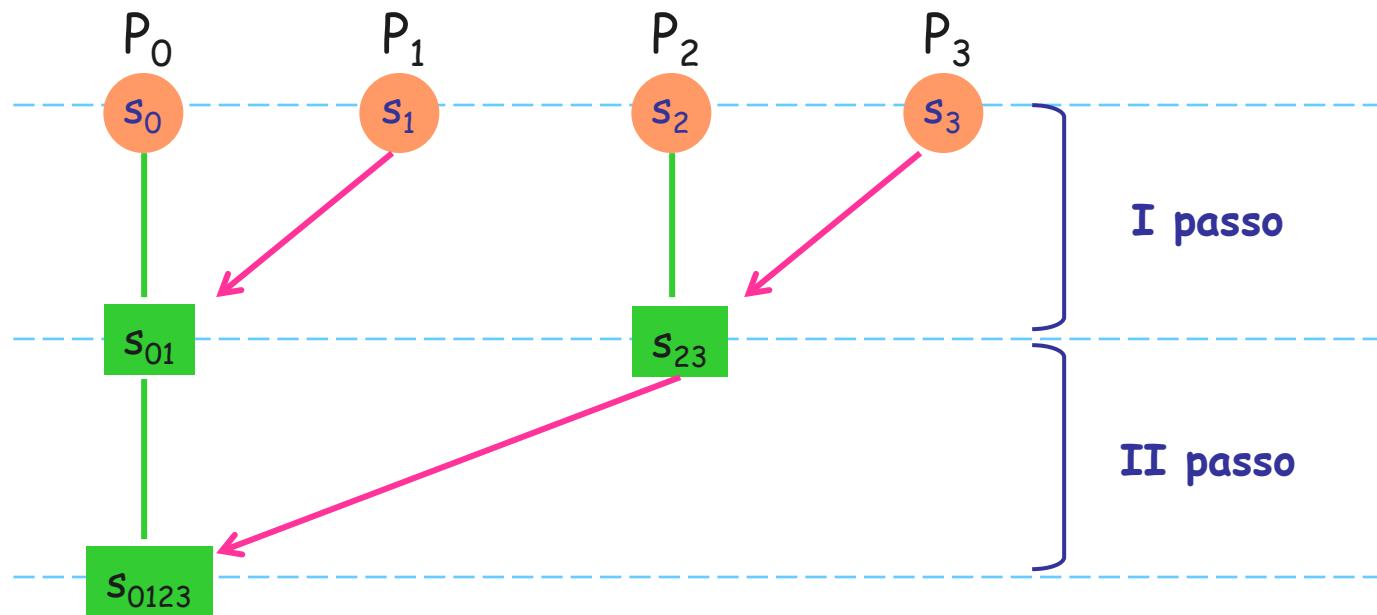
Tale processore contiene la somma totale.

Operazioni concorrenti

## II strategia

$p=4$

Calcolo  
somme  
parziali



Il numero di "passi" è 2

## II strategia

Ogni processore

- calcola la propria somma parziale.

Ad ogni passo,

- coppie distinte di processori comunicano contemporaneamente
- in ogni coppia, un processore invia all'altro la propria somma parziale che provvede all'aggiornamento della somma

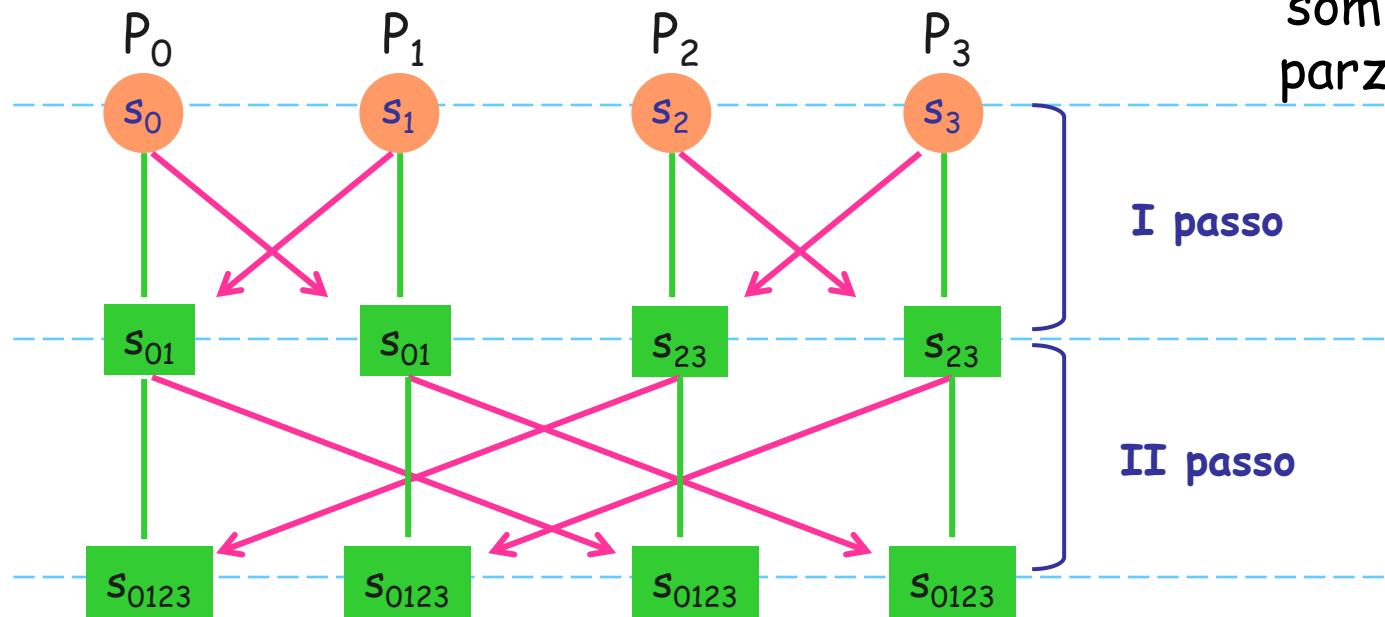
Il risultato è in un unico processore predefinito

Operazioni concorrenti

### III strategia

$p=4$

Calcolo  
somme  
parziali



Il numero di "passi" è 2

# III strategia

Ogni processore

- calcola la propria somma parziale .

Ad ogni passo

- coppie distinte di processori comunicano contemporaneamente:

- in ogni coppia i processori si scambiano le proprie somme parziali

Il risultato è in tutti i processori

Operazioni concorrenti

## I, II, III strategia

Ogni processore calcola la sua somma parziale  
ed invia tale valore agli altri processori in  
modo da ottenere la somma totale

PARALLELISMO

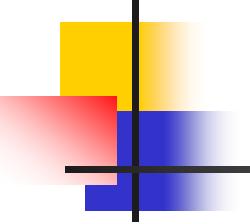


COMUNICAZIONE  
TRA I PROCESSORI

# Algoritmo per la somma di $N=kp$ numeri su MIMD-Distributed Memory

## I Strategia

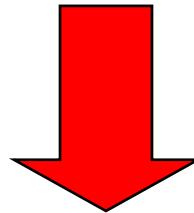
```
begin
    forall Pi , 0≤ i ≤p-1 do
        sumi := 0
        h := i * (n/p)
        for j = h to h+(n/p)-1 do
            sumi := sumi + aj
    endfor
    if P0 then
        for k = 1 to p-1 do
            recv(sumk , Pk)
            sumtot:=sumtot+sumk
        endfor
    else if Pi then
        send(sumi , P0)
    endif
    endif
endforall
end
```



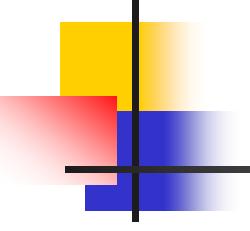
# Uno strumento software per lo sviluppo di algoritmi in ambiente di calcolo **MIMD-Distributed Memory**

**Message Passing Interface**  
**MPI**

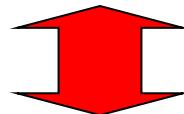
... ogni processore può conoscere i dati nella memoria di un altro processore o far conoscere i propri, attraverso il trasferimento di dati.



*Message Passing :*  
modello per la progettazione  
di software in  
ambiente di Calcolo Parallelo.

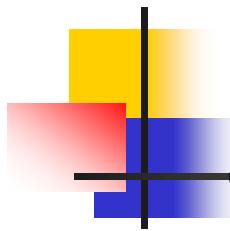


La necessità di **rendere portabile**  
il modello **Message Passing**  
ha condotto alla definizione  
e all'implementazione  
di un **ambiente standard**.



# Message Passing Interface MPI

# Problema



Valutare l'efficienza di un algoritmo  
in ambiente di calcolo parallelo



Cosa si intende per “**EFFICIENZA**” ?

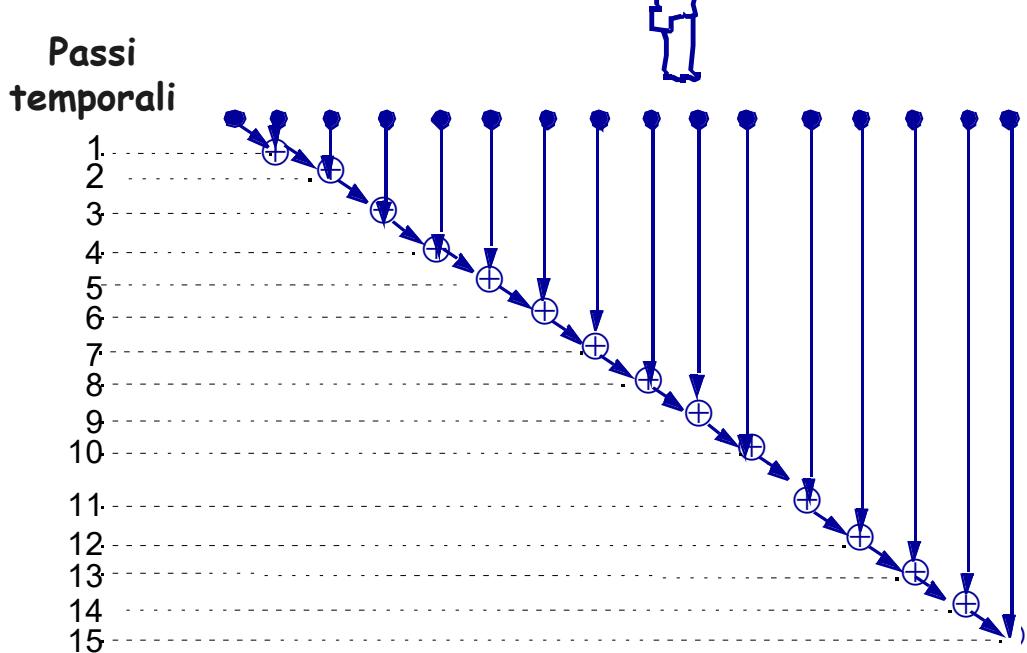
# Efficienza di un algoritmo sequenziale

- COMPLESSITA' di TEMPO  $T(n)$   
Numero di operazioni eseguite dall'algoritmo

- COMPLESSITA' di SPAZIO  $S(n)$   
Numero di variabili utilizzate dall'algoritmo

# Esempio: calcolo della somma di $n=16$ numeri

## ALGORITMO SEQUENZIALE 1 studente



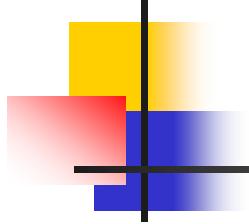
numero di addizioni = 15

passi temporali = 15

complessità di  
tempo

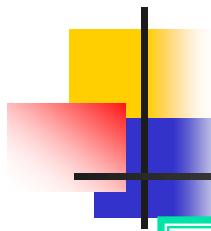
$$T(n)=n-1 \text{ addizioni}$$

# In un algoritmo sequenziale



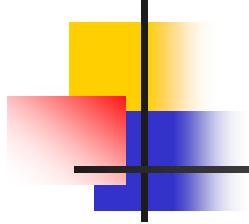
Il numero complessivo di operazioni  
determina anche  
il numero dei passi temporali  
(Il tempo di esecuzione)

# In un software sequenziale



L'efficienza del software  
dipende dal  
tempo di esecuzione  
delle  $T(n)$  operazioni fl.p.

# Domanda



Cosa si intende  
per efficienza di un algoritmo  
in ambiente parallelo?

# Domanda



Si può ancora legare il tempo di esecuzione  
al numero delle operazioni che costituiscono  
l'algoritmo ?

# Esempio: calcolo della somma di $n=16$ numeri

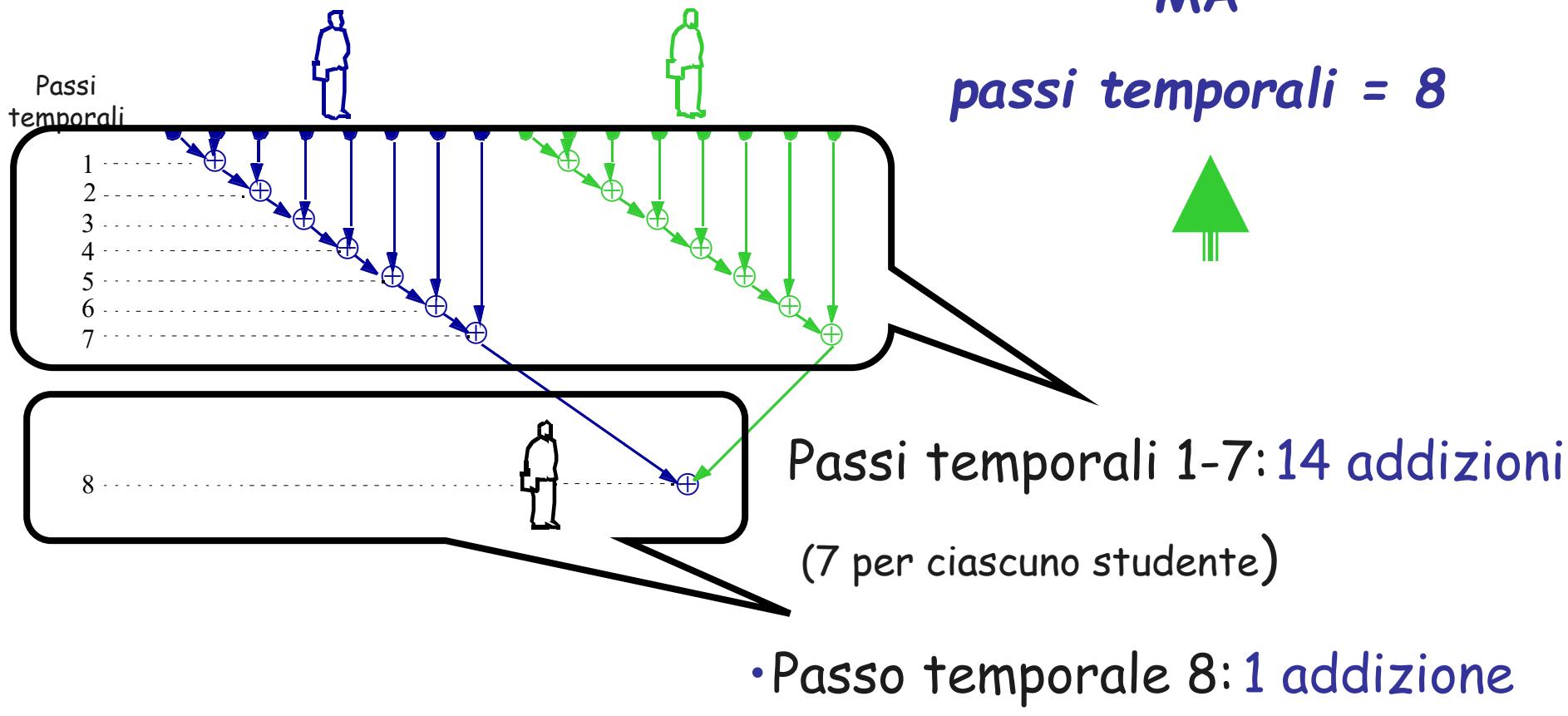
## ALGORITMO PARALLELO

2 studenti

numero di addizioni = 15

MA

passi temporali = 8



# Esempio: calcolo della somma di $n=16$ numeri

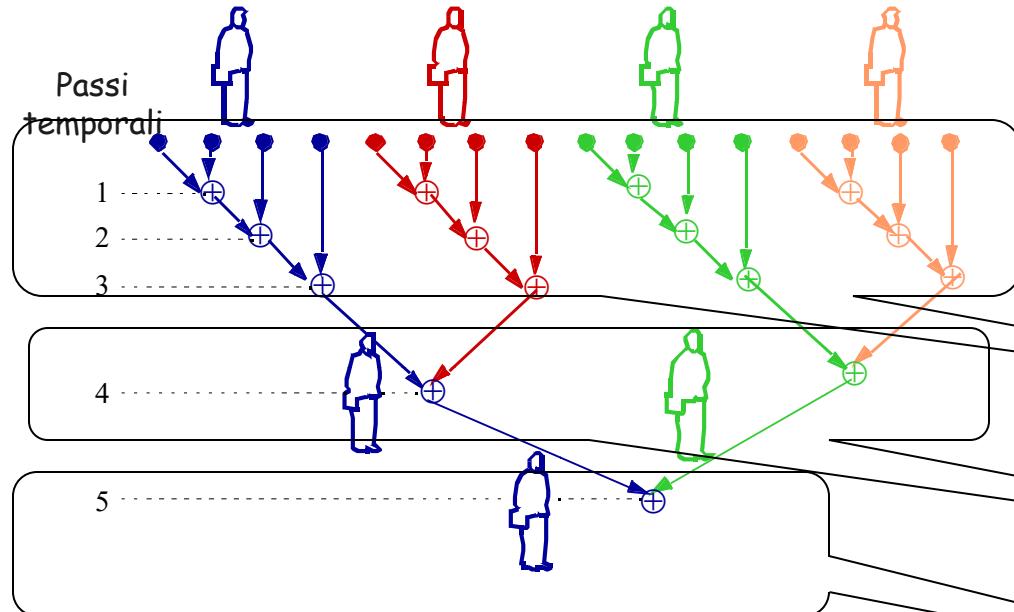
## ALGORITMO PARALLELO

4 studenti

numero di addizioni = 15

MA

passi temporali = 5

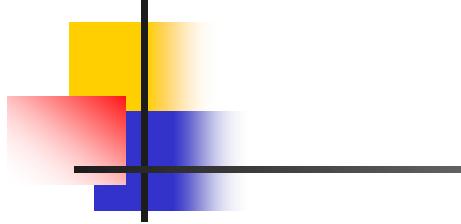


Passi temporali 1-3: 12 addizioni

Passo temporale 4: 2 addizioni

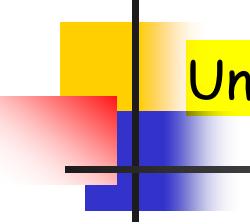
Passo temporale 5: 1 addizione

# Nell'algoritmo parallelo della somma



Il numero delle operazioni  
non è legato  
al numero dei passi temporali

# Infatti ...



Un calcolatore parallelo è in grado di eseguire più operazioni concorrentemente (allo stesso passo temporale)

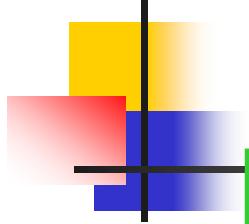


Il tempo di esecuzione non è proporzionale alla complessità di tempo (ovvero non dipende soltanto dal numero di operazioni fl. p. effettuate)



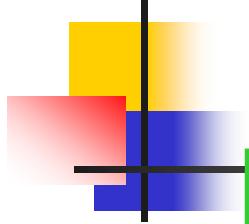
La complessità di tempo non è adatta a misurare l'efficienza di un algoritmo parallelo

... e allora



Che cosa si intende per  
efficienza  
di un algoritmo  
in ambiente parallelo?

... e allora



Che cosa si intende per  
efficienza  
di un algoritmo  
in ambiente parallelo?

Da questo momento ci riferiremo a una macchina che sia in grado di eseguire l'operatore

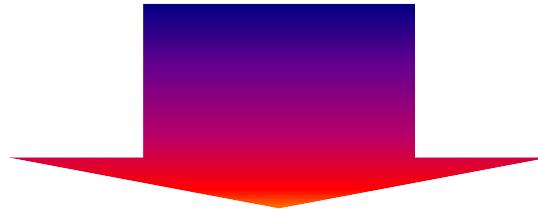


# Esempio:

Se  $p$  indica il numero di processori,  $p=1, 2, 3, \dots$

$T(p)$  = tempo di esecuzione su  $p$  processori

$$p = 2$$



Ci aspettiamo che  $T(1)$  sia il doppio di  $T(2)$ :

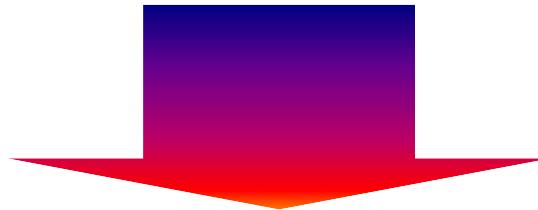
$$\frac{T(1)}{T(2)} = 2$$

# Esempio:

Se  $p$  indica il numero di processori,  $p=1, 2, 3, \dots$

$T(p)$  = tempo di esecuzione su  $p$  processori

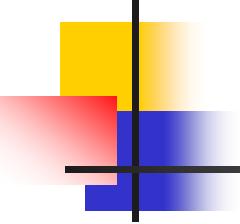
$$p = 4$$



Ci aspettiamo che  $T(1)$  sia il quadruplo di  $T(4)$ :

$$\frac{T(1)}{T(4)} = 4$$

# In generale



Con  $p$  processori ci aspettiamo che

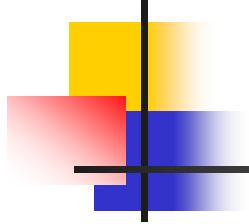
$T(1)$  sia  $p$  volte  $T(p)$

$$\frac{T(1)}{T(p)} = p$$

ovvero

ci aspettiamo di ridurre  $p$  volte il tempo di esecuzione

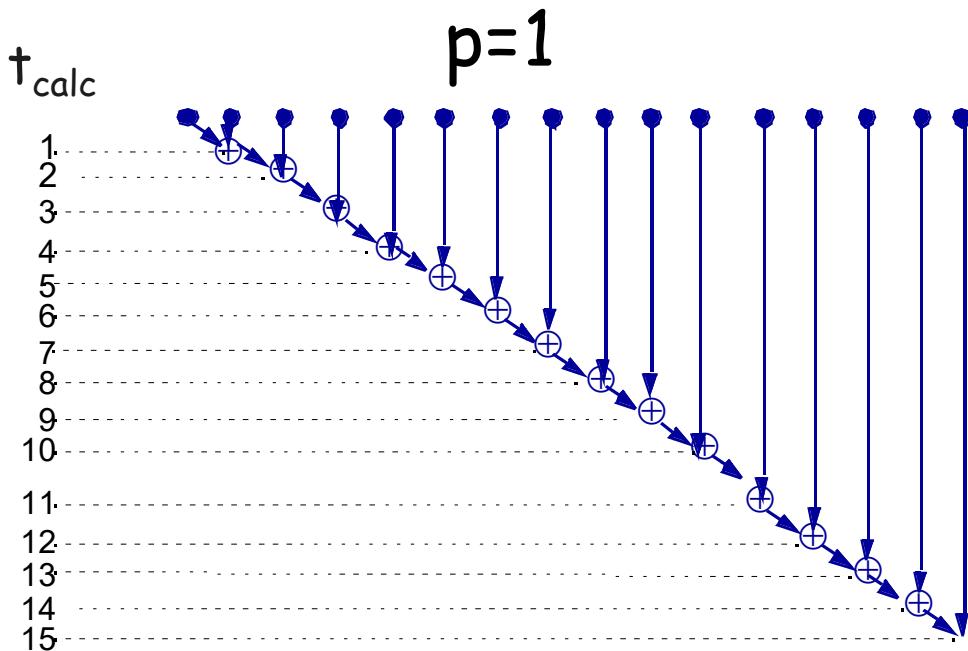
# IDEA



Misuriamo di quanto si riduce il tempo di esecuzione su  $p$  processori rispetto al tempo di esecuzione su 1 processore...

# Esempio: calcolo della somma di $n=16$ numeri

## ALGORITMO SEQUENZIALE



*numero di addizioni = 15*

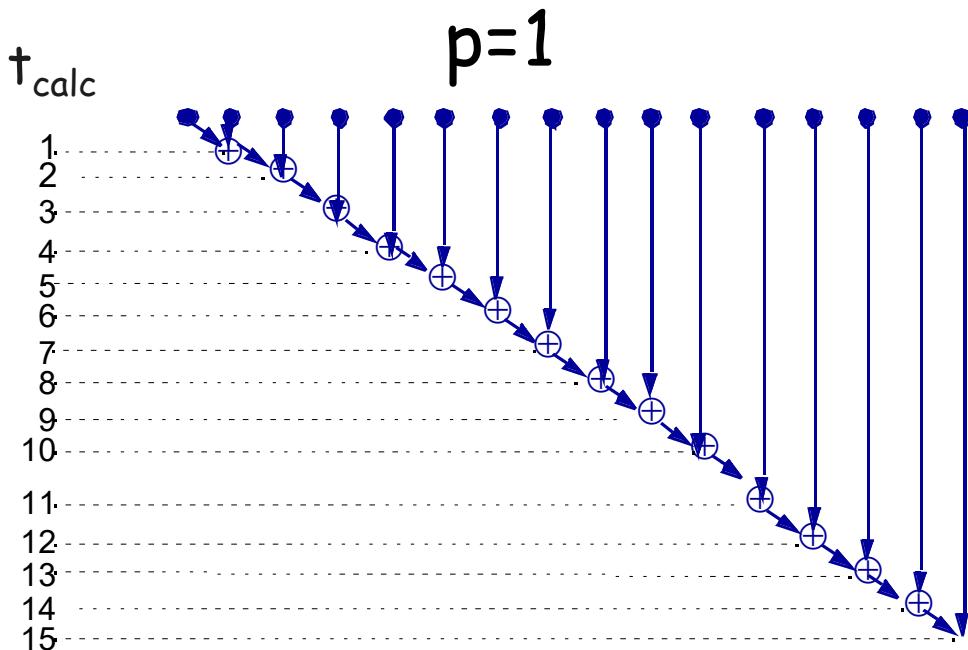
Tempo di  
esecuzione

$$T(1) = 15 + t_{\text{calc}}$$

$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

# Esempio: calcolo della somma di $n=16$ numeri

## ALGORITMO SEQUENZIALE



*numero di addizioni = 15*

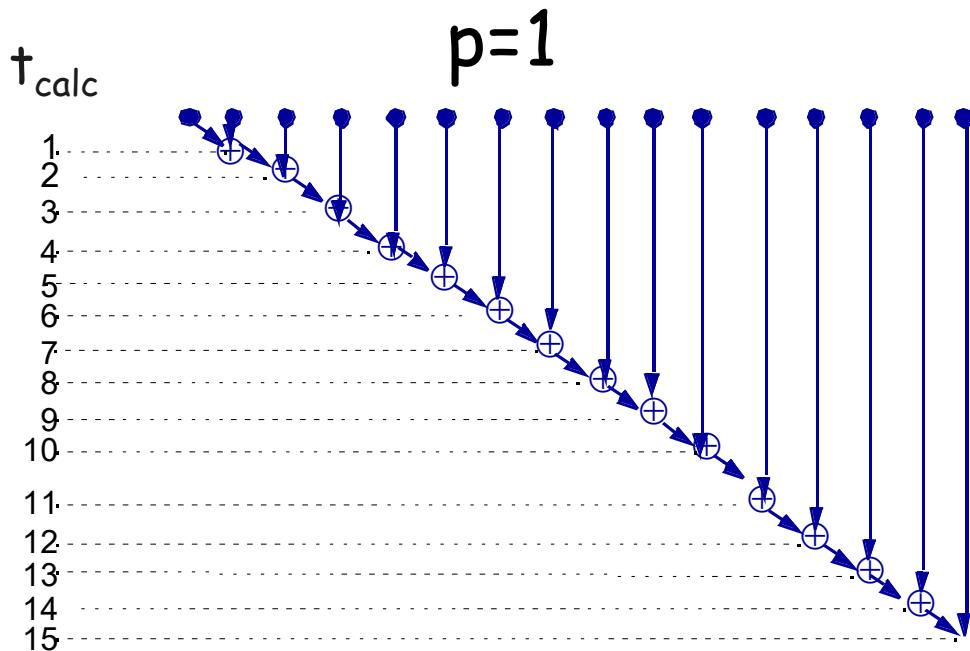
Tempo di  
esecuzione

$$T(1)=15 + t_{\text{calc}}$$

$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

# Esempio: calcolo della somma di $n=16$ numeri

## ALGORITMO SEQUENZIALE



$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

In questo caso, dopo il primo, ogni operatore dipende da tutti i precedenti.

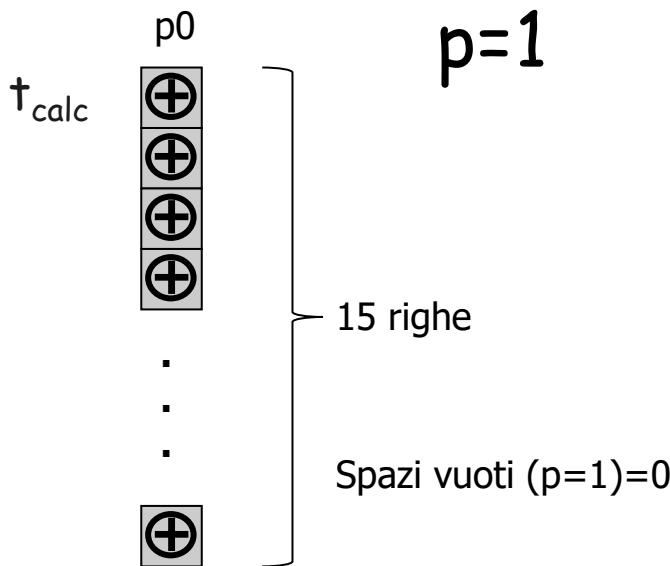
*numero di addizioni = 15*

Tempo di  
esecuzione

$$T(1)=15 + t_{\text{calc}}$$

# Esempio: calcolo della somma di $n=16$ numeri

## ALGORITMO SEQUENZIALE



In questo caso, dopo il primo, ogni operatore dipende da tutti i precedenti.  
Disegniamo uno sotto l'altro gli operatori dipendenti

*numero di addizioni = 15*

$$\text{righe}(1) = 15$$

Tempo di esecuzione

$$T(1)=15 + t_{\text{calc}}$$

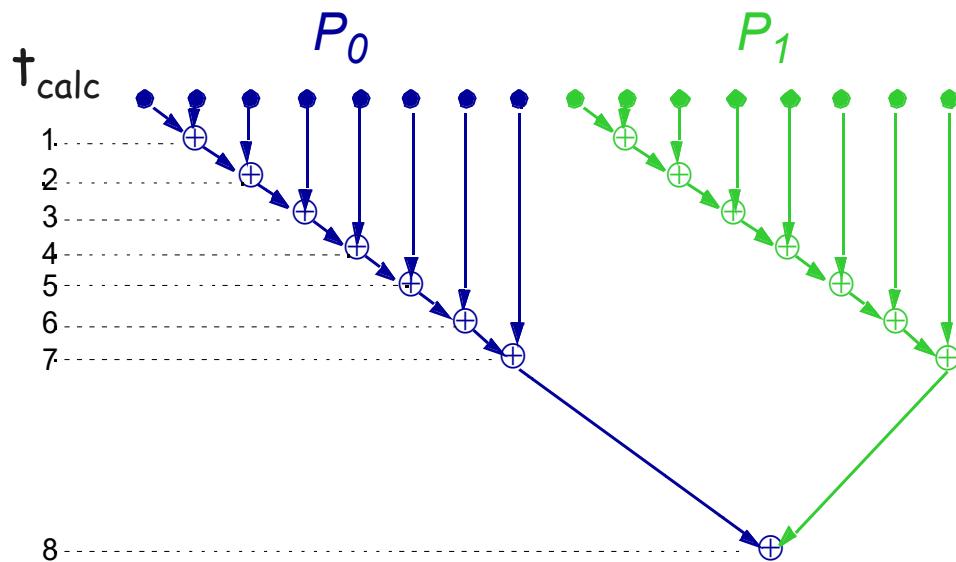
$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

$$\text{spazi_vuoti}(1) = 0$$

# Esempio: calcolo della somma di $n=16$ numeri

## ALGORITMO PARALLELO

$p=2$



In questo caso, per 7 passi due operatori possono essere eseguiti concorrentemente: sono INDIPENDENTI tra loro

*numero di addizioni = 15*

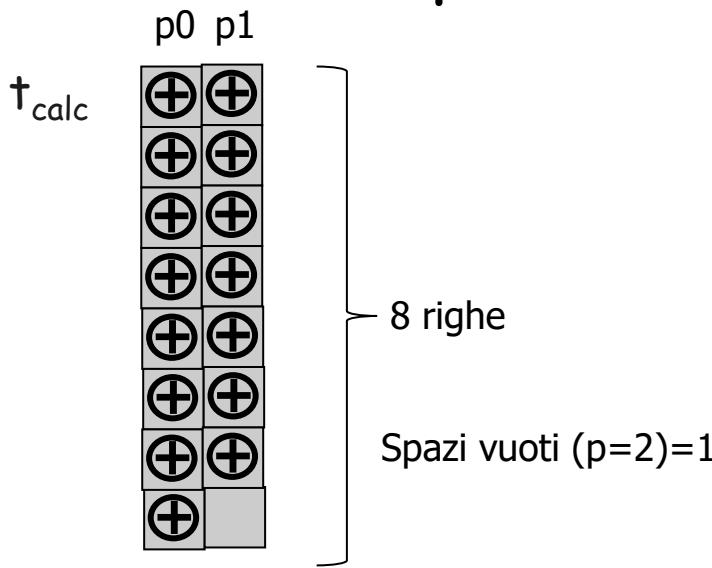
$$T(2)=8 + t_{\text{calc}}$$

$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

# Esempio: calcolo della somma di $n=16$ numeri

## ALGORITMO PARALLELO

$p=2$



Disegniamo uno a fianco all'altro gli operatori che sono INDIPENDENTI tra loro, cioè che possono essere eseguiti concorrentemente

*numero di addizioni = 15*

$$\text{righe}(2) = 8$$

$$T(2)=8 + t_{\text{calc}}$$

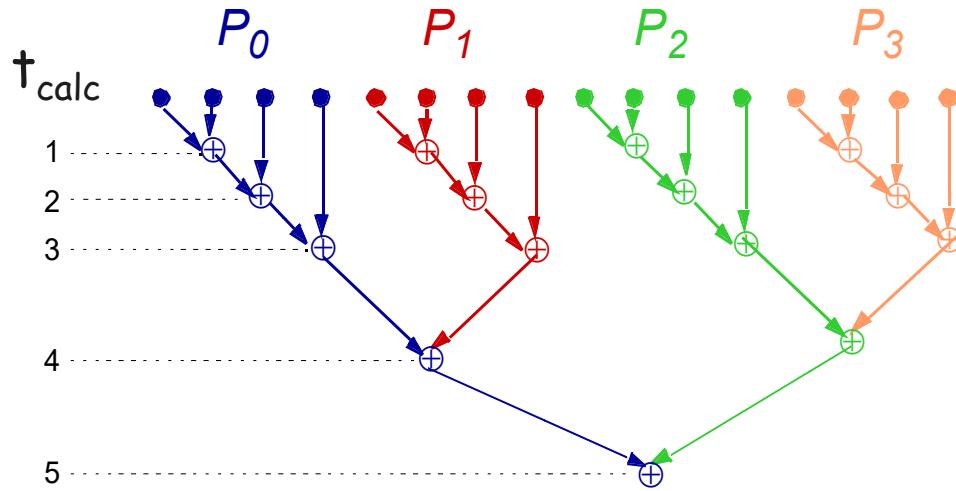
$$\text{spazi_vuoti}(2) = 1$$

$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

# Esempio: calcolo della somma di $n=16$ numeri

## ALGORITMO PARALLELO

$p=4$



*numero di addizioni = 15*

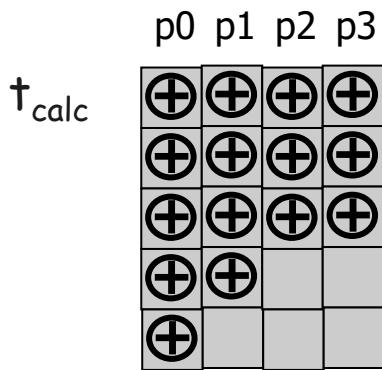
$$T(4) = 5 + t_{\text{calc}}$$

$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

# Esempio: calcolo della somma di $n=16$ numeri

## ALGORITMO PARALLELO

$p=4$



*numero di addizioni = 15*

$$\text{righe}(4) = 5$$

$$T(4)=5 + t_{\text{calc}}$$

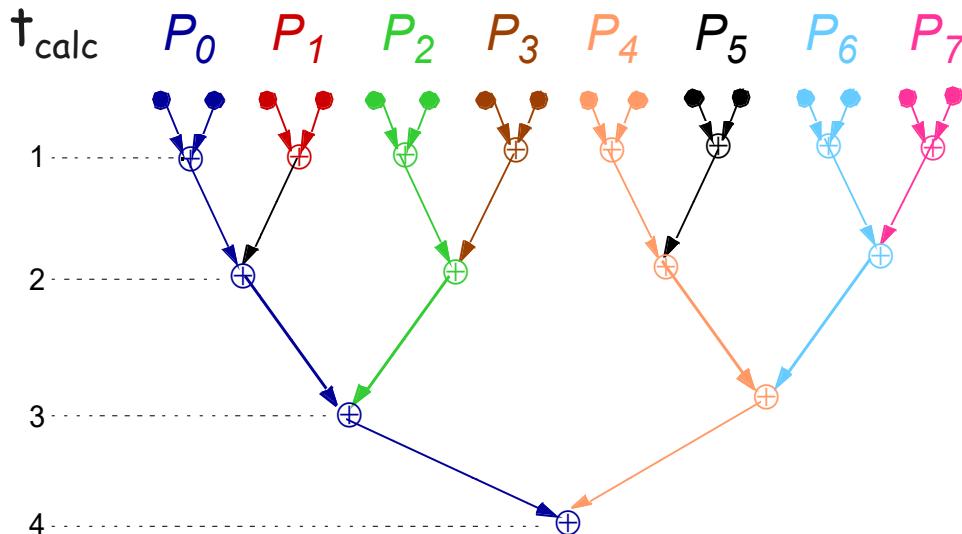
$$\text{spazi_vuoti}(4) = 5$$

$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

# Esempio: calcolo della somma di $n=16$ numeri

## ALGORITMO PARALLELO

$p=8$



*numero di addizioni = 15*

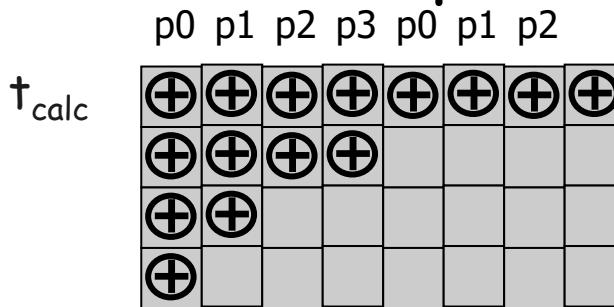
$$T(8)=4 \cdot t_{\text{calc}}$$

$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

# Esempio: calcolo della somma di $n=16$ numeri

## ALGORITMO PARALLELO

$p=8$



4 righe

*numero di addizioni = 15*

$$\text{righe}(8) = 4$$

Spazi vuoti ( $p=8$ )=17

$$T(8)=4 \cdot t_{\text{calc}}$$

$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

$$\text{spazi_vuoti}(8) = 17$$

# In sintesi...

In generale possiamo dire che per qualsiasi algoritmo parallelo descritto attraverso una tabella di operatori come quelle viste

$$T(p) = \text{righe}(p) \cdot t_{\text{calc}}$$

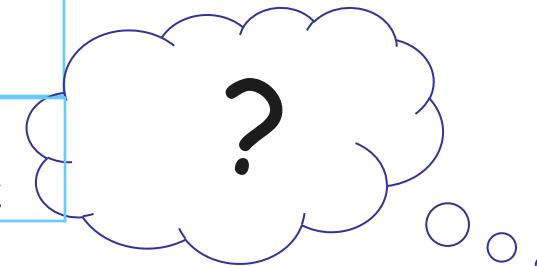
p	$T_p$
1	$15t_{\text{calc}}$
2	$8t_{\text{calc}}$
4	$5t_{\text{calc}}$
8	$4t_{\text{calc}}$
$2^k$	? $t_{\text{calc}}$

# In sintesi...

In generale possiamo dire che per qualsiasi algoritmo parallelo descritto attraverso una tabella di operatori come quelle viste

$$T(p) = \text{righe}(p) t_{\text{calc}}$$

p	$T_p$
1	$15t_{\text{calc}}$
2	$8t_{\text{calc}}$
4	$5t_{\text{calc}}$
8	$4t_{\text{calc}}$
$2^k$	$? t_{\text{calc}}$



In generale quanto vale  $T(p)$  per l'algoritmo della somma?

# In generale: calcolo di $T(p)$

ALGORITMO PARALLELO della somma di  $n$  numeri  
posto  $p=2^k$  processori

$$p=1 \quad T(1)=15 + t_{\text{calc}}$$

$$p=2 \quad T(2)=8 + t_{\text{calc}} = (7+1) + t_{\text{calc}}$$

$$p=4 \quad T(4)=5 + t_{\text{calc}} = (3+2) + t_{\text{calc}}$$

$$p=8 \quad T(8)=4 + t_{\text{calc}} = (1+3) + t_{\text{calc}}$$

.....

$$T(p) = \left( \frac{n}{p} - 1 + \log_2 p \right) t_{\text{calc}}$$

$n = 16$

$t_{\text{calc}}$  = tempo di esecuzione di 1 addizione

# Domanda...

$p$	$T_p$
1	$15t_{\text{calc}}$
2	$8t_{\text{calc}}$
4	$5t_{\text{calc}}$
8	$4t_{\text{calc}}$

Qual è l'algoritmo che impiega meno tempo?



Quanto è più veloce di quello sequenziale?

# Esempio: calcolo della somma di $n=16$ numeri

$p$	$T(p)$	$\frac{T(1)}{T(p)}$
1	$15t_{\text{calc}}$	1.00
2	$8t_{\text{calc}}$	1.88
4	$5t_{\text{calc}}$	3.00
8	$4t_{\text{calc}}$	3.75

Maggiore riduzione del tempo ovvero maggiore aumento della velocità

L'algoritmo su 8 processori è il più veloce  
E' più veloce di 3.75 volte di quello su 1 processore

# Speed-up

Si definisce il rapporto  $T(1)$  su  $T(p)$

$$S(p) = \frac{T(1)}{T(p)}$$

Lo speed up misura la riduzione del tempo di esecuzione rispetto all'algoritmo su 1 processore

$$S(p) < p$$

$$\left. \begin{array}{l} \text{SPEEDUP IDEALE} \\ S^{\text{ideale}}(p) = p \end{array} \right\}$$

# Speed-up

Si definisce il rapporto  $T(1)$  su  $T(p)$

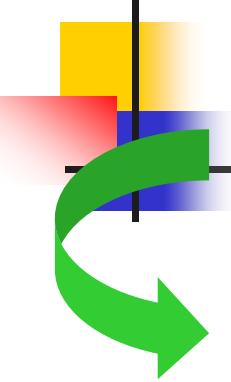
$$S(p) = \frac{\text{righe}(1)}{\text{righe}(p)}$$

Lo speed up misura la riduzione del tempo di esecuzione rispetto all'algoritmo su 1 processore

$$S(p) < p$$

$$\left. \begin{array}{l} \text{SPEEDUP IDEALE} \\ S^{\text{ideale}}(p) = p \end{array} \right\}$$

# Osservazione


$$S^{\text{ideale}}(p) = \frac{T(1)}{T(p)} = p$$

$$O_h = pT(p) - T(1)$$

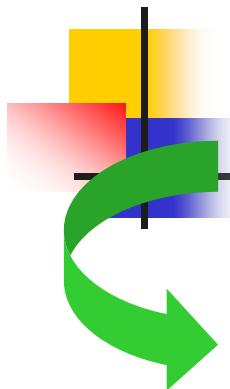
$$T(p) = (O_h + T(1)) / p$$

OVERHEAD totale

$$S(p) = \frac{T(1)}{T(p)} = \frac{T(1)}{(O_h + T(1))/p} = \frac{pT(1)}{O_h + T(1)} = \frac{p}{\frac{O_h}{T(1)} + 1}$$

L'OVERHEAD totale misura  
quanto lo speed up differisce da quello ideale

# Osservazione


$$S^{\text{ideale}}(p) = \frac{T(1)}{T(p)} = p$$

$$O_h = pT(p) - T(1)$$

OVERHEAD totale

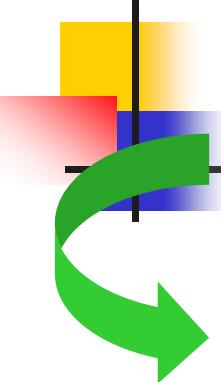
$$O_h = p * \text{righe}(p) * t_{\text{calc}} - \text{righe}(1) * t_{\text{calc}}$$

$$O_h = p * \text{righe}(p) - \text{righe}(1)$$

$$O_h = \text{spazi\_vuoti}(p)$$

L'OVERHEAD totale misura  
quanto lo speed up differisce da quello ideale

# Osservazione


$$S^{\text{ideale}}(p) = \frac{T(1)}{T(p)} = p$$

$$O_h = pT(p) - T(1)$$

OVERHEAD totale

$$O_h = p * \text{righe}(p) * t_{\text{calc}} - \text{righe}(1) * t_{\text{calc}}$$

$$O_h = p * \text{righe}(p) - \text{righe}(1)$$

$$O_h = \text{spazi\_vuoti}(p)$$

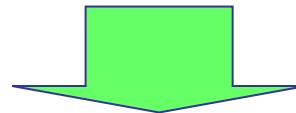
La rappresentazione degli operatori in tabella dà immediate informazioni sul tempo d'esecuzione e sull'overhead totale dell'algoritmo!

L'OVERHEAD totale misura  
quanto lo speed up differisce da quello ideale

## L'overhead nell'algoritmo parallelo della somma

$$T(1) = n - 1$$

$$T(p) = n/p - 1 + \log_2 p$$

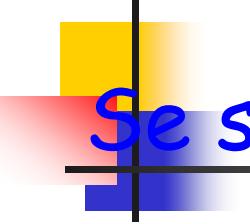


$$\begin{aligned} O_h &= p T(p) - T(1) = p(n/p + \log_2 p) - (n-1) = \\ &= n + p \log_2 p - n + 1 = O(p \log_2 p) \end{aligned}$$

p	$O_h$
2	2
4	8
8	24
$2^k$	$p \log_2 p$

Al crescere di p  
l'overhead aumenta!

# Quindi

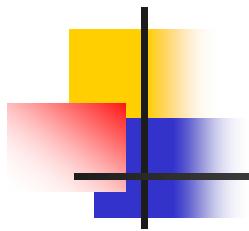


Se si vuole calcolare la somma di 16 numeri  
nel minor tempo possibile

l'algoritmo su 8 processori è da preferire

Infatti, aumentando il numero di processori  
si riduce

il tempo impiegato per eseguire le operazioni  
richieste



MA....

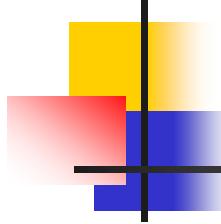
# Esempio: calcolo della somma di $n=16$ numeri

p	Speed-up ottenuto	Speed-up ideale
2	1.88	2
4	3.00	4
8	3.75	8

Lo speed-up su 8 processori è il maggiore  
MA

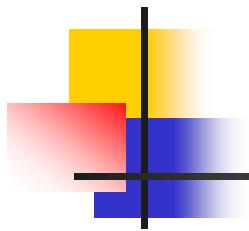
Lo speed-up su 2 processori è "il più vicino"  
allo speed-up ideale...

# Cioè



Ho utilizzato 8 processori per  
avere un incremento  
di appena 4 volte

## In altre parole



... lo **speed up** non basta a

fornire informazioni sull'efficienza

dell'algoritmo parallelo!

... e allora ?

# Esempio: calcolo della somma di $n=16$ numeri

... se si rapporta lo speed-up al numero di processori...

$p$	$S(p)$	$\frac{S(p)}{p}$
2	1.88	0.94
4	3.00	0.75
8	3.75	0.47

Rapporto più grande



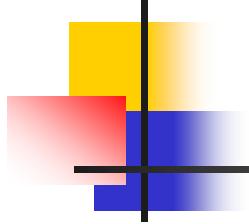
maggior sfruttamento dei  
processori per  $p=2$

## In altre parole

L'utilizzo di un maggior numero di processori NON è sempre una garanzia di sviluppo di algoritmi paralleli "efficaci"

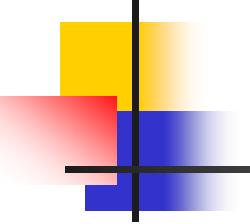
OVVERO

Di algoritmi che sfruttano  
tutte le risorse della macchina parallela !



Come misurare se e quanto  
è stata sfruttato il calcolatore parallelo ?

# Efficienza



Si definisce il rapporto  $E_p$  su  $p$

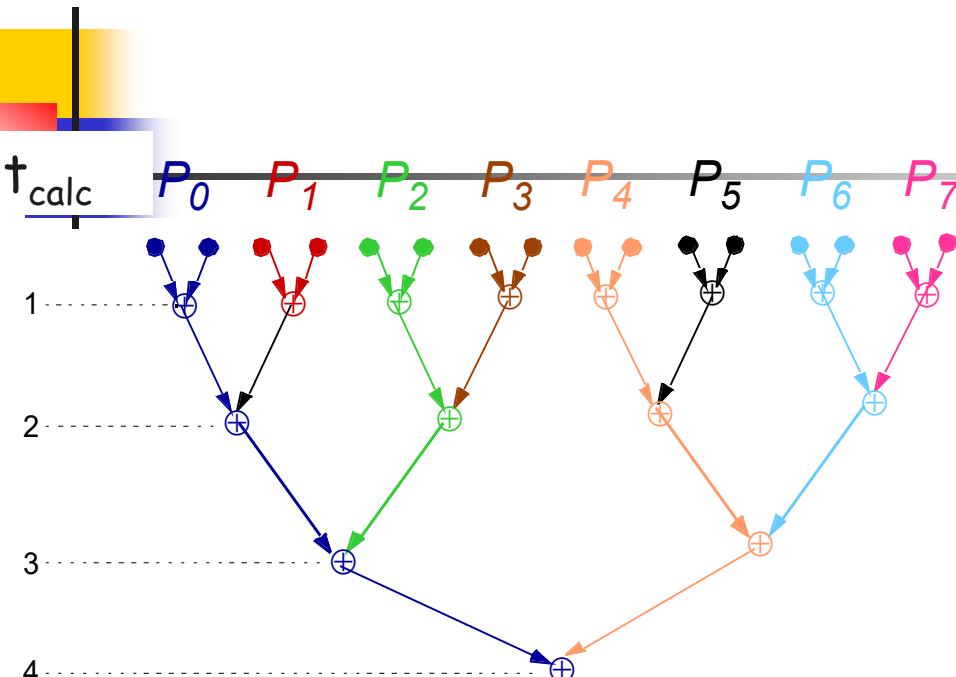
$$E(p) = \frac{S(p)}{p}$$

misura quanto l'algoritmo sfrutta il parallelismo del calcolatore

## EFFICIENZA IDEALE

$$E^{\text{ideale}}(p) = \frac{S^{\text{ideale}}(p)}{p} = 1$$

# Infatti, per la somma con p=8



$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

al I passo:

lavorano  $p=8$  processori

al II passo:

lavorano  $p/2=4$  processori

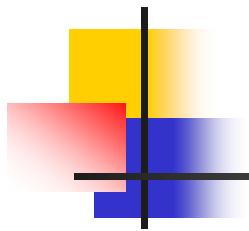
al III passo:

lavorano  $p/4=2$  processori

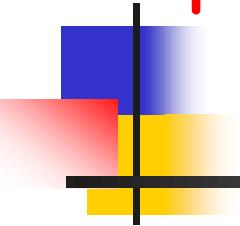
al IV passo:

lavora  $p/8=1$  processore!

Ad ogni passo si dimezza il numero  
di processori attivi



# Fine Lezione



# Parallel and Distributed Computing

a.a. 2021-2022

---

## Somma di N numeri

Su architettura MIMD-DM

Prof. Giuliano Laccetti

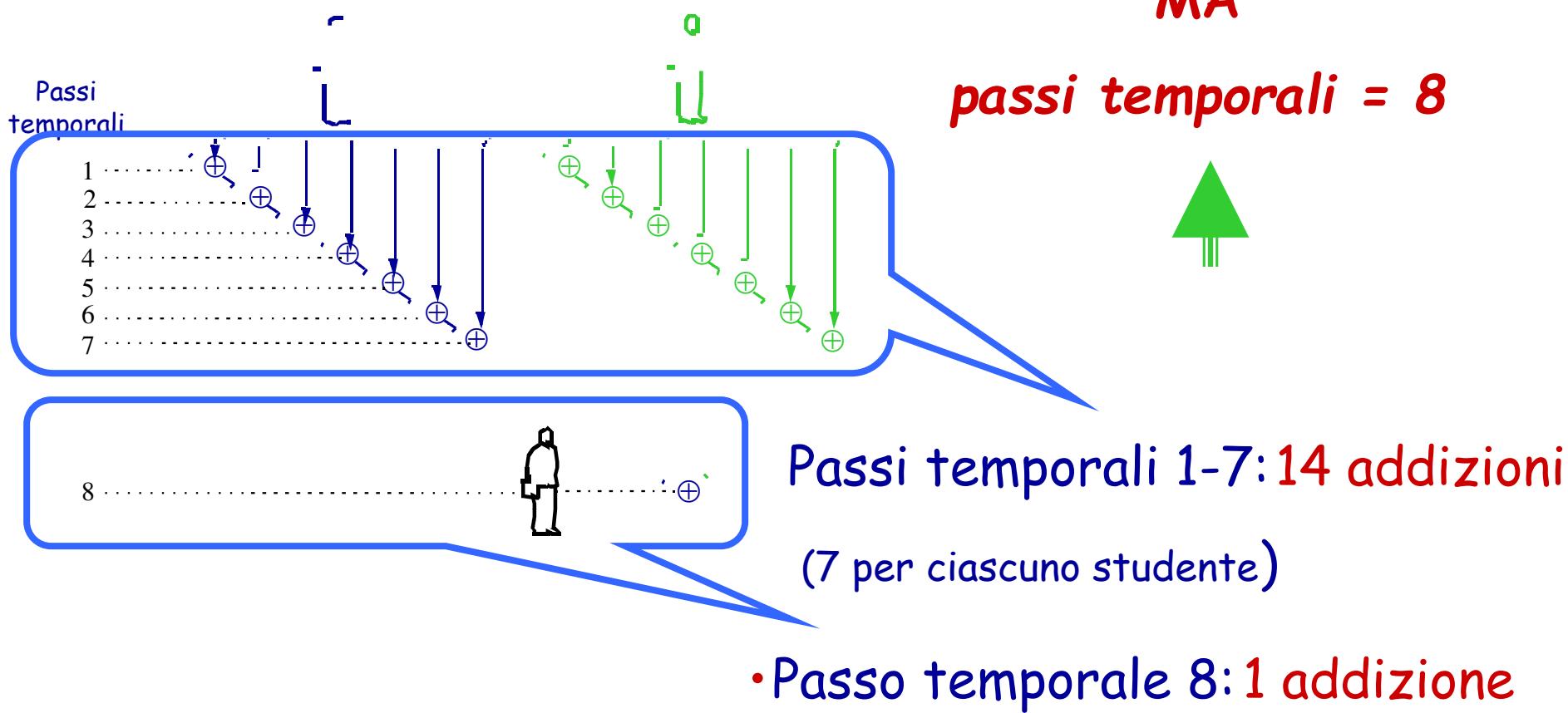
# Esempio: calcolo della somma di $n=16$ numeri

ALGORITMO PARALLELO  
2 studenti

numero di addizioni = 15

MA

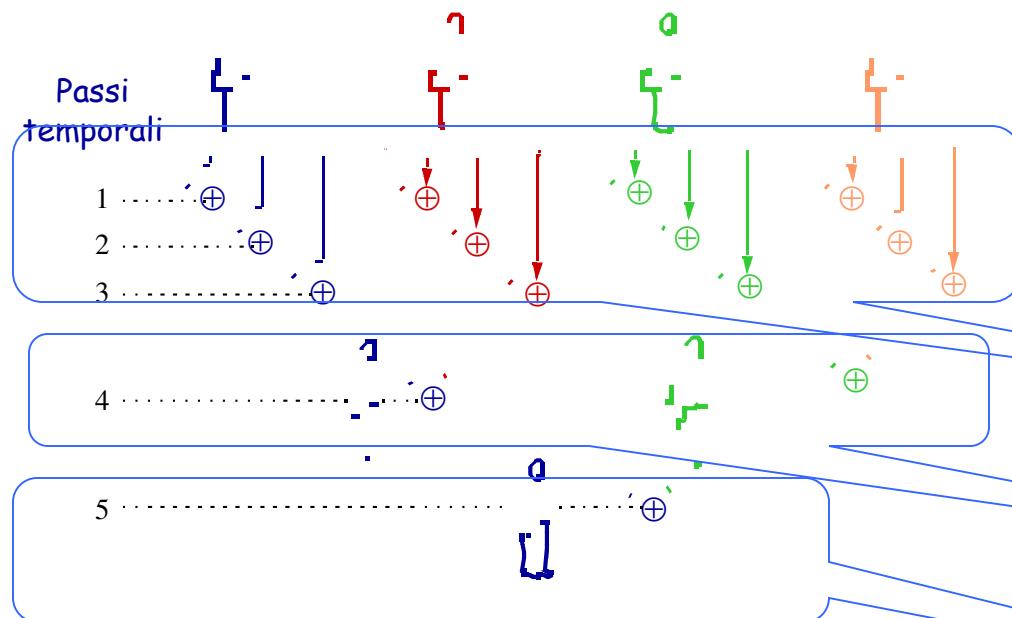
passi temporali = 8



# Esempio: calcolo della somma di $n=16$ numeri

ALGORITMO PARALLELO  
4 studenti

numero di addizioni = 15



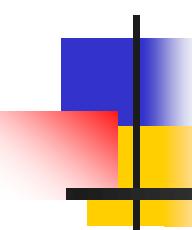
MA  
passi temporali = 5



Passi temporali 1-3: 12 addizioni

Passo temporale 4: 2 addizioni

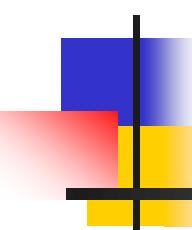
Passo temporale 5: 1 addizione



... e allora

---

Che cosa si intende per  
efficienza  
di un algoritmo  
in ambiente parallelo?



... e allora

---

*Che cosa si intende per  
efficienza  
di un algoritmo  
in ambiente parallelo?*

Da questo momento ci riferiremo a una macchina che sia in grado di eseguire l'operatore

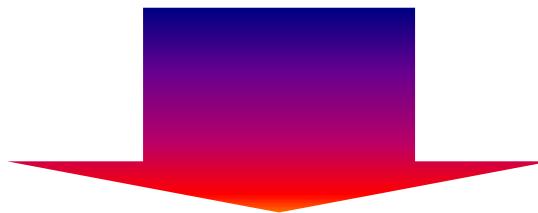


# Esempio:

Se  $p$  indica il numero di processori,  $p=1, 2, 3, \dots$

$T(p)$  = tempo di esecuzione su  $p$  processori

$$p = 2$$



Ci aspettiamo che  $T(1)$  sia il doppio di  $T(2)$ :

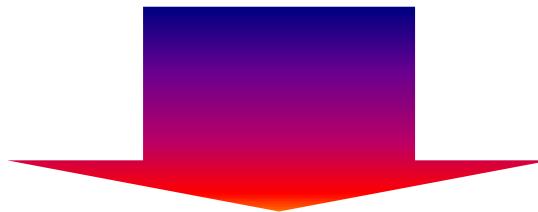
$$\frac{T(1)}{T(2)} = 2$$

# Esempio:

Se  $p$  indica il numero di processori,  $p=1, 2, 3, \dots$

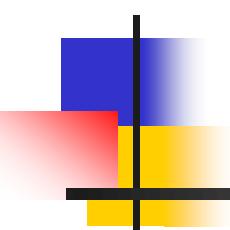
$T(p)$  = tempo di esecuzione su  $p$  processori

$$p = 4$$



Ci aspettiamo che  $T(1)$  sia il quadruplo di  $T(4)$ :

$$\frac{T(1)}{T(4)} = 4$$



# In generale

Con  $p$  processori ci aspettiamo che

$T(1)$  sia  $p$  volte  $T(p)$

$$\frac{T(1)}{T(p)} = p$$

ovvero

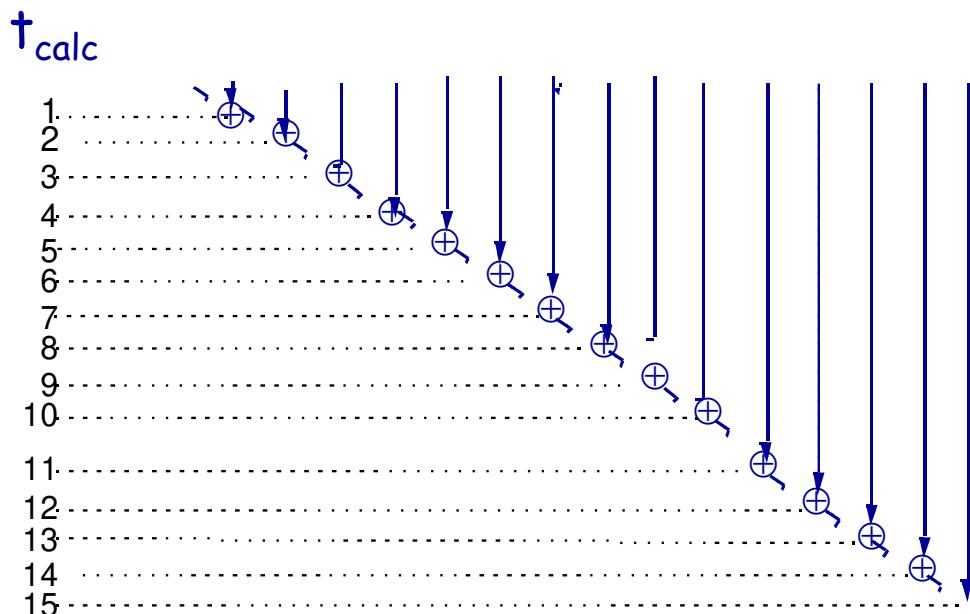
ci aspettiamo di **ridurre**  $p$  volte il tempo di esecuzione

Misuriamo di quanto si riduce il tempo di esecuzione su  $p$  processori rispetto al tempo di esecuzione su 1 processore...

# Esempio: calcolo della somma di $n=16$ numeri

## ALGORITMO SEQUENZIALE

$p=1$



numero di addizioni = 15

Tempo di esecuzione

$$T(1)=15 + t_{\text{calc}}$$

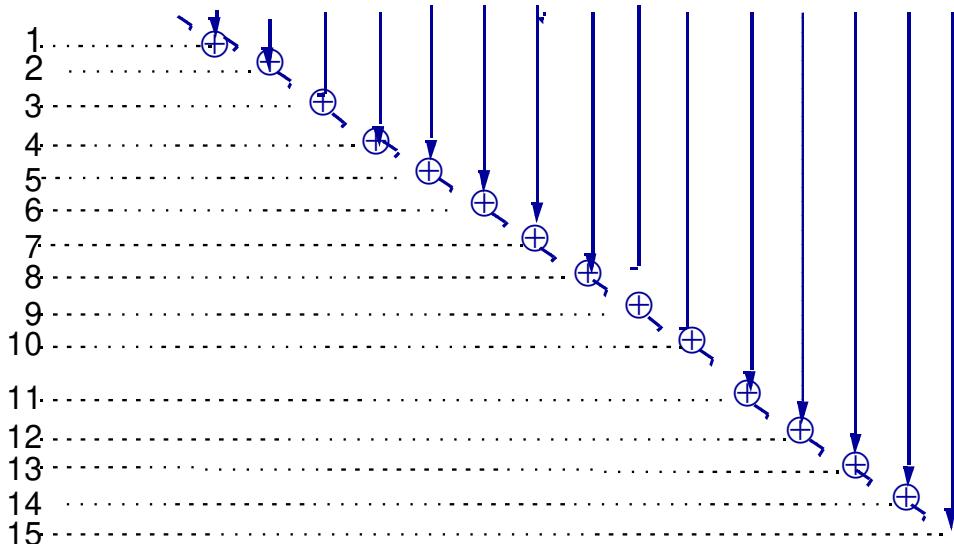
$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

# Esempio: calcolo della somma di $n=16$ numeri

## ALGORITMO SEQUENZIALE

$p=1$

$t_{\text{calc}}$



$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

Diciamo che un operatore DIPENDE da un altro se la sua esecuzione è subordinata all'esecuzione dell'altro.

*numero di addizioni = 15*

Tempo di esecuzione

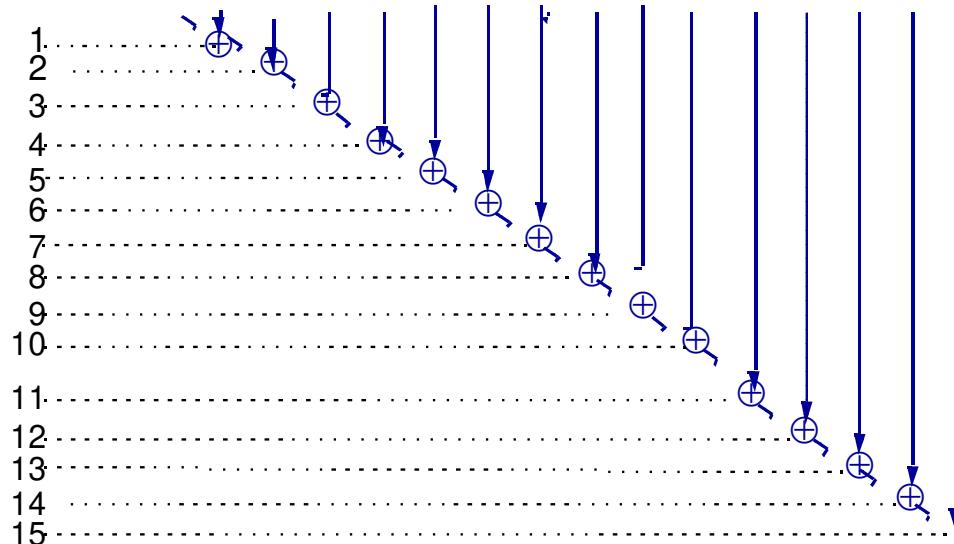
$$T(1) = 15 + t_{\text{calc}}$$

# Esempio: calcolo della somma di $n=16$ numeri

## ALGORITMO SEQUENZIALE

$p=1$

$t_{\text{calc}}$



In questo caso, dopo il primo, ogni operatore dipende da tutti i precedenti.

*numero di addizioni = 15*

Tempo di esecuzione

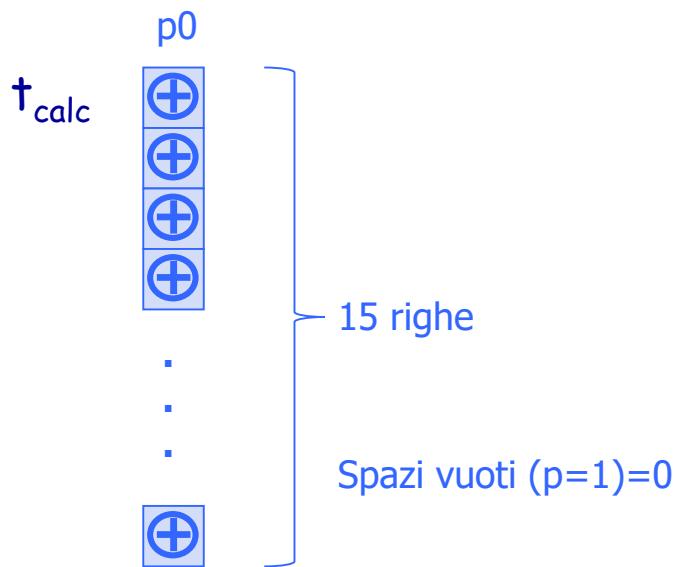
$$T(1)=15 + t_{\text{calc}}$$

$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

# Esempio: calcolo della somma di $n=16$ numeri

## ALGORITMO SEQUENZIALE

$p=1$



In questo caso, dopo il primo,  
ogni operatore dipende da tutti i  
precedenti.  
Disegniamo uno sotto l'altro gli  
operatori dipendenti

**numero di addizioni = 15**

**righe(1) = 15**

Tempo di  
esecuzione

$$T(1)=15 + t_{\text{calc}}$$

$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

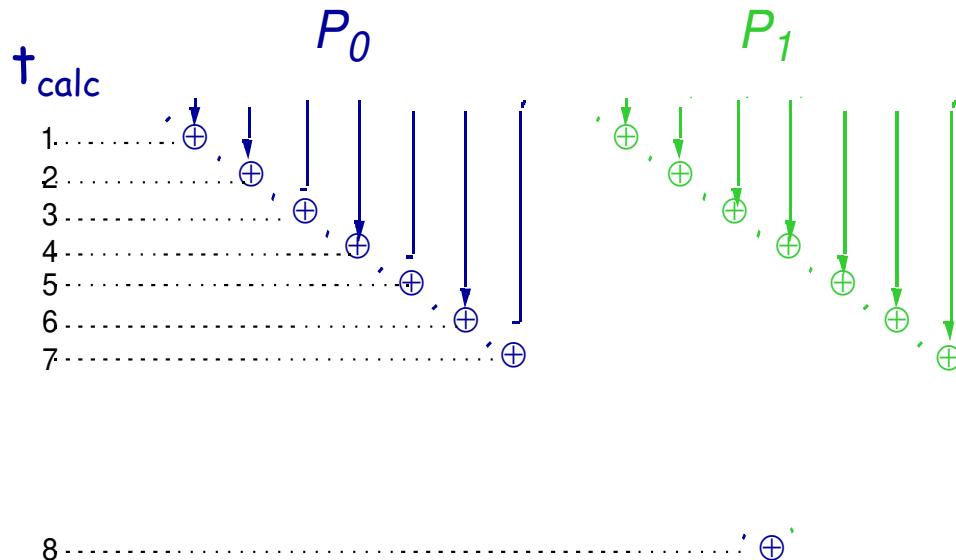
$$\text{spazi_vuoti}(1) = 0$$

# Esempio: calcolo della somma di $n=16$ numeri

## ALGORITMO PARALLELO

$p=2$

In questo caso, per 7 passi due operatori possono essere eseguiti concorrentemente: sono INDIPENDENTI tra loro



*numero di addizioni = 15*

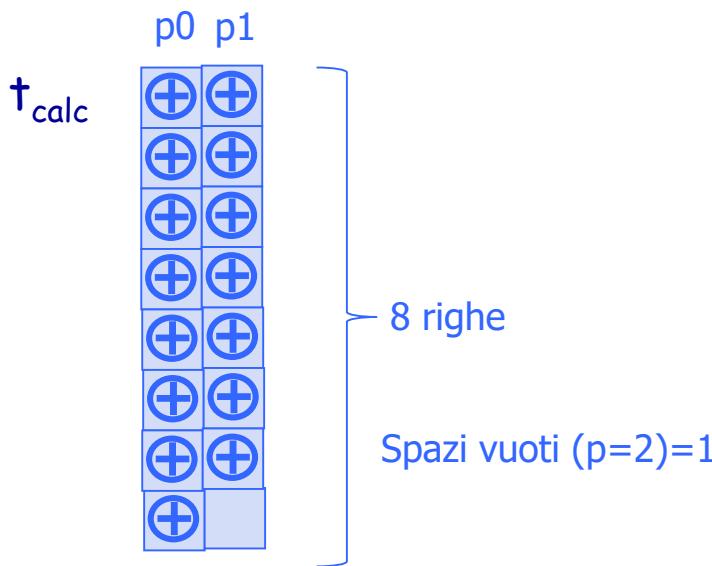
$$T(2) = 8 \cdot t_{\text{calc}}$$

$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

# Esempio: calcolo della somma di $n=16$ numeri

## ALGORITMO PARALLELO

$p=2$



Disegniamo uno a fianco all'altro gli operatori che sono INDIPENDENTI tra loro, cioè che possono essere eseguiti concorrentemente

*numero di addizioni = 15*

$$\text{righe}(2) = 8$$

$$T(2)=8 + t_{\text{calc}}$$

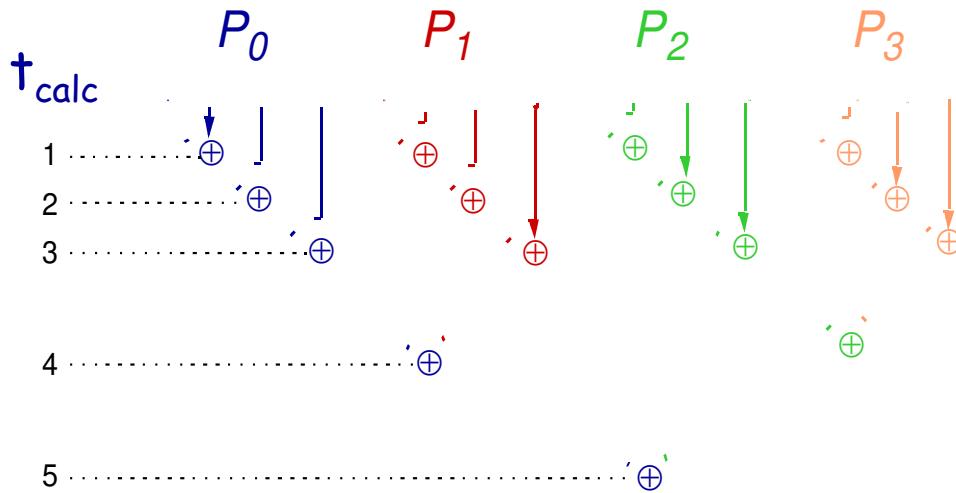
$$\text{spazi_vuoti}(2) = 1$$

$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

# Esempio: calcolo della somma di $n=16$ numeri

## ALGORITMO PARALLELO

$p=4$



*numero di addizioni = 15*

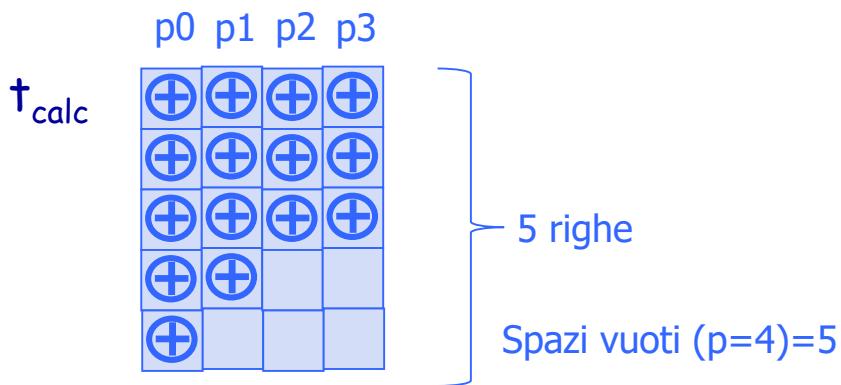
$$T(4) = 5 + t_{\text{calc}}$$

$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

# Esempio: calcolo della somma di $n=16$ numeri

## ALGORITMO PARALLELO

$p=4$



*numero di addizioni = 15*

$$\text{righe}(4) = 5$$

$$T(4)=5 + t_{\text{calc}}$$

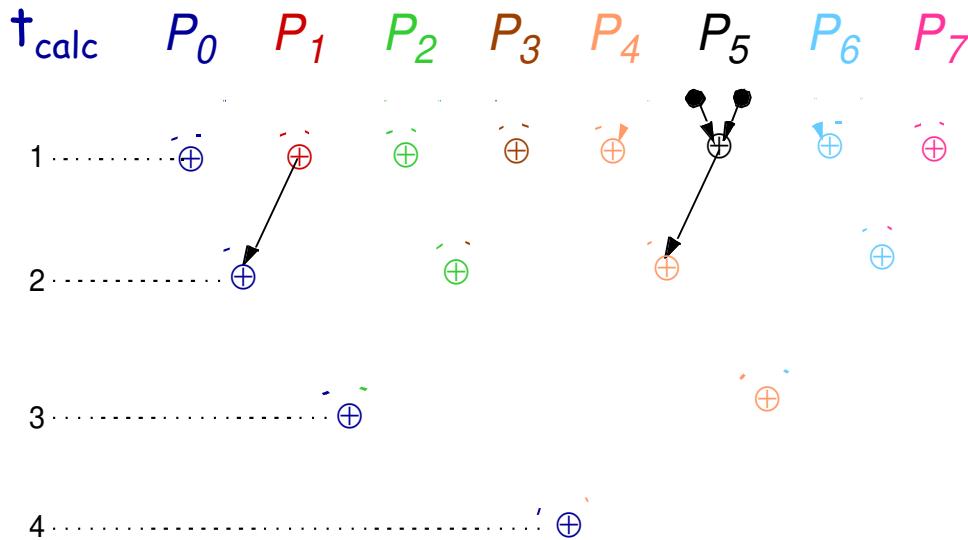
$$\text{spazi_vuoti}(4) = 5$$

$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

# Esempio: calcolo della somma di $n=16$ numeri

## ALGORITMO PARALLELO

$p=8$



*numero di addizioni = 15*

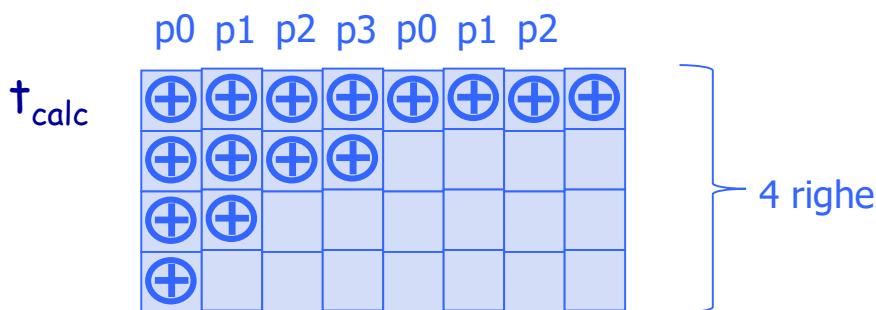
$$T(8)=4 \cdot t_{\text{calc}}$$

$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

# Esempio: calcolo della somma di $n=16$ numeri

## ALGORITMO PARALLELO

$p=8$



*numero di addizioni = 15*

*righe(8) = 4*

Spazi vuoti ( $p=8$ )=17

$$T(8)=4 + t_{\text{calc}}$$

$$\text{spazi_vuoti}(8) = 17$$

$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

# In sintesi...

In generale possiamo dire che per qualsiasi algoritmo parallelo descritto attraverso una tabella di operatori come quelle viste

$$T(p) = \text{righe}(p) \cdot t_{\text{calc}}$$

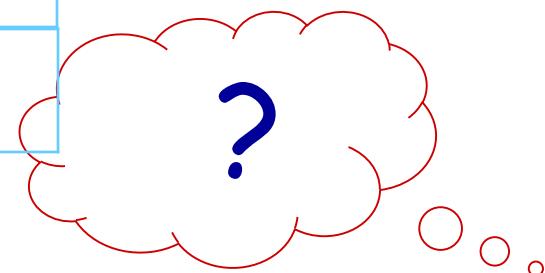
p	$T_p$
1	$15t_{\text{calc}}$
2	$8t_{\text{calc}}$
4	$5t_{\text{calc}}$
8	$4t_{\text{calc}}$
$2^k$	? $t_{\text{calc}}$

# In sintesi...

In generale possiamo dire che per qualsiasi algoritmo parallelo descritto attraverso una tabella di operatori come quelle viste

$$T(p) = \text{righe}(p) t_{\text{calc}}$$

p	$T_p$
1	$15t_{\text{calc}}$
2	$8t_{\text{calc}}$
4	$5t_{\text{calc}}$
8	$4t_{\text{calc}}$
$2^k$	? $t_{\text{calc}}$



In generale quanto vale  $T(p)$  per l'algoritmo della somma?

# In generale: calcolo di $T(p)$

ALGORITMO PARALLELO della somma di  $n$  numeri  
posto  $p=2^k$  processori

$$p=1 \quad T(1)=15 + t_{\text{calc}}$$

$$p=2 \quad T(2)=8 + t_{\text{calc}} = (7+1) + t_{\text{calc}}$$

$$p=4 \quad T(4)=5 + t_{\text{calc}} = (3+2) + t_{\text{calc}}$$

$$p=8 \quad T(8)=4 + t_{\text{calc}} = (1+3) + t_{\text{calc}}$$

.....

$$n = 16$$

$$T(p) = \left( \frac{n}{p} - 1 + \log_2 p \right) t_{\text{calc}}$$

$t_{\text{calc}}$  = tempo di esecuzione di 1 addizione

# Domanda...

p	$T_p$
1	$15t_{\text{calc}}$
2	$8t_{\text{calc}}$
4	$5t_{\text{calc}}$
8	$4t_{\text{calc}}$

Qual è l'algoritmo che impiega meno tempo?



Quanto è più veloce di quello sequenziale?

# Esempio: calcolo della somma di $n=16$ numeri

$p$	$T(p)$	$\frac{T(1)}{T(p)}$
1	$15t_{\text{calc}}$	1.00
2	$8t_{\text{calc}}$	1.88
4	$5t_{\text{calc}}$	3.00
8	$4t_{\text{calc}}$	3.75

Maggiore  
riduzione del  
tempo ovvero  
maggiore  
aumento della  
velocità

L'algoritmo su 8 processori è il più veloce  
E' più veloce di 3.75 volte di quello su 1 processore

# Speed-up

Si definisce il rapporto  $T(1)$  su  $T(p)$

$$S(p) = \frac{T(1)}{T(p)}$$

Lo speed up misura la riduzione del tempo di esecuzione rispetto all'algoritmo su 1 processore

$$S(p) < p$$

SPEEDUP IDEALE

$$S^{\text{ideale}}(p) = p$$

# Speed-up

Si definisce il rapporto  $T(1)$  su  $T(p)$

$$S(p) = \frac{\text{righe}(1)}{\text{righe}(p)}$$

Lo speed up misura la riduzione del tempo di esecuzione rispetto all'algoritmo su 1 processore

$$S(p) < p$$

SPEEDUP IDEALE

$$S^{\text{ideale}}(p) = p$$

# Osservazione



$$S^{\text{ideale}}(p) = \frac{T(1)}{T(p)} = p$$

$$O_h = pT(p) - T(1)$$



$$T(p) = (O_h + T(1)) / p$$

OVERHEAD totale



$$S(p) = \frac{T(1)}{T(p)} = \frac{T(1)}{(O_h + T(1))/p} = \frac{pT(1)}{O_h + T(1)} = \frac{p}{\frac{O_h}{T(1)} + 1}$$

L'OVERHEAD totale misura  
quanto lo speed up differisce da quello ideale

# Osservazione



$$S^{\text{ideale}}(p) = \frac{T(1)}{T(p)} = p$$

$$O_h = pT(p) - T(1)$$

OVERHEAD totale

$$O_h = p * \text{righe}(p) * t_{\text{calc}} - \text{righe}(1) * t_{\text{calc}}$$

$$O_h = p * \text{righe}(p) - \text{righe}(1)$$

$$O_h = \text{spazi\_vuoti}(p)$$

L'OVERHEAD totale misura  
quanto lo speed up differisce da quello ideale

# Osservazione



$$S^{\text{ideale}}(p) = \frac{T(1)}{T(p)} = p$$

$$O_h = pT(p) - T(1)$$

OVERHEAD totale

$$O_h = p * \text{righe}(p) * t_{\text{calc}} - \text{righe}(1) * t_{\text{calc}}$$

$$O_h = p * \text{righe}(p) - \text{righe}(1)$$

$$O_h = \text{spazi\_vuoti}(p)$$

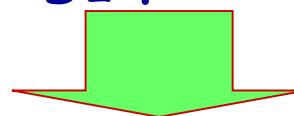
La rappresentazione degli operatori in tabella dà immediate informazioni sul tempo d'esecuzione e sull'overhead totale dell'algoritmo!

L'OVERHEAD totale misura  
quanto lo speed up differisce da quello ideale

# L'overhead nell'algoritmo parallelo della somma

$$T(1) = n - 1$$

$$T(p) = n/p - 1 + \log_2 p$$

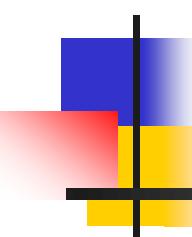


$$O_h = p T(p) - T(1) = p (n/p + \log_2 p) - (n-1) =$$

$$= n + p \log_2 p - n + 1 = O(p \log_2 p)$$

p	$O_h$
2	2
4	8
8	24
$2^k$	$p \log_2 p$

Al crescere di p  
l'overhead aumenta!



# Quindi

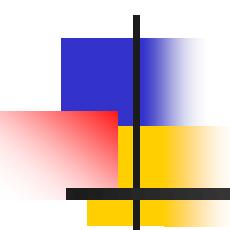
---

Se si vuole calcolare la somma di 16 numeri  
nel minor tempo possibile

l'algoritmo su 8 processori è da preferire

Infatti, aumentando il numero di processori  
si riduce

il tempo impiegato per eseguire le operazioni  
richieste



MA....

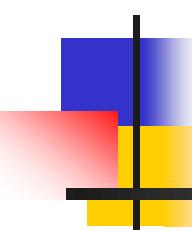
## Esempio: calcolo della somma di $n=16$ numeri

$p$	Speed-up ottenuto	Speed-up ideale
2	1.88	2
4	3.00	4
8	3.75	8

Lo speed-up su 8 processori è il maggiore

MA

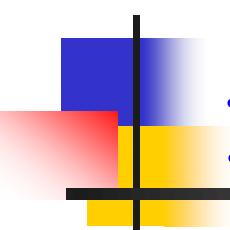
Lo speed-up su 2 processori è "il più vicino"  
allo speed-up ideale...



Cioè

---

Ho utilizzato 8 processori per  
avere un incremento  
di appena 4 volte



## In altre parole

---

... lo speed up non basta a  
fornire informazioni sull'efficienza  
dell'algoritmo parallelo!

... e allora ?

# Esempio: calcolo della somma di $n=16$ numeri

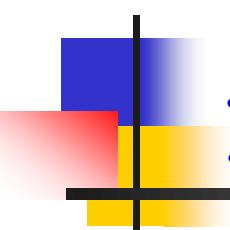
... se si rapporta lo speed-up al numero di processori...

$p$	$S(p)$	$\frac{S(p)}{p}$
2	1.88	0.94
4	3.00	0.75
8	3.75	0.47

Rapporto più grande



maggior sfruttamento dei  
processori per  $p=2$



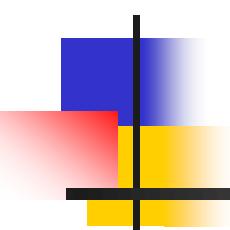
## In altre parole

---

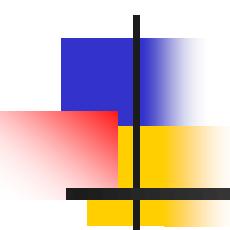
l'utilizzo di un maggior numero di processori **NON**  
è sempre una garanzia di sviluppo di algoritmi  
paralleli "efficaci"

OVVERO

Di algoritmi che sfruttano  
tutte le risorse della macchina parallela !



Come misurare se e quanto  
è stata sfruttato il calcolatore parallelo ?



# Efficienza

Si definisce il rapporto  $E_p$  su  $p$

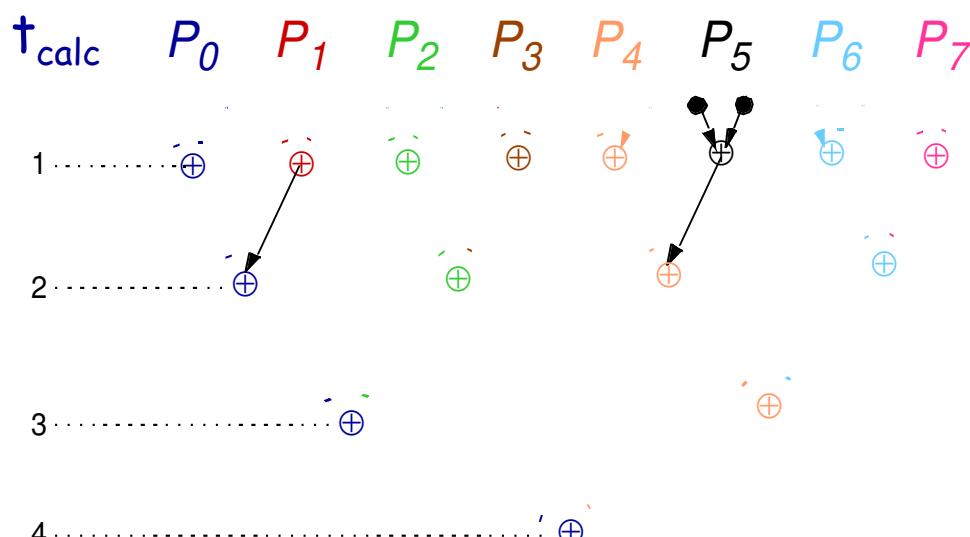
$$E(p) = \frac{S(p)}{p}$$

misura quanto l'algoritmo sfrutta il parallelismo del calcolatore

## EFFICIENZA IDEALE

$$E^{\text{ideale}}(p) = \frac{S^{\text{ideale}}(p)}{p} = 1$$

# Infatti, per la somma con p=8



$t_{\text{calc}}$  = tempo di esecuzione di 1 addizione

al I passo:

lavorano  $p=8$  processori

al II passo:

lavorano  $p/2=4$  processori

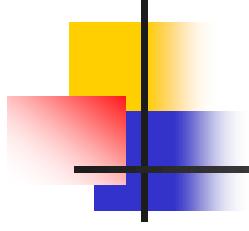
al III passo:

lavorano  $p/4=2$  processori

al IV passo:

lavora  $p/8=1$  processore!

Ad ogni passo si dimezza il numero  
di processori attivi



# Fine Lezione

# Valutazione dell'efficienza di algoritmi e software in ambiente parallelo

## parte 1



**PARALLEL AND DISTRIBUTED COMPUTING**

**PROF. G. LACCETTI**

**A.A. 2021/2022**

# Problema

2

Valutare l'efficienza di un algoritmo  
in ambiente di calcolo parallelo



Cosa si intende per "EFFICIENZA" ?

# Efficienza di un algoritmo sequenziale

3

- COMPLESSITA' di TEMPO  $T(n)$   
Numero di operazioni eseguite dall'algoritmo
- COMPLESSITA' di SPAZIO  $S(n)$   
Numero di variabili utilizzate dall'algoritmo

# In un algoritmo sequenziale

4

Il numero complessivo di operazioni

determina anche

il numero dei passi temporali

(Il tempo di esecuzione)

# In un software sequenziale

5

L'efficienza del software dipende dal  
tempo di esecuzione  
delle  $T(n)$  operazioni fl.p.

# Domanda

6

Cosa si intende  
per **efficienza** di un algoritmo  
in **ambiente parallelo**?

# Nell'algoritmo parallelo della somma...

7

Il numero delle operazioni

non è legato

al numero dei passi temporali

# Infatti...

8

Un calcolatore parallelo è in grado di eseguire più operazioni  
**concorrentemente**  
(allo stesso passo temporale)



Il tempo di esecuzione **non è proporzionale** alla complessità di tempo (ovvero  
**non dipende soltanto** dal numero di operazioni fl. p. effettuate)



La complessità di tempo **non è adatta** a misurare  
l'efficienza di un algoritmo parallelo

# In generale

9

Con  $p$  processori ci aspettiamo che

$T(1)$  sia  $p$  volte  $T(p)$

$$\frac{T(1)}{T(p)} = p$$

ovvero

ci aspettiamo di ridurre **p volte** il tempo di esecuzione

Da questo momento ci riferiremo a una macchina che sia in grado di eseguire l'operatore



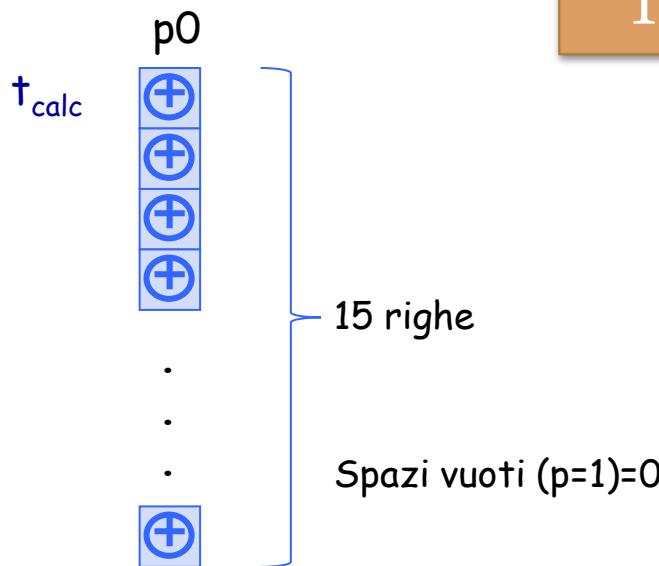
# Esempio: Somma di n=16 numeri

10

Diciamo che un operatore DIPENDE da un altro se la sua esecuzione è subordinata all'esecuzione dell'altro.

Disegniamo uno sotto l'altro gli operatori dipendenti

Disegniamo uno a fianco all'altro gli operatori che sono INDIPENDENTI tra loro, cioè che possono essere eseguiti concorrentemente



1 processore

$$\text{righe}(1) = 15$$

Tempo di esecuzione

$$T(1)=15 \cdot t_{\text{calc}}$$

$$\text{spazi_vuoti}(1) = 0$$

$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

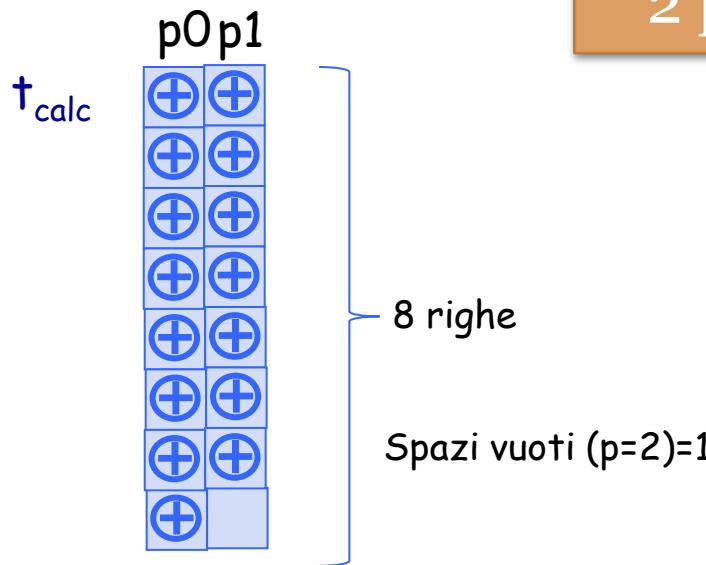
# Esempio: Somma di n=16 numeri

11

Diciamo che un operatore DIPENDE da un altro se la sua esecuzione è subordinata all'esecuzione dell'altro.

Disegniamo uno sotto l'altro gli operatori dipendenti

Disegniamo uno a fianco all'altro gli operatori che sono INDIPENDENTI tra loro, cioè che possono essere eseguiti concorrentemente



2 processori

$$\text{righe}(2) = 8$$

Tempo di esecuzione

$$T(2) = 8 \cdot t_{\text{calc}}$$

$$\text{spazi_vuoti}(2) = 1$$

$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

# Esempio: Somma di n=16 numeri

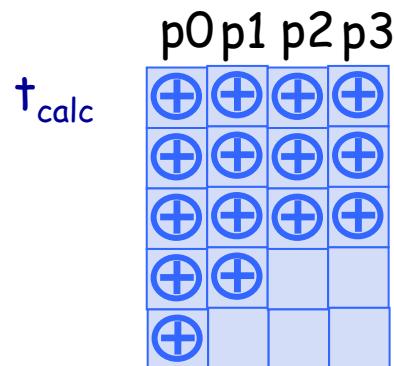
12

Diciamo che un operatore DIPENDE da un altro se la sua esecuzione è subordinata all'esecuzione dell'altro.

Disegniamo uno sotto l'altro gli operatori dipendenti

Disegniamo uno a fianco all'altro gli operatori che sono INDIPENDENTI tra loro, cioè che possono essere eseguiti concorrentemente

4 processori



$$\text{righe}(4) = 5$$

Tempo di esecuzione

$$T(2) = 5 + t_{\text{calc}}$$

$$\text{spazi_vuoti}(4) = 5$$

$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

# Esempio: Somma di n=16 numeri

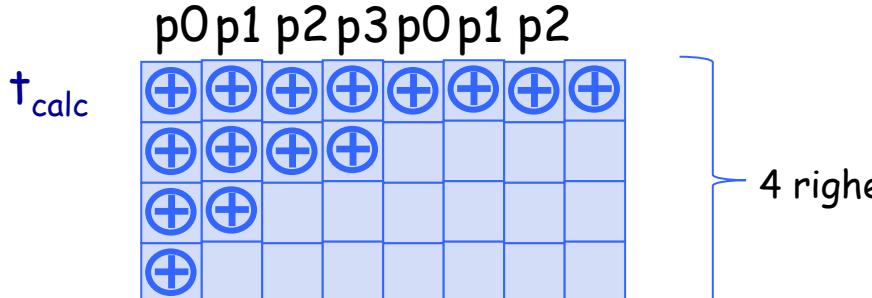
13

Diciamo che un operatore DIPENDE da un altro se la sua esecuzione è subordinata all'esecuzione dell'altro.

Disegniamo uno sotto l'altro gli operatori dipendenti

Disegniamo uno a fianco all'altro gli operatori che sono INDIPENDENTI tra loro, cioè che possono essere eseguiti concorrentemente

8 processori



$$\text{righe}(8) = 4$$

Tempo di esecuzione

$$T(8)=4 \cdot t_{\text{calc}}$$

$$\text{spazi_vuoti}(8) = 17$$

Spazi vuoti ( $p=8$ )=17

$t_{\text{calc}} = \text{tempo di esecuzione di 1 addizione}$

# Speed Up

14

Si definisce SPEED UP il rapporto

$$S(p) = \frac{T(1)}{T(p)}$$

misura

la riduzione del tempo di esecuzione rispetto all'algoritmo su 1 processore

E in generale è

$$S(p) < p = S^{ideale}(p)$$

# Speed Up

15

Si definisce SPEED UP il rapporto

Possiamo  
scrivere anche

$$S(p) = \frac{\text{righe}(1)}{\text{righe}(p)}$$

misura

la riduzione del tempo di esecuzione rispetto all'algoritmo su 1 processore

E in generale è

$$S(p) < p = S^{\text{ideale}}(p)$$

# Speed Up

16

$$S^{ideale}(p) = \frac{T(1)}{T(p)} = p$$

**OVERHEAD totale**

$$O_h(p) = pT(p) - T(1)$$



L'OVERHEAD totale misura  
quanto lo speed up differisce da quello ideale

# Speed Up

17

$$S^{ideale}(p) = \frac{T(1)}{T(p)} = p$$

**OVERHEAD totale**

$$O_h(p) = pT(p) - T(1)$$

→  $O_h(p) = p * \text{righe}(p) * \text{tcalc} - \text{righe}(1) * \text{tcalc}$

→  $O_h(p) = p * \text{righe}(p) - \text{righe}(1)$

→  $O_h(p) = \text{spazi_vuoti}(p)$



L'OVERHEAD totale misura  
quanto lo speed up differisce da quello ideale

# Speed Up

18

$$S^{ideale}(p) = \frac{T(1)}{T(p)} = p$$

**OVERHEAD totale**

$$O_h(p) = pT(p) - T(1)$$

→  $O_h(p) = p * \text{righe}(p) * \text{tcalc} - \text{righe}(1) * \text{tcalc}$

→  $O_h(p) = p * \text{righe}(p) - \text{righe}(1)$

→  $O_h(p) = \text{spazi_vuoti}(p)$



OVERHEAD = ...

**La rappresentazione degli operatori in tabella dà immediate informazioni sul tempo d'esecuzione e sull'overhead totale dell'algoritmo!**

# Speed Up

19

$$S^{ideale}(p) = \frac{T(1)}{T(p)} = p$$

**OVERHEAD totale**

$$O_h(p) = pT(p) - T(1)$$

$$T(p) = \frac{(O_h(p) + T(1))}{p}$$

$$S(p) = \frac{T(1)}{T(p)} = \frac{T(1)}{(O_h + T(1))/p} = \frac{pT(1)}{O_h + T(1)} = \frac{p}{\frac{O_h}{T(1)} + 1}$$

# Quindi

20

Se si vuole calcolare la somma di 16 numeri  
nel minor tempo possibile

l'algoritmo su 8 processori è da preferire



Aumentando il numero di processori  
si riduce

il tempo impiegato per eseguire le operazioni richieste

# Esempio: Somma di n=16 numeri

21

p	S(p)	S <sup>Ideale</sup> (p)
2	1.88	2
4	3.00	4
8	3.75	8

Lo speed-up su 8 processori è il maggiore

MA

Lo speed-up su 2 processori è “il più vicino” allo speed-up ideale...

# Esempio: Somma di n=16 numeri

22

$p$	$S(p)$	$\frac{S(p)}{p}$
2	1.88	0.94
4	3.00	0.75
8	3.75	0.47

Lo speed-up su 8 processori è il maggiore

MA

maggior sfruttamento dei processori per  $p=2$

# In altre parole...

23

l'utilizzo di un maggior numero di processori NON è sempre una garanzia  
di sviluppo di algoritmi paralleli “efficaci”

OVVERO

Di algoritmi che sfruttano tutte le risorse della macchina parallela !

**Come misurare se e quanto  
è stato “sfruttato” il calcolatore parallelo ?**

# Efficienza

24

Si definisce EFFICIENZA il rapporto

$$E(p) = \frac{S(p)}{p}$$

misura

quanto l'algoritmo sfrutta il parallelismo del calcolatore

E in generale è

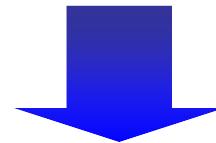
$$E(p) < 1 = \frac{S^{ideale}(p)}{p} = E^{ideale}(p)$$

# Efficienza ed Overhead totale

25

In generale, ricordando le definizioni

$$\begin{cases} E(p) = \frac{S(p)}{p} \\ O_h = pT(p) - T(1) \end{cases}$$



$$E(p) = \frac{S(p)}{p} = \frac{T(1)}{pT(p)} = \frac{T(1)}{O_h + T(1)} = \frac{1}{\frac{O_h}{T(1)} + 1}$$

# Domanda

26

E' possibile ottenere  
speed-up prossimi allo speed-up ideale ?

# $T(1)$ si può decomporre in 2 parti:

27

una parte relativa alle operazioni che sono eseguite esclusivamente in sequenziale

una parte relativa alle operazioni che possono essere eseguite concorrentemente

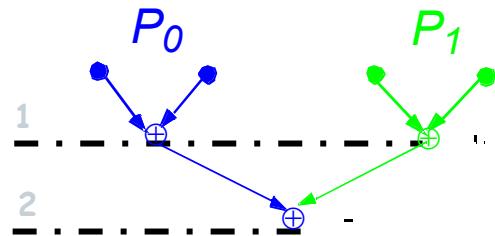
$T_s$

$T_c$

$$T(1) = T_s + T_c$$

# Somma di $n = 4$ su $p=2$ , $T(1) = (n - 1)t_{\text{calc}} = 3t_{\text{calc}}$

28



2 addizioni eseguite concorrentemente ( $T_c$ ) da 2 processori ( $p$ ) al passo 1

1 addizione eseguita in sequenziale ( $T_s$ ) da 1 processore al passo 2

$$T(1) = T_s + T_c$$



$$T(1) = (1 + 2)t_{\text{calc}} = 3t_{\text{calc}}$$

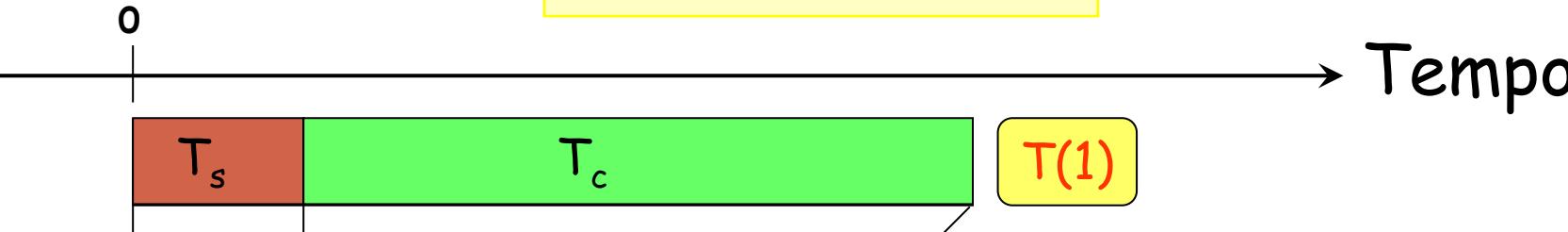
$$T_s = 1 t_{\text{calc}}$$

Operazione che deve essere eseguita in sequenziale

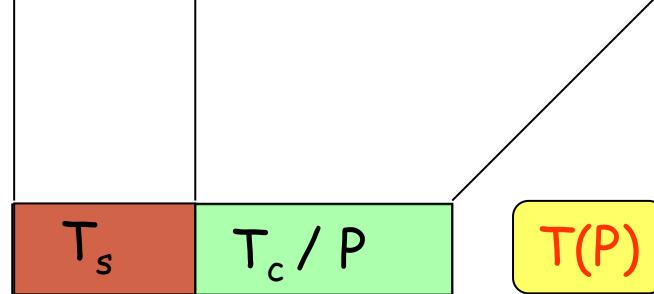
$$T_c = 2 t_{\text{calc}}$$

Operazioni che possono essere eseguite in parallelo

$$T(1) = T_s + T_c$$



Operazioni che  
possono essere eseguite  
da 1 processore      Operazioni  
che possono essere  
eseguite concorrentemente da  
 $P$  processori



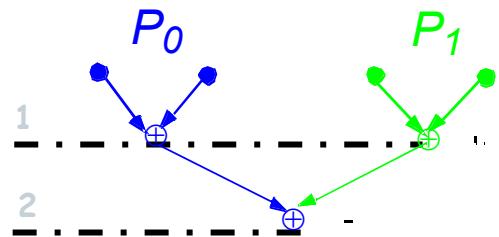
Calcoliamo il tempo impiegato  
dall'algoritmo parallelo  
con  $P$  processori, tenendo presente  
che  
le operazioni relative alla parte  
parallela

$$T(p) = T_s + \frac{T_c}{p}$$

sono ora seguite concorrentemente  
dai  $P$  processori

# Somma di $n = 4$ su $p=2$ , $T(1) = (n - 1)t_{\text{calc}} = 3t_{\text{calc}}$

30



2 addizioni eseguite concorrentemente ( $T_c$ ) da 2 processori ( $p$ ) al passo 1

1 addizione eseguita in sequenziale ( $T_s$ ) da 1 processore al passo 2

$$T(1) = T_s + T_c/p$$



$$T(p) = (1+1)t_{\text{calc}} = 2t_{\text{calc}}$$

$$T_s = 1 t_{\text{calc}}$$

Operazione che deve essere eseguita in sequenziale

$$T_c/p = 2/2 = 1 t_{\text{calc}}$$

Operazioni che possono essere eseguite in parallelo

# Calcoliamo lo speed-up:

31

Somma di  $n = 4$  su  $p=2$ ,

$$T(1) = 3 t_{\text{calc}}$$

$$T(2) = 2 t_{\text{calc}}$$



$$S(2) = \frac{T(1)}{T(p)} = \frac{3t_{\text{calc}}}{2t_{\text{calc}}} = 1,5 < 2 = S^{\text{ideale}}(p)$$

# Nel calcolare lo speed up...

32

... abbiamo calcolato di

quanto si riduce il tempo impiegato

da un solo processore se

le operazioni concorrenti

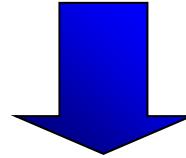
sono eseguite da p processori

# In generale

33

Il tempo di esecuzione di un algoritmo parallelo, distribuito su  $p$  processori, comprende 2 componenti:

- $T_s$  tempo per eseguire la parte seriale
- $T_c/p$  tempo per eseguire la parte parallela



$$T(p) = T_s + \frac{T_c}{p}$$

# Overhead totale $O_h$

34

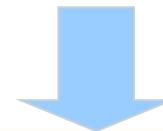
$$O_h = pT(p) - T(1)$$

dove

$$T(p) = T_s + \frac{T_c}{p}$$



posto  $T(1)=1$  ,  $T_s=\alpha$  ,  $T_c=1-\alpha$



$$O_h = (p - 1)\alpha$$

Con  $\alpha$  parte seriale  
cioè la parte non  
parallelizzabile  
dell'algoritmo

Osservazione:

$O_h$  dipende da  $p$  e da  $\alpha$

# Quindi l'efficienza...

35

$$\left\{ \begin{array}{l} E(p) = \frac{S(p)}{p} = \frac{1}{O_h + 1} \\ O_h = (p - 1)\alpha \end{array} \right.$$

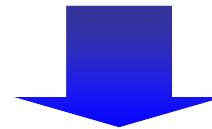


$$E(p) = \frac{1}{O_h + 1} = \frac{1}{(p - 1)\alpha + 1}$$

# ...e lo Speed Up

36

$$E(p) = \frac{1}{(p-1)\alpha + 1}$$



$$S(p) = p \cdot E(p) = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

*Legge di Ware (Amdahl)*

# ...e lo Speed Up

37

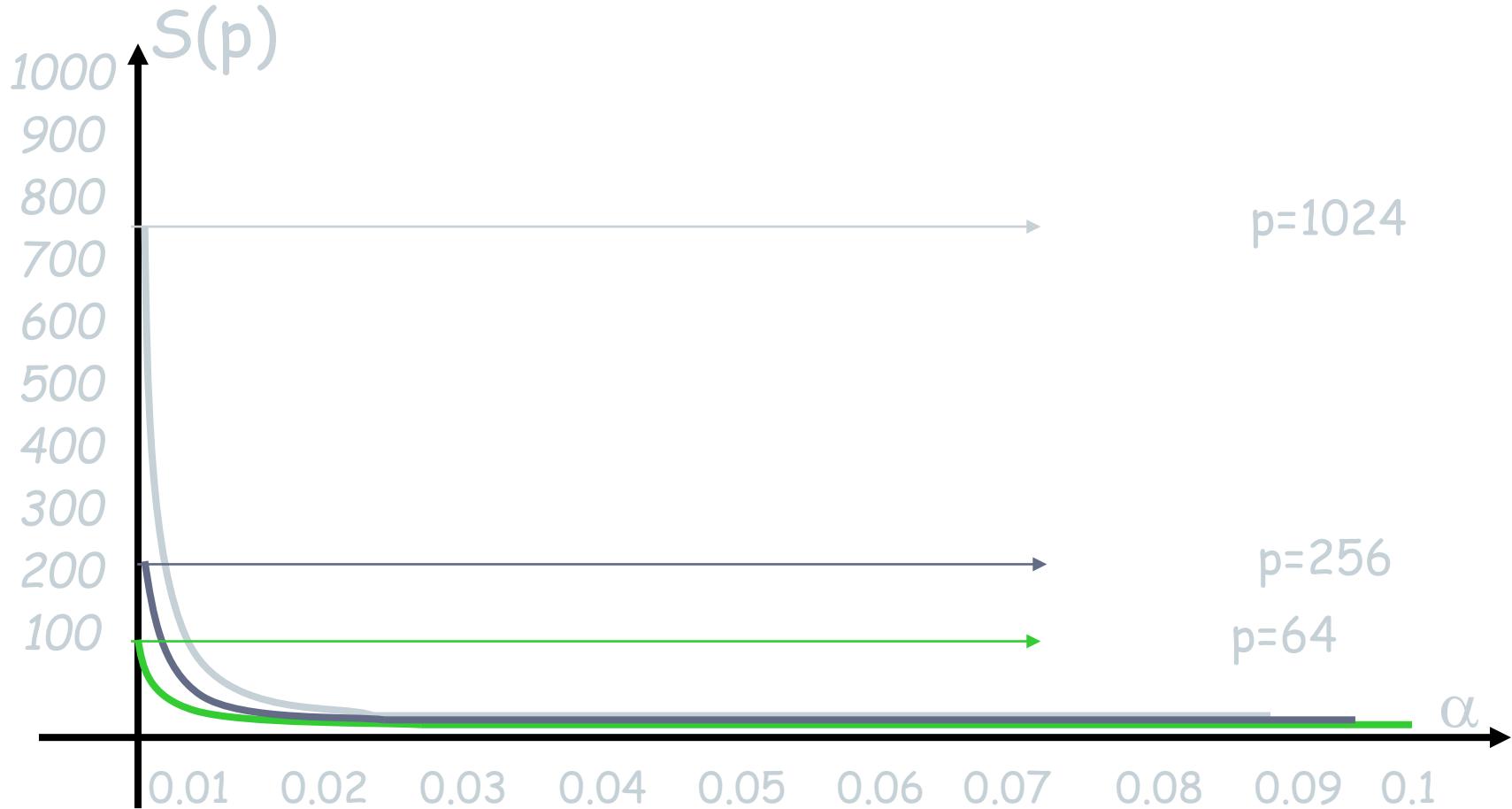
Analizziamo l'andamento dello speed-up all'aumentare del numero p di processori

$$S(p) = \frac{1}{\alpha + \frac{(1-\alpha)}{p}} \longrightarrow \frac{1}{\alpha}$$

Il valore asintotico di

$$S(p) \text{ è } \frac{1}{\alpha}$$

## Andamento speed up secondo la legge di Amdahl



Una "piccola" parte sequenziale può **degradare fortemente lo speed-up**, quando il numero di processori è sufficientemente elevato!

# Esempio 1 (n fissato e p variabile)

39

Applichiamo la legge di Amdahl con  $n=32$  e  $p=2, 4, 8, 16$

$\alpha_1$  = frazione di  $T_1$  eseguita da un solo processore (sequenziale)

$\alpha_p$  = frazione eseguita con parallelismo totale  $p$

Al crescere del numero  $p$  di processori...

$p$	$\alpha_1$	$\alpha_p$	$S(p)$	$E(p)$
2	0,032	0,968	1,9	0,95
4	0,032	0,903	3,4	0,85
8	0,032	0,775	5,1	0,6
16	0,032	0,516	6,2	0,3

Speed up ed  
efficienza  
degradano  
perché la parte  
parallela diminuisce!

# In sintesi

40

Se la dimensione  $n$  del problema è fissata,  
al crescere del numero  $p$  di processori,  
non solo non si riescono ad ottenere  
speed up vicini a quello ideale

MA

Le prestazioni peggiorano!

(non conviene utilizzare un maggior numero di processori!!)

# Esempio 2 (n aumenta e p fissato)

41

Applichiamo la legge di Amdahl con  $p=2$  e  $n = 8, 16, 32, 64$

$\alpha_1$  = frazione di  $T_1$  eseguita da un solo processore (sequenziale)

$\alpha_p$  = frazione eseguita con parallelismo totale  $p$

Al crescere della dimensione n...

n	$\alpha$	$1-\alpha$	$S(2,n)$	$E(2,n)$
8	0,14	0,86	1,75	0,875
16	0,06	0,94	1,8	0,9
32	0,03	0,97	1,9	0,96
64	0,01	0,99	1,96	0,99

La parte  
*sequenziale*  
tende a zero!

# Esempio 2 (n aumenta e p fissato)

42

Applichiamo la legge di Amdahl con  $p=2$  e  $n = 8, 16, 32, 64$

$\alpha_1$  = frazione di  $T_1$  eseguita da un solo processore (sequenziale)

$\alpha_p$  = frazione eseguita con parallelismo totale  $p$

Al crescere della dimensione n...

n	$\alpha$	$1-\alpha$	$S(2,n)$	$E(2,n)$
8	0,14	0,86	1,75	0,875
16	0,06	0,94	1,8	0,9
32	0,03	0,97	1,9	0,96
64	0,01	0,99	1,96	0,99

La parte *parallela* cresce, *tende a 1*

# Esempio 2 (n aumenta e p fissato)

43

Applichiamo la legge di Amdahl con  $p=2$  e  $n = 8, 16, 32, 64$

$\alpha_1$  = frazione di  $T_1$  eseguita da un solo processore (sequenziale)

$\alpha_p$  = frazione eseguita con parallelismo totale  $p$

Al crescere della dimensione n...

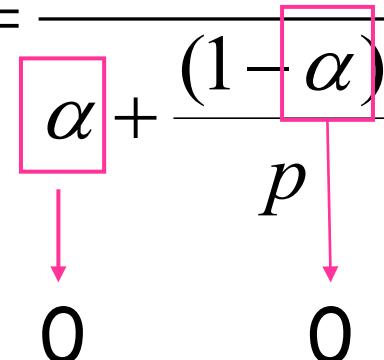
n	$\alpha$	$1-\alpha$	$S(2,n)$	$E(2,n)$
8	0,14	0,86	1,75	0,875
16	0,06	0,94	1,8	0,9
32	0,03	0,97	1,9	0,96
64	0,01	0,99	1,96	0,99

Speed up ed  
efficienza sono  
“costanti”!

# Infatti

44

Se  $p$  è fissato, al crescere della dimensione  $n$  del problema...

$$S(p) = \frac{1}{\alpha + \frac{(1-\alpha)}{p}}$$


Il valore asintotico di

$S(p)$  è  $p$

$(S^{\text{ideale}}(p) = p)$

# In sintesi

45

Fissando il numero  $p$  di processori e aumentando la dimensione del problema si possono ottenere speed up prossimi a quello ideale

MA

Non è possibile aumentare in maniera indefinita la dimensione  $n$  del problema:  
le risorse (hardware) sono limitate!

# Quindi

46

Secondo la legge di Ware...

Aumentando il numero  $p$  di processori e mantenendo fissata la dimensione  $n$  del problema si riesce ad utilizzare in maniera efficiente l'ambiente di calcolo parallelo, se  $p \leq p_o$

Aumentando la dimensione  $n$  del problema e mantenendo fisso il numero  $p$  di processori le prestazioni dell'algoritmo parallelo non degradano se  $n \leq n_0$

# Domanda

47

Calcolando:  $S(p) = \frac{1}{\alpha + \frac{(1-\alpha)}{p}}$

Cosa succede se **aumentiamo**  
il numero **p** di processori e  
la dimensione **n** del problema ?

# Esempio: Somma di n numeri

48

Applichiamo la legge di Amdhal con  $p= 2, 4, 8, 16$

Consideriamo quindi  $n= 8, 16, 32, 64$

$$\frac{n}{p} = 4 \text{ è costante}$$

n	p	$\alpha_1$	$\alpha_p$	$S(p,n)$	$E(p,n)$
8	2	0,14	0,86	1,75	0,875
16	4	0,06	0,8	3	0,75
32	8	0,03	0,77	5,1	0,64
64	16	0,01	0,76	9	0,56

La frazione di operazioni eseguite in sequenziale decresce!

# Esempio: Somma di n numeri

49

Applichiamo la legge di Amdhal con  $p= 2, 4, 8, 16$

Consideriamo quindi  $n= 8, 16, 32, 64$

$$\frac{n}{p} = 4 \text{ è costante}$$

n	p	$\alpha_1$	$\alpha_p$	S(p,n)	E(p,n)
8	2	0,14	0,86	1,75	0,875
16	4	0,06	0,8	3	0,75
32	8	0,03	0,77	5,1	0,64
64	16	0,01	0,76	9	0,56

La frazione di operazioni eseguite *in parallelo su tutti i p processori* rimane quasi costante!

# Esempio: Somma di n numeri

50

Applichiamo la legge di Amdhal con  $p = 2, 4, 8, 16$

Consideriamo quindi  $n = 8, 16, 32, 64$

$$\frac{n}{p} = 4 \text{ è costante}$$

n	p	$\alpha_1$	$\alpha_p$	$S(p,n)$	$E(p,n)$
8	2	0,14	0,86	1,75	0,875
16	4	0,06	0,8	3	0,75
32	8	0,03	0,77	5,1	0,64
64	16	0,01	0,76	9	0,56

Lo speed up AUMENTA!

# Esempio: Somma di n numeri

51

Applichiamo la legge di Amdhal con  $p = 2, 4, 8, 16$

Consideriamo quindi  $n = 8, 16, 32, 64$

$$\frac{n}{p} = 4 \text{ è costante}$$

n	p	$\alpha_1$	$\alpha_p$	$S(p,n)$	$E(p,n)$
8	2	0,14	0,86	1,75	0,875
16	4	0,06	0,8	3	0,75
32	8	0,03	0,77	5,1	0,64
64	16	0,01	0,76	9	0,56

L'efficienza è  
"quasi costante"

# Quindi

52

Aumentando sia  $n$  che  $p$ ,  
le prestazioni  
dell'algoritmo parallelo  
non degradano

Ma  
aumentando sia  $n$  che  $p$   
cosa ci aspettiamo dall'algoritmo parallelo?

# Quindi

53

Se  $p=2$  ci si aspetta di calcolare  
nel tempo  $T(1,n)$  la somma di  $2n$   
numeri

$$T(1,n) = T(2,2n)$$

Se  $p=4$  ci si aspetta di calcolare  
nel tempo  $T(1,n)$  la somma di  $4n$   
numeri

In generale:

$$T(1,n) = T(4,4n)$$

$$T(1,n) = T(2,2n) = T(4,4n) = \dots = T(p,pn)$$

# In generale per la somma...

54

$$T(1,n) = T(2,2n) = T(4,4n) = \dots = T(p,pn)$$

Pertanto...

$$\frac{T(1,n)}{T(p,pn)} = 1 \quad \longleftrightarrow \quad p \cdot \frac{T(1,n)}{T(p,pn)} = p \quad \longleftrightarrow \quad \frac{T(1,pn)}{T(p,pn)} = p = S_p^{ideale}$$

..se si assume

$$pT(1,n) = T(1,pn)$$

Ovvero se si assume  $T(1,pn)$  uguale al tempo che si ottiene **moltiplicando per p il tempo** per risolvere su 1 processore il problema di dimensione n...

# In generale per la somma...

55

$$SS(p,n) = \frac{T(1,pn)}{T(p,pn)} = \frac{p T(1,n)}{T(p,np)}$$

**SPEEDUP SCALATO**

# In generale per la somma...

56

dividendo  $SS(p,n)$  per il numero di processori

$$ES(p,n) = \frac{SS(p,n)}{p} = \frac{T(1,n)}{T(p,pn)}$$

**EFFICIENZA SCALATA**

# Valutazione dell'efficienza di algoritmi e software in ambiente parallelo

## parte 2-o



**PARALLEL AND DISTRIBUTED COMPUTING**

**PROF. G. LACCETTI**

**A.A. 2021/2022**

# Domanda

2

E' possibile ottenere  
speed-up (scalati) prossimi allo  
speed-up ideale?

# $T(p)$ si può decomporre in 2 parti:

3

una parte relativa alle operazioni che sono eseguite esclusivamente in sequenziale

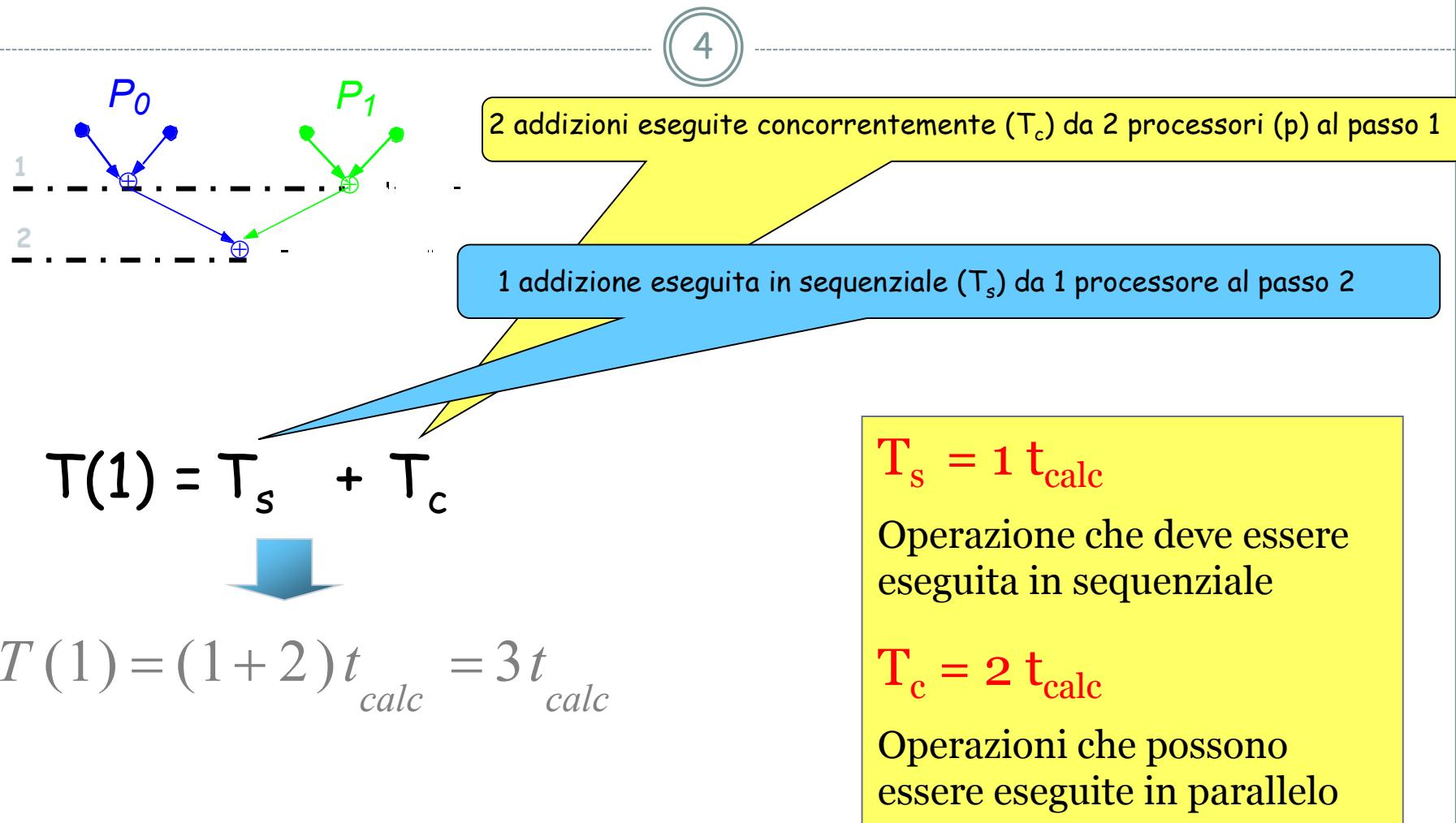
una parte relativa alle operazioni che possono essere eseguite concorrentemente

$T_s$

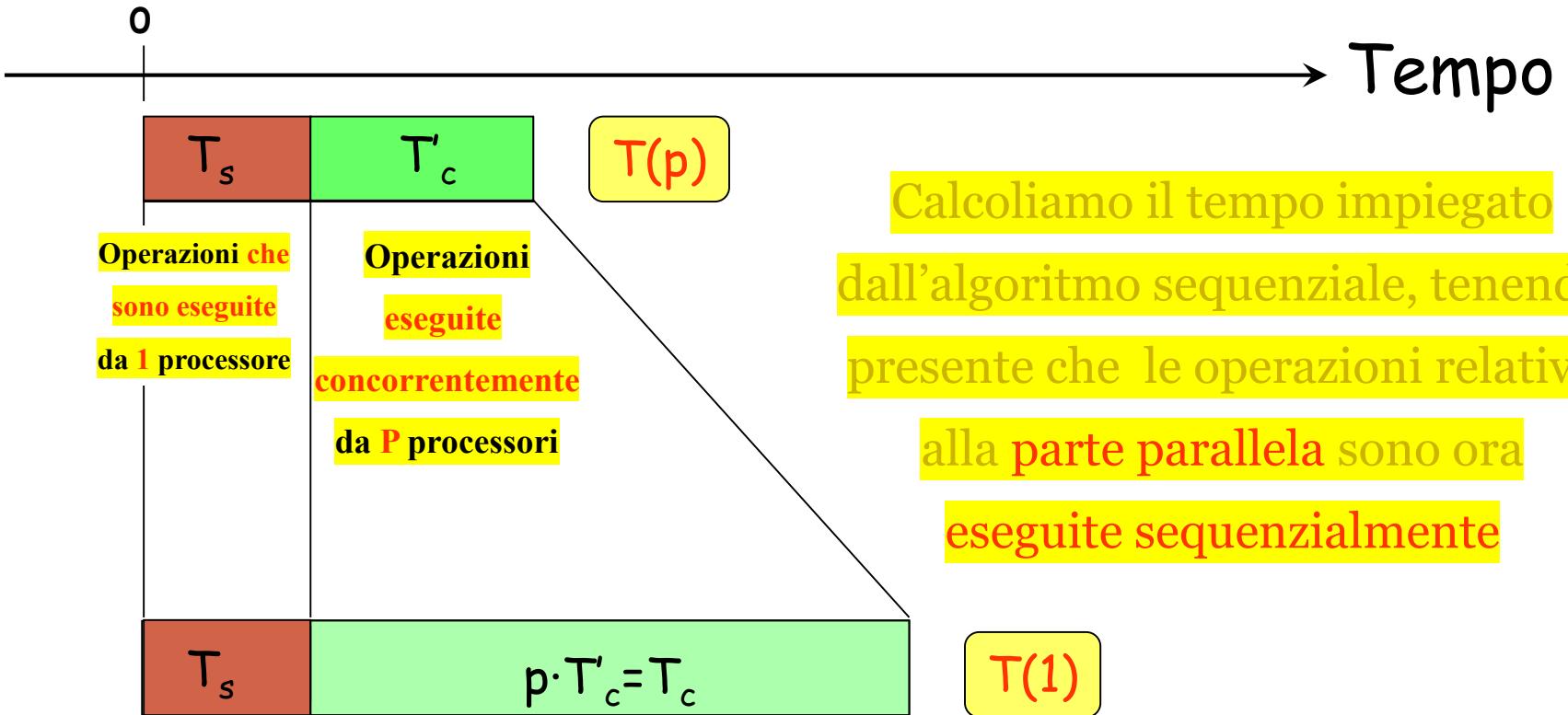
$T'_c$

$$T(1) = T_s + T'_c$$

# Somma di $n = 4$ su $p=2$ , $T(1) = (n - 1)t_{\text{calc}} = 3t_{\text{calc}}$



$$T(p) = T_s + T_c'$$



$$T(1) = T_s + p \cdot T_c'$$

# Lo speed up...

6

Avendo :

$$\left\{ \begin{array}{l} T(1) = T_s + p \cdot T_c' \\ T(p) = T_s + T_c' \end{array} \right.$$

Si assume  $T(p)$  come tempo unitario



$$T(p) = T_s + T_c' = 1 \longrightarrow T_c' = 1 - T_s$$



$$T(1) = T_s + p(1 - T_s)$$

# Lo speed up...

7

Posto  $\mathbf{T}_s = \alpha'$

(frazione di  $T(p)$  per la componente sequenziale)



$$SS(p) = \frac{T(1)}{T(p)} = \frac{T_s + p(1-T_s)}{1} = \alpha' + p(1-\alpha') = p + (1-p)\alpha'$$

Modello di **SPEED UP SCALATO**

**LEGGE DI GUSTAFSON**

(Reinterpretazione della legge di Ware)

# Lo speed up...

8

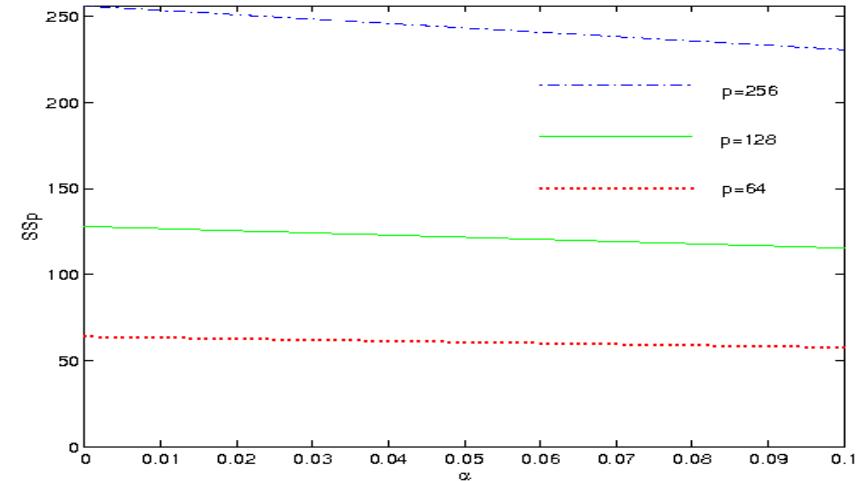
$$SS(p) = \alpha' + p(1 - \alpha') = p + (1 - p)\alpha'$$

$$\alpha' = 0.10$$

$$p = 10 \rightarrow SS(p) = 9.10$$

$$p = 64 \rightarrow SS(p) = 57.60$$

$$p = 100 \rightarrow SS(p) = 90.1$$



Si possono ottenere  
Speed up prossimi allo speed-up Ideale!

# Quindi

9

Nell'algoritmo della somma

abbiamo calcolato speedup ed efficienza scalata

aumentando la complessità di tempo del problema

in maniera proporzionale al numero di processori

ovvero

Nel passare da  $p_0$  a  $p_1$  processori con  $p_1 = k p_0$ ,

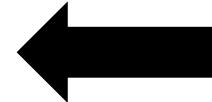


Siamo passati da  $T(p_0, n_0)$  a  $T(p_1, n_1)$  con  $n_1 = k n_0$

# Quindi

10

$$T(1, n_1) = \frac{p_1}{p_0} T(1, n_0)$$



$$T(1, n_1) = I(p_0, p_1, n_0)$$

k

la funzione  $I$  esprime la legge secondo cui deve aumentare la *complessità di tempo* dell'algoritmo sequenziale con dimensione  $n_1$  su  $p_1$  processori a partire da  $p_0$  e da  $T(1, n_0)$  affinché l'efficienza scalata non degradi.

I è detta ISOEFFICIENZA

# Domanda

11

Come si calcola l'**isoefficienza**  
per un qualsiasi algoritmo ?

# Domanda

12

Nel passare da  $p_0$  a  $p_1$  processori con  $p_1 > p_0$ ,  
come deve aumentare (*scalare*)  
la complessità del problema  
affinché l'efficienza sia costante ?

# Come calcolare l'isoefficienza?

13

$$E(p_0, n_0) = \frac{S(p_0, n_0)}{p_0} = \frac{1}{\frac{O_h(n_0, p_0)}{T(1, n_0)} + 1} = E(p_1, n_1) = \frac{S(p_1, n_1)}{p_1} = \frac{1}{\frac{O_h(n_1, p_1)}{T(1, n_1)} + 1}$$



$$T(1, n_1) = T(1, n_0) \frac{O_h(n_1, p_1)}{O_h(n_0, p_0)}$$

$$\frac{O_h(n_0, p_0)}{T(1, n_0)} = \frac{O_h(n_1, p_1)}{T(1, n_1)}$$

**ISOEFFICIENZA**

# Come calcolare l'isoefficienza della somma?

14

$$O_h(p) = p \log_2 p$$

$$T(1,n) = n$$



$$n_1 = n_0 \frac{p_1 \log_2 p_1}{p_0 \log_2 p_0} = I(n_0, p_0, p_1)$$

**ISOEFFICIENZA**

Nel passare da  $p_0$  a  $p_1$  con  $p_1 > p_0$ , la complessità di tempo del problema deve aumentare, rispetto alla dimensione iniziale  $n_0$ , del fattore

$$k = (p_1 \log_2 p_1) / (p_0 \log_2 p_0)$$

# In generale

15

Un algoritmo si dice **scalabile** se  
l'efficienza rimane **costante** (o non degrada)  
al crescere del numero dei processori e  
della dimensione del problema

# Quello per la somma è un algoritmo scalabile?

16

$$\frac{n_1}{n_0} = \frac{p_1 \log_2 p_1}{p_0 \log_2 p_0}$$

k

$$p_0 = 4$$

$$n_0 = 64$$

$$p_1 = 8$$

$$n_1 = ? \times n_0 = 192$$

$$p_1 = 16$$

$$n_1 = ? \times n_0 = 512$$

$$k = (8 \times 3) / (4 \times 2) = 3$$

$$k = (16 \times 4) / (4 \times 2) = 8$$

# Somma di n numeri, p processori: isoefficienza

17

$\frac{p}{n}$	1	4	8	16	32
64	1.0	.80	.57	.33	.17
192	1.0	.92	.80	.60	.38
512	1.0	.97	.91	.80	.62

L'efficienza rimane costante al crescere di p e di n

La somma è scalabile!

# Speed up ed efficienza scalata

18

$$SS = \frac{T(p_0, n_0)}{T(p_1, n_1)}$$

Speed-up scalato

$$ES = \frac{SS}{p}$$

Efficienza scalata

# Problema

19

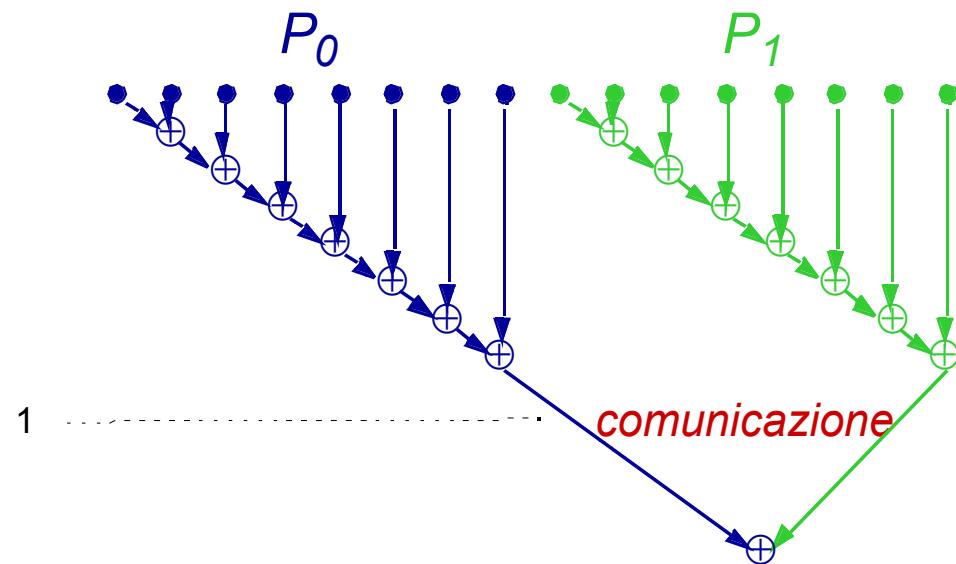
Se si esegue l'algoritmo su un  
**calcolatore MIMD a memoria distribuita**,  
il tempo di esecuzione dipende solo  
dal numero di operazioni eseguite  
in differenti passi temporali?

# Esempio: calcolo della somma di n=16 numeri

20

## ALGORITMO PARALLELO

$p=2$



**L'algoritmo richiede la comunicazione di 1 dato tra 2 processori**



$$T(2) = 8 t_{\text{calc}} + t_{\text{com}}$$

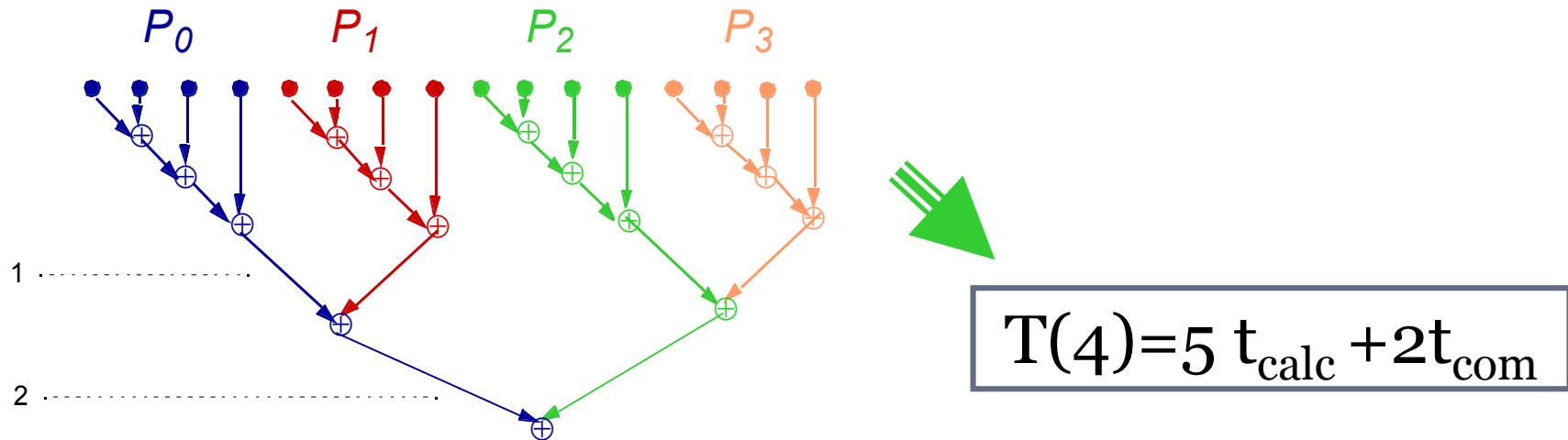
$t_{\text{com}}$  = tempo per comunicare un dato tra due processori

# Esempio: calcolo della somma di n=16 numeri

21

## ALGORITMO PARALLELO

$p=4$



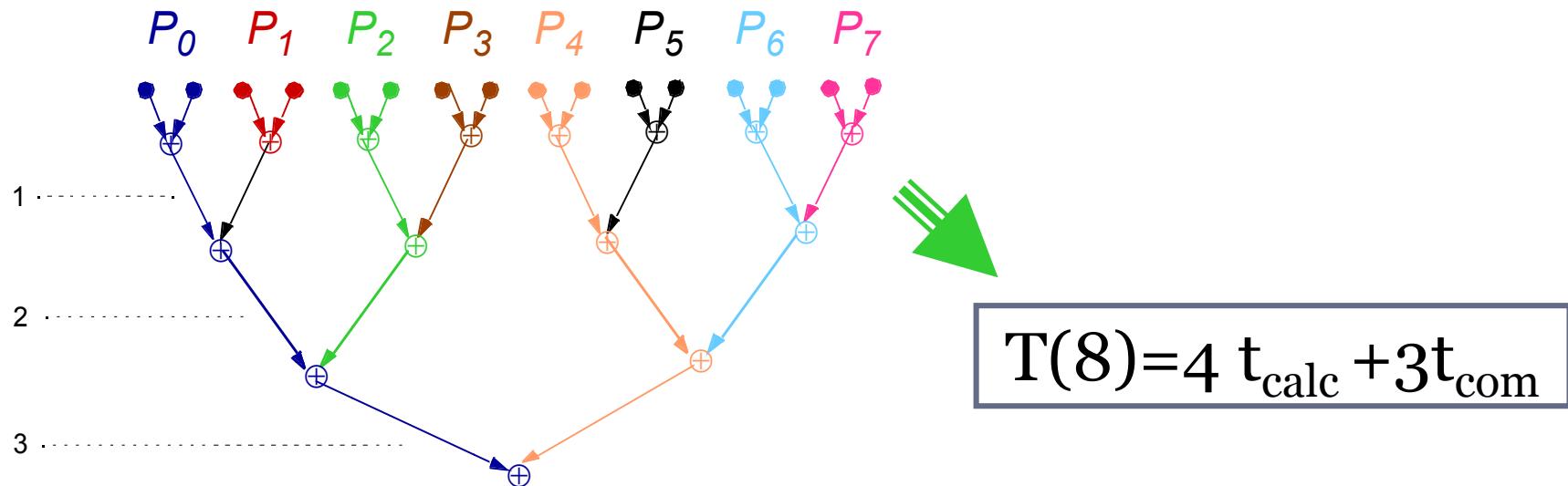
$t_{\text{com}}$  = tempo per comunicare un dato tra due processori

# Esempio: calcolo della somma di n=16 numeri

22

## ALGORITMO PARALLELO

$p=8$

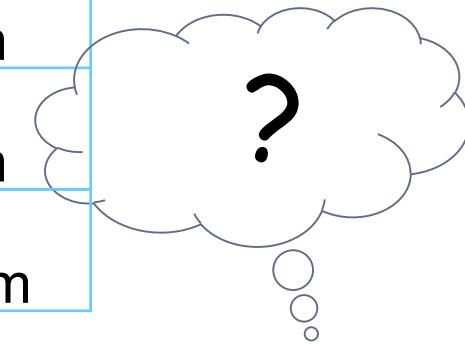


$t_{\text{com}}$  = tempo per comunicare un dato tra due processori

# In sintesi...

23

p	T(p)
1	$15t_{\text{calc}}$
2	$8t_{\text{calc}} + 1 t_{\text{com}}$
4	$5t_{\text{calc}} + 2 t_{\text{com}}$
8	$4t_{\text{calc}} + 3 t_{\text{com}}$
$2^k$	$? t_{\text{calc}} + ? t_{\text{com}}$



In generale quanto vale  $T(p)$  considerando le comunicazioni?

# In generale: calcolo di $T(p)$ considerando le comunicazioni

24

ALGORITMO PARALLELO della somma di  $n$  numeri su  $p=2^k$  processori

$$p=1 \quad T(1)=15 \text{ } t_{\text{calc}}$$

$$p=2 \quad T(2)=8 \text{ } t_{\text{calc}} = (7+1) \text{ } t_{\text{calc}} + 1 \text{ } t_{\text{com}}$$

$$p=4 \quad T(4)=5 \text{ } t_{\text{calc}} = (3+2) \text{ } t_{\text{calc}} + 2 \text{ } t_{\text{com}}$$

$$p=8 \quad T(8)=4 \text{ } t_{\text{calc}} = (1+3) \text{ } t_{\text{calc}} + 3 \text{ } t_{\text{com}}$$

.....

$$p=2^k \quad T(p) = \left(\frac{n}{p} - 1 + \log_2 p\right) t_{\text{calc}} + (\log_2 p) t_{\text{com}}$$

$t_{\text{calc}}$  = tempo di esecuzione di 1 addizione

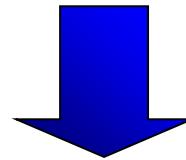
$t_{\text{com}}$  = tempo di 1 comunicazione

# In generale

25

Il tempo di esecuzione di un algoritmo parallelo, distribuito su  $p$  processori, comprende 3 componenti:

- $T_s$  tempo per eseguire la parte seriale
- $T_c/p$  tempo per eseguire la parte parallela
- $T_o$  costo di comunicazione con  $T_o(p) \geq 0, p > 1$



$$T(p) = T_s + \frac{T_c}{p} + T_o(p)$$

# Overhead di comunicazione unitario

26

$t_{com}$  = tempo di comunicazione di 1 dato

$t_{calc}$  = tempo di esecuzione di 1 operazione fl.p.

definiamo

Overhead di Comunicazione Unitario

il rapporto:

$$OC = \frac{t_{com}}{t_{calc}}$$

# Esempio: calcolo della somma di n=16 numeri

27

supponiamo  
 $T(1)=15t_{\text{calc}}$

Trascurando la comunicazione

p	$T_p$	$S_P$	$E_p$
2	$8t_{\text{calc}}$	1.88	0.94
4	$5t_{\text{calc}}$	3.00	0.75
8	$4t_{\text{calc}}$	3.75	0.47

$$p=2^k \quad T(p) = \left(\frac{n}{p} - 1 + \log_2 p\right) t_{\text{calc}}$$

$$OC = \frac{t_{\text{com}}}{t_{\text{calc}}} = 2$$

Considerando la comunicazione

comunicazioni

p	$T_p$	$S_P$	$E_p$
2	$(8+2\cdot 1)t_{\text{calc}}$	1.50	0.75
4	$(5+2\cdot 2)t_{\text{calc}}$	1.67	0.42
8	$(4+2\cdot 3)t_{\text{calc}}$	1.88	0.24

$$p=2^k \quad T(p) = \left(\frac{n}{p} - 1 + \log_2 p\right) t_{\text{calc}} + (\log_2 p) t_{\text{com}}$$

# Overhead di comunicazione

28

$t_{com}$  = tempo di comunicazione di 1 dato

$t_{calc}$  = tempo di esecuzione di 1 operazione fl.p.

definiamo

Overhead di Comunicazione Totale

il rapporto:

$$OC_p = \frac{\tau_{com}(p)}{\tau_{calc}(p)}$$

# Overhead di comunicazione

29

$t_{com}$  = tempo di comunicazione di 1 dato

$t_{calc}$  = tempo di esecuzione di 1 operazione fl.p.

definiamo

## Overhead di Comunicazione Totale

il rapporto:

$$OC_p = \frac{\tau_{com}(p)}{\tau_{calc}(p)}$$

Fornisce una misura del “peso” della comunicazione sul tempo di esecuzione dell’ algoritmo

# Esempio: calcolo della somma di n=16 numeri

30

$$OC = \frac{t_{com}}{t_{calc}} = 2$$

Posto ad esempio:

p	$T_{calc}(p)$	$T_{com}(p)$	$OC_p$
2	$8t_{calc}$	$1t_{com}$	0.25
4	$5t_{calc}$	$2t_{com}$	0.80
8	$4t_{calc}$	$3t_{com}$	1.50

Su 8 processori il tempo di comunicazione pesa di più rispetto al tempo di esecuzione

# Valutazione dell'efficienza di algoritmi e software in ambiente parallelo

parte 2-1

a.a. 2021/2022

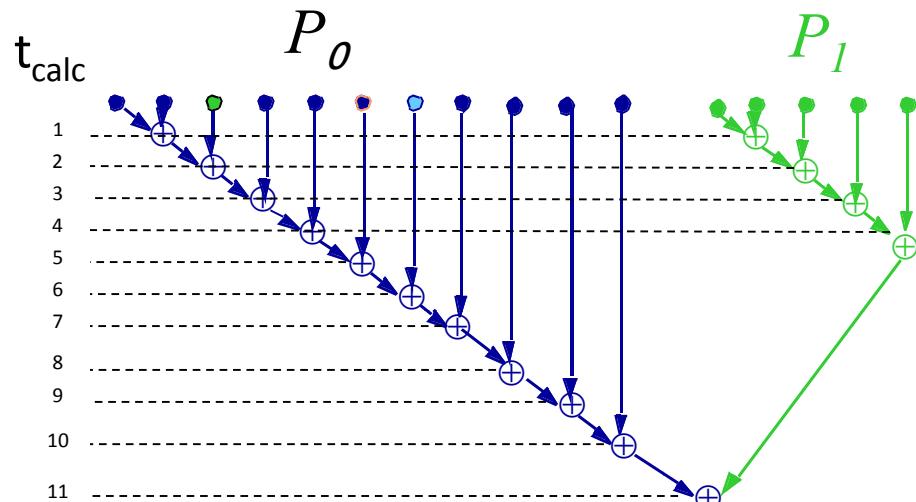


# **Il problema del bilanciamento**

Esempio 1:  $n=16$  e  $p=2$

numero di addendi di  $P_0$  : 11

numero di addendi di  $P_1$  : 5



$$T(2)=11 \ t_{\text{calc}}$$

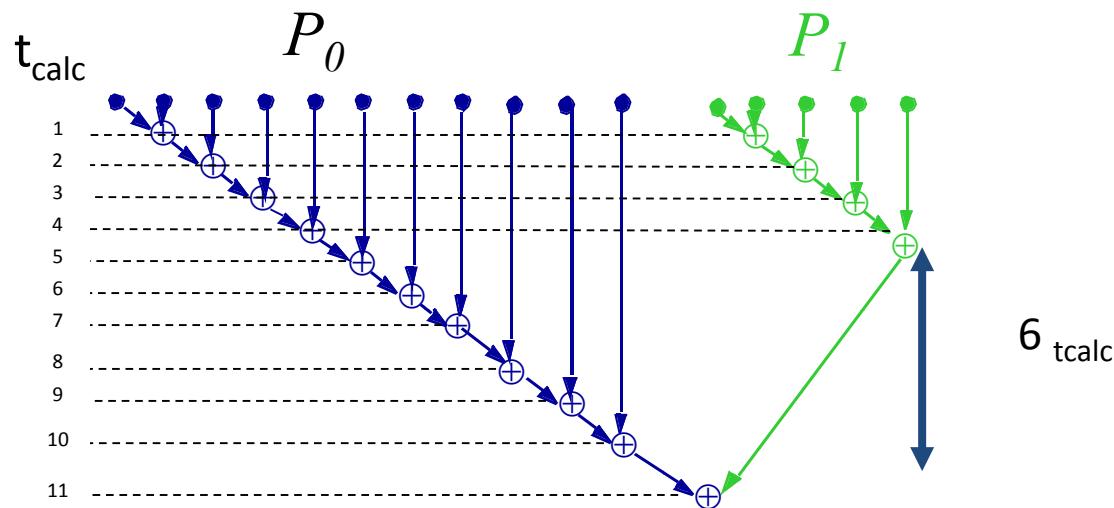
Passi temporali da 1 a 4: 4  
addizioni (eseguite  
concorrentemente da 2  
processori)

Passi temporali da 5 a 11: 7  
addizioni (eseguite  
sequenzialmente da un solo  
processore)

Esempio 1:  $n=16$  e  $p=2$

numero di addendi di  $P_0$  : 11

numero di addendi di  $P_1$  : 5

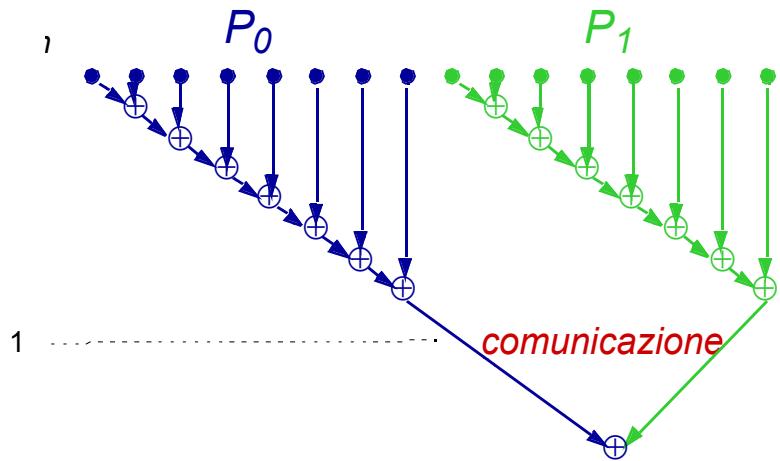


$$T(2) = 11t_{calc} + 1t_{com}$$

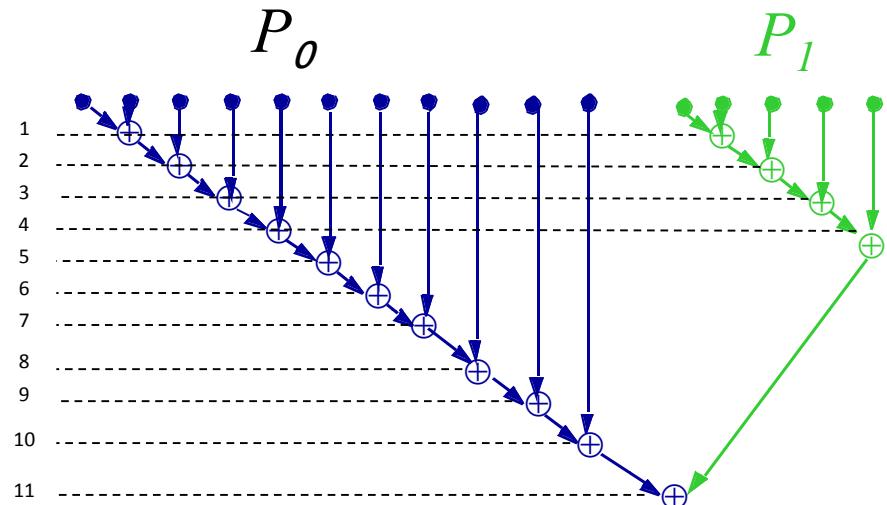
di cui 6  $t_{calc}$  sono di “sincronizzazione”

Tempo di attesa  
necessario per  
Coordinare  
il lavoro tra i  
processori

## Confronto: n=16 e p=2



$$T(2) = \boxed{8t_{calc}} + 1t_{com}$$



$$T(2) = \boxed{11t_{calc}} + 1t_{com}$$

## Osservazione 1

L'algoritmo precedente non e'  
bilanciato



Una cattiva ripartizione del carico di lavoro tra i processori induce un tempo di attesa e  
quindi un aumento del tempo di esecuzione



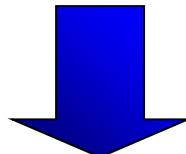
Il tempo di esecuzione può dipendere dal  
bilanciamento del carico computazionale tra i  
processori

# In generale

Il tempo di esecuzione di un algoritmo parallelo, distribuito su  $p$  processori, comprende 3 componenti:

- $T_s$  tempo per eseguire la parte seriale
- $T_c/p$  tempo per eseguire la parte parallela
- $T_0$  costo di comunicazione *e sincronizzazione* con

$$T_0(p) \geq 0, p > 1$$



$$T(p) = T_s + \frac{T_c}{p} + T_0(p)$$

## In conclusione

La valutazione delle prestazioni di un algoritmo parallelo deve tener conto di:

- Numero di processori
- Tempo di calcolo
- Tempo di comunicazione
- Bilanciamento del carico computazionale
- Tempo di sincronizzazione
- ...



Necessità di utilizzare più parametri di valutazione  
 $(T(p), S(p), E(p), Oc(p), \dots)$

## In conclusione

Inoltre, differenti caratteristiche hw/sw dei calcolatori paralleli:

- Numero di processori
- Architettura e potenza dei processori
- Tipo e numero di memorie
- Connessione tra i processori
- Connessione tra processori e memorie
- Omogeneità/eterogeneità del sistema
- Software message-passing
- ...



Una valutazione effettiva delle prestazioni di un algoritmo richiede  
l'implementazione in uno specifico ambiente di calcolo e la misura di  $T(p)$

# Scelta algoritmo per misurare $T(1)$

## Domanda

Lo speed-up

$$S(p) = \frac{T(1)}{T(p)}$$

misura

la riduzione del tempo di esecuzione dell'algoritmo sequenziale  
rispetto al tempo di esecuzione  
dell' algoritmo parallelo

Quale algoritmo scegliere per misurare  $T(1)$  ?

## PRIMA SCELTA

$T(1)$  = tempo di esecuzione dell'algoritmo parallelo  
su 1 processore



$S(p)$  dà informazioni su quanto l'algoritmo si presta  
all'implementazione su un'architettura parallela

Svantaggio:

l'algoritmo parallelo su 1 processore potrebbe eseguire più  
operazioni del necessario

## SECONDA SCELTA

$T(1)$  = tempo di esecuzione del migliore algoritmo sequenziale



$S(p)$  dà informazioni sulla riduzione effettiva del tempo nella risoluzione di un problema con  $p$  processori

### Difficoltà:

- individuazione del “miglior” algoritmo sequenziale
- disponibilità di software che implementa tale algoritmo

Fine Lezione



# VALUTAZIONE DELLE **PRESTAZIONI** DI UN CALCOLATORE





# PRESTAZIONI DI UN CALCOLATORE

Materiale tratto da lezioni di *Calcolo Parallelo e Distribuito* tenute da A. Murli e dal testo A. Murli - *Lezioni di calcolo parallelo*



# COSA È UN CALCOLATORE?

Un calcolatore è un esecutore di algoritmi!



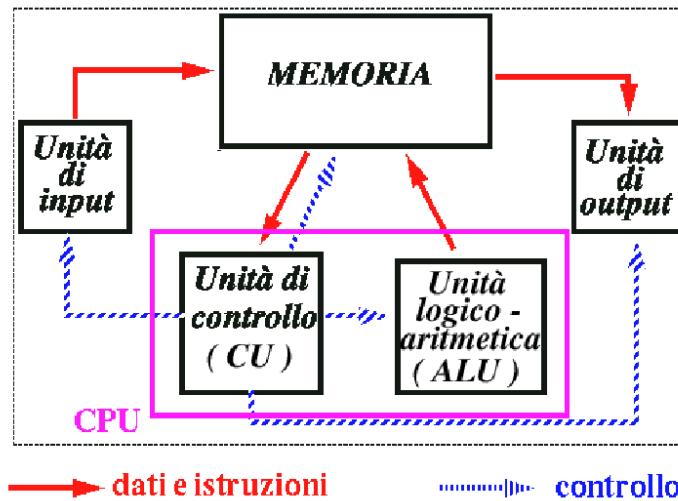
## COSA È UN ALGORITMO?

Un algoritmo è un procedimento per la risoluzione di una classe di problemi, costituito da un insieme finito di direttive non ambigue che specificano una sequenza di operazioni da eseguire su un insieme finito di dati e istruzioni.



# COSA È UN CALCOLATORE?

Il calcolatore che prendiamo in considerazione è quello schematizzato da Von Neumann nel 1945

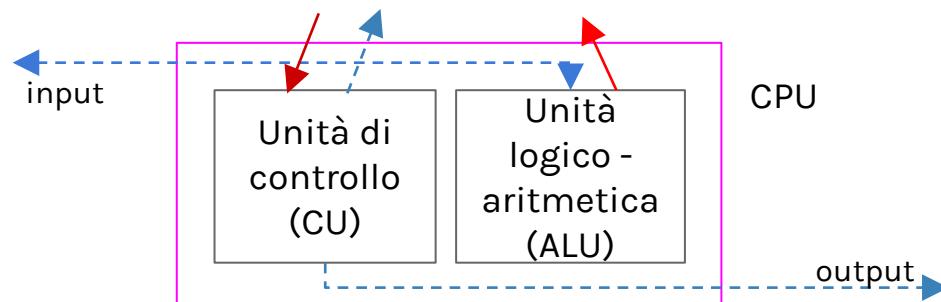




# DA COSA È COMPOSTO UN CALCOLATORE?

Esso è composto:

- ▶ CPU: è l'esecutore delle operazioni (programma);
  - ▷ Arithmetic Logic Unit (ALU)
  - ▷ Control Unit (CU)





# DA COSA È COMPOSTO UN CALCOLATORE?

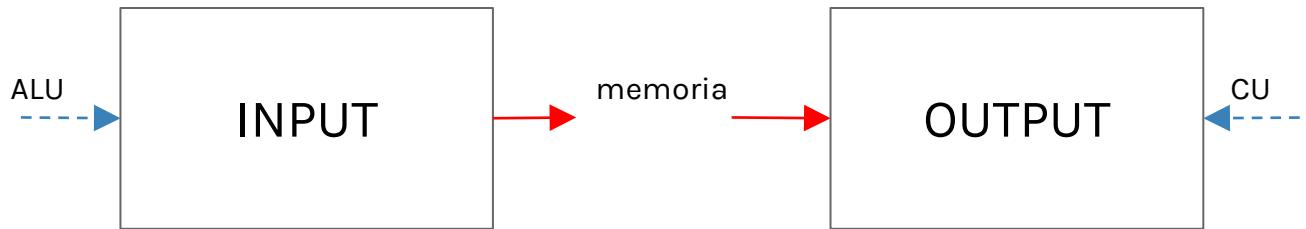
- ▶ Memoria: E' un supporto fisico dove vengono immagazzinate le informazioni (istruzioni + dati);





# DA COSA È COMPOSTO UN CALCOLATORE?

- ▶ I/O: E' l'interfaccia per interagire con l'esterno.





# COSA INTENDIAMO PER PRESTAZIONE?



# ESEMPIO

Aeroplano	Capacità passeggeri	Velocità di crociera (Km/H)	Autonomia (Km)
<i>Boeing 777</i>	375	980	7400
<i>Boeing 747</i>	470	980	6640
<i>BAC/Sud Concorde</i>	132	2160	6400
<i>Douglas DC-8-50</i>	146	870	13950

Scegliamo l'aereo che ha le migliori prestazioni!



# L'AEREO CON LE MIGLIORI PRESTAZIONI?

Se per prestazione intendo la capacità di trasportare un singolo passeggero da un luogo ad un altro nel minor tempo possibile (quindi considero il mezzo più veloce) allora la scelta migliore è il **Concorde**

Aeroplano	Velocità di crociera (Km/H)
<i>Boeing 777</i>	980
<i>Boeing 747</i>	980
<i>BAC/Sud Concorde</i>	2160
<i>Douglas DC-8-50</i>	870



## L'AEREO CON LE MIGLIORI PRESTAZIONI?

Se per prestazione intendo la capacità di trasportare un singolo passeggero da un luogo ad un altro nel minor tempo possibile (quindi considero il mezzo più veloce) allora la scelta migliore è il **Concorde**





# L'AEREO CON LE MIGLIORI PRESTAZIONI?

Se invece intendo l'aereo con il maggior numero di passeggeri allora il migliore sarà il **Boeing 747**

Aeroplano	Capacità passeggeri
<i>Boeing 777</i>	375
<i>Boeing 747</i>	470
<i>BAC/Sud Concorde</i>	132
<i>Douglas DC-8-50</i>	146

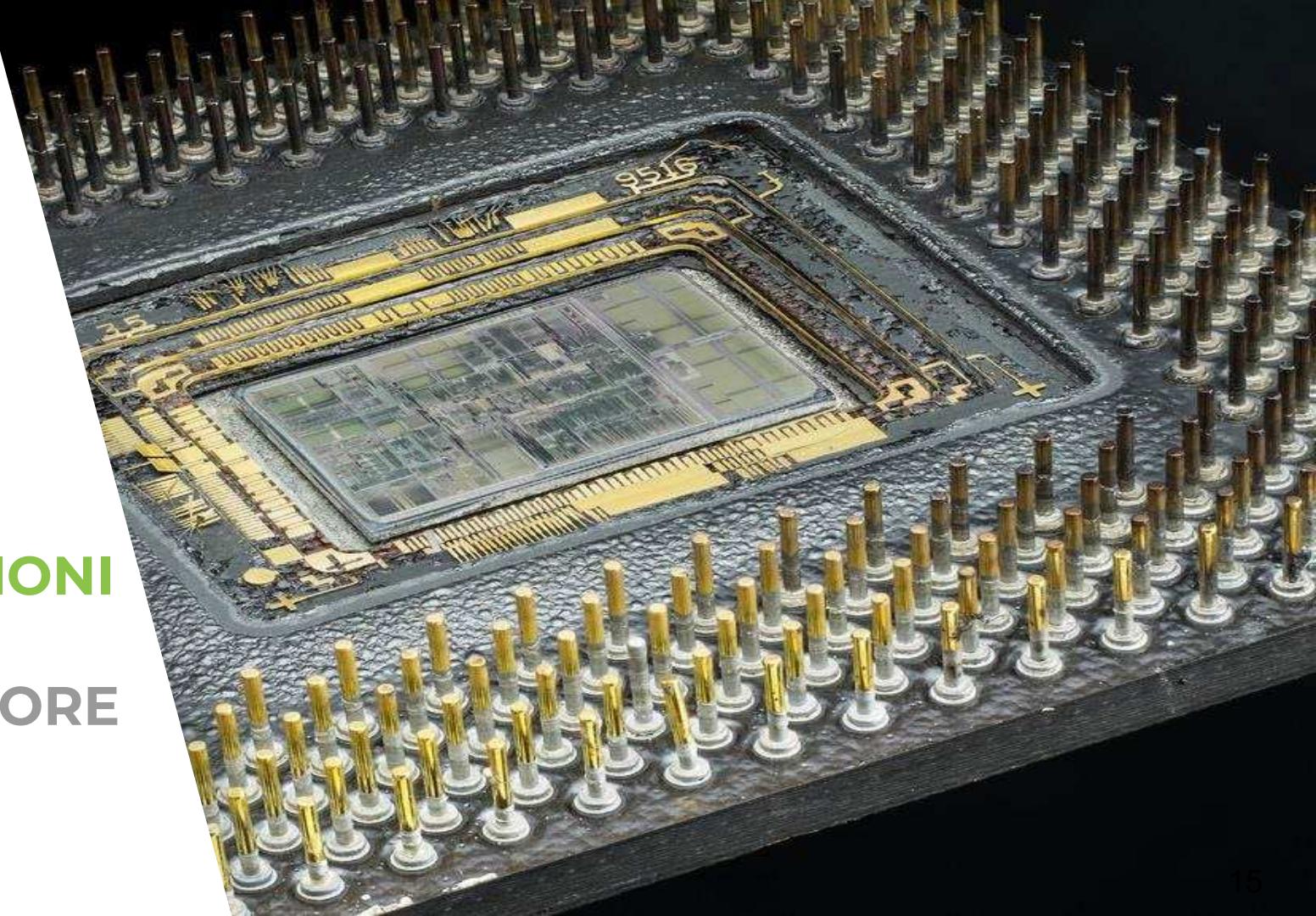


## L'AEREO CON LE MIGLIORI PRESTAZIONI?

Se invece intendo l'aereo con il maggior numero di passeggeri allora il migliore sarà il **Boeing 747**



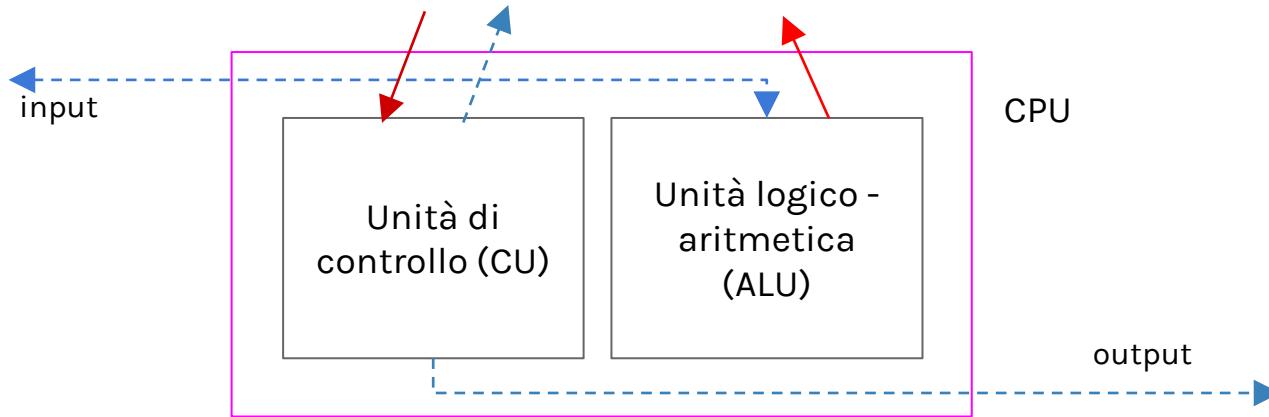
# PRESTAZIONI DI UN PROCESSORE (CPU)





## PRESTAZIONI DI UN PROCESSORE (CPU)

Concentriamoci adesso solo sulla componente CPU ed analizziamo i fattori importanti al fine della valutazione delle prestazioni





# COSA INTENDIAMO PER PRESTAZIONI DI UN PROCESSORE?

La prestazione di un processore è la capacità di eseguire uno specifico compito in un determinato tempo.



## PRESTAZIONI DI UN PROCESSORE

Le prestazioni di un processore possono essere valutate come:

- ▶ Il tempo impiegato a fornire una risposta ad un compito assegnato (tempo di risposta);

Quindi decidiamo di valutare le prestazioni in termini di **TEMPO DI RISPOSTA**



## PRESTAZIONI DI UN PROCESSORE

In questo contesto valutiamo le prestazioni in termini di tempo di risposta o **CPU TIME** che è il tempo che la CPU impiega per effettuare le operazioni ad un compito assegnato.



DA COSA DIPENDE ?

**IL CPU TIME DIPENDE  
DALLE  
CARATTERISTICHE  
DEL PROCESSORE**

Cosa caratterizza un processore?



# CARATTERISTICHE DI UN PROCESSORE

Tra i dati forniti dal costruttore c'è IL CICLO DI  
**CLOCK**



## CLOCK

L'unità di misura base del processore è il **CLOCK**,  
definibile come orologio del calcolatore.

Più in dettaglio...

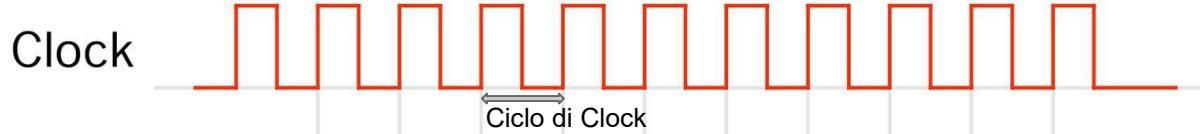
“

UN DISPOSITIVO SINCRONO CHE  
PRODUCE IMPULSI AD INTERVALLI  
REGOLARI.



## CICLO DI CLOCK

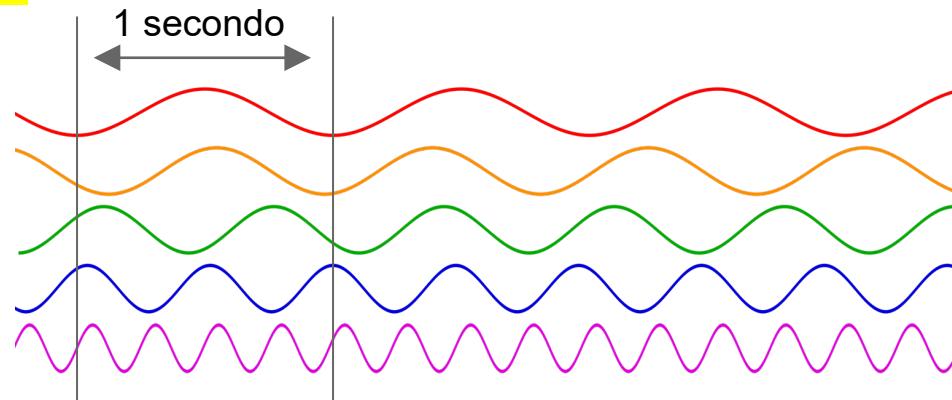
Anche detto periodo di clock, è la misura del tempo che trascorre tra due colpi di clock successivi





## FREQUENZA

La frequenza di clock è il numero di colpi di clock che vengono eseguiti in una determinata unità di tempo, si misura in Hertz (numero di cicli di clock eseguiti in un secondo).



# TIPOLOGIE DI PRESTAZIONE





## TIPOLOGIE DI PRESTAZIONE

- ▶ Peak Performance
- ▶ Prestazione sostenuta (o actual)



## COME SI MISURANO?

**FLOPS** (FLoating-point Operations Per Second) e rappresenta una misura del numero di operazioni floating-point al secondo che un processore esegue.



# COME SI CALCOLA LA PEAK PERFORMANCE

Definiamo:

$N_c$  è il massimo numero di operazioni floating point che possono essere eseguite in un ciclo di clock.

$T_c$  è il ciclo di clock. La Peak Performance  $P_{max}$  risulterà:

$$P_{max} = \frac{N_c}{T_c}$$



## PEAK PERFORMANCE

La Peak Performance indica il numero massimo di operazioni floating point (virgola mobile) che possono essere eseguite dal processore in una unità di tempo (generalmente il secondo).



## FREQUENZA

La frequenza di clock può essere espressa anche in funzione del tempo come reciproco del ciclo di clock:

$$f = \frac{1}{T_c}$$



# COME SI CALCOLA LA PEAK PERFORMANCE

E' possibile calcolare la Peak Performance come:

$$P_{max} = N_c \times f$$

Generalmente il costruttore mette a disposizione questa misura.



# ESEMPIO:

Cerchiamo la **peak performance** del processore Intel Pentium 4.



Intel® Pentium® 4 Processor 511  
(1M Cache, 2.80 GHz, 533 MHz FSB)

Specifiche	
Informazioni di base	<b>Numero del processore</b> 511
Prestazioni	<b>Stato</b> Launched
Informazioni supplementari	<b>Data di lancio</b> Q4'05
Specifiche package	<b>Litografia</b> 90 nm
Tecnologie evolute	<b>Prezzo consigliato per il cliente</b> N/A
Tecnologia Intel® Platform Protection	
Prodotti compatibili	
Immagini prodotto	
Ordinazione / specifiche / istruzioni	
Download di driver	

Specifiche	
<b>Informazioni di base</b>	
Numero del processore	511
Stato	Launched
Data di lancio	Q4'05
Litografia	90 nm
Prezzo consigliato per il cliente	N/A
<b>Prestazioni</b>	
Numero di core	1
Frequenza base del processore	2.80 GHz
Cache	1 MB L2
Velocità del bus	533 MHz FSB
TDP	84 W
Intervallo di tensione VID	1.25V-1.400V



# ESEMPIO:

Il costruttore ci dice che i dati di questo processore sono:

- ▶  $f=2.8$  GHz
- ▶ Peak Performance = 5.6 Ghz

Intel® Pentium® 4 Processor 500 Series							
Processor Number	Frequency Type	Clock GHz	CTP	GFLOP	APP 1-way	APP 2-way	APP 4-way
505	Base	2.66	10184	5.332	0.0015996	0.0031992	0.0063984
	Single Core Max Turbo	N/A	N/A	N/A	N/A	N/A	N/A
	GPU ONLY	N/A	N/A	N/A	N/A	N/A	N/A
506	Base	2.66	10184	5.332	0.0015996	0.0031992	0.0063984
	Single Core Max Turbo	N/A	N/A	N/A	N/A	N/A	N/A
	GPU ONLY	N/A	N/A	N/A	N/A	N/A	N/A
511	Base	2.8	10696	5.6	0.00168	0.00336	0.00672
	Single Core Max Turbo	N/A	N/A	N/A	N/A	N/A	N/A
	GPU ONLY	N/A	N/A	N/A	N/A	N/A	N/A



# PRESTAZIONI DI VARI PROCESSORI

CPU	f	Numero operazioni per ciclo di clock	Peak Performance
Intel R Xeon Phi Coprocessor 3120A (Knights Corner)	1.1 GHz	16	17,6 Gflop/s
Intel i5-3470s (SandyBridge)	2.9 GHz	8	23,2 Gflop/s
Intel Core 2 Quad Q8300			
Intel R Xeon R Processor X5560 (nehalem)			

**ESERCIZIO:** Trovare la Peak Performance ed il numero di operazioni per ciclo di clock per l'INTEL Core 2 Quad Q8300 e dell' X5560

ref: <http://www.intel.com/content/www/us/en/support/processors/000005755.html>



## PRESTAZIONE SOSTENUTA

La prestazione sostenuta (o effettiva) è il numero di **FLOPS** che un calcolatore riesce ad eseguire, fissato un algoritmo.



## COMPLESSITÀ DI TEMPO

Per la valutazione della prestazione sostenuta occorre introdurre la complessità di tempo dell'algoritmo che desideriamo eseguire,  $T(N)$ . Essa rappresenta il numero di operazioni eseguite e dipende dalla dimensione dell'input.



# ESEMPI DI COMPLESSITÀ

ALGORITMO	$T(n)$
somma di n numeri	$n - 1$
ricerca del massimo in un array	$n - 1$
fusione di due array ordinati	$n + m$
prodotto scalare	$n + (n-1)$



## CALCOLO DELLA PRESTAZIONE SOSTENUTA

Detti  $T(n)$  la complessità di tempo e  $D$  il tempo di esecuzione dell'algoritmo, esprimiamo la prestazione sostenuta come:

$$SP = \frac{T(n)}{D}$$

misurata in FLOPS.



## ESERCIZIO

Scrivere un algoritmo in C:

- ▶ che calcoli la somma di N numeri
- ▶ calcoli la prestazione sostenuta (solo del blocco  
in cui viene effettuata la somma)



# ESERCIZIO

Scrivere un algoritmo in C:

- ▶ che calcoli il prodotto matrice vettore

$$y = A * x$$

- ▶ dove x e y sono due vettori di dimensione n, A è una matrice di dimensione m,n.
- ▶ calcoli la prestazione sostenuta (solo del blocco in cui vengono effettuate le operazioni)



# ESERCIZIO

Scrivere un algoritmo in C:

- ▶ che calcoli il prodotto matrice matrice

$$C = A * B$$

- ▶ dove A,B,C sono matrici di dimensione n,m;
- ▶ calcoli la prestazione sostenuta (solo del blocco in cui vengono effettuate le operazioni)



## RICAPITOLANDO

- ▶ Calcolare la peak performance del processore del proprio PC
- ▶ Scrivere un programma che calcoli il prodotto matrice vettore e che valuti la performance sostenuta
- ▶ Confrontare i risultati ottenuti con la peak performance



# COME PRENDERE I TEMPI?

```
#include <sys/time.h>
//main
.
.
.
gettimeofday(... , ...);
{ ..... }
gettimeofday(... , ...);
//end
```



Il blocco che ci interessa valutare



# COME COMPILARE IL SORGENTE?

Il comando per la compilazione è:

```
$gcc -o nome_eseguibile nome_sorgente.c
```



## CHE COSA È UN'ISTRUZIONE?

Il linguaggio direttamente comprensibile da un calcolatore è il LINGUAGGIO MACCHINA, esso è costituito dai simboli 0,1. Quindi un'istruzione è un comando eseguito da un processore in linguaggio macchina.



# COMPOSIZIONE DI UNA ISTRUZIONE

Le istruzioni contengono principalmente:

- ▶ Quale operazione eseguire;
- ▶ le locazioni di memoria che contengono gli operandi.



# COMPOSIZIONE DI UNA ISTRUZIONE

Ad esempio, una istruzione può essere memorizzata in questo modo:

CODICE OPERATIVO	INDIRIZZO OPERANDO 1	INDIRIZZO OPERANDO 2	INDIRIZZO RISULTATO
------------------	----------------------	----------------------	---------------------

ES. Esecuzione di  $2 \times 3 = 6$

moltiplicazione	indirizzo di 2	indirizzo di 3	indirizzo di 6
0111	1000	1001	1011



## ESEMPIO

Desideriamo eseguire l'operazione:

$$2 \times 3 + 7$$

Analizziamone le fasi al livello delle istruzioni eseguite dal processore.



# ESEMPIO

INDIRIZZI

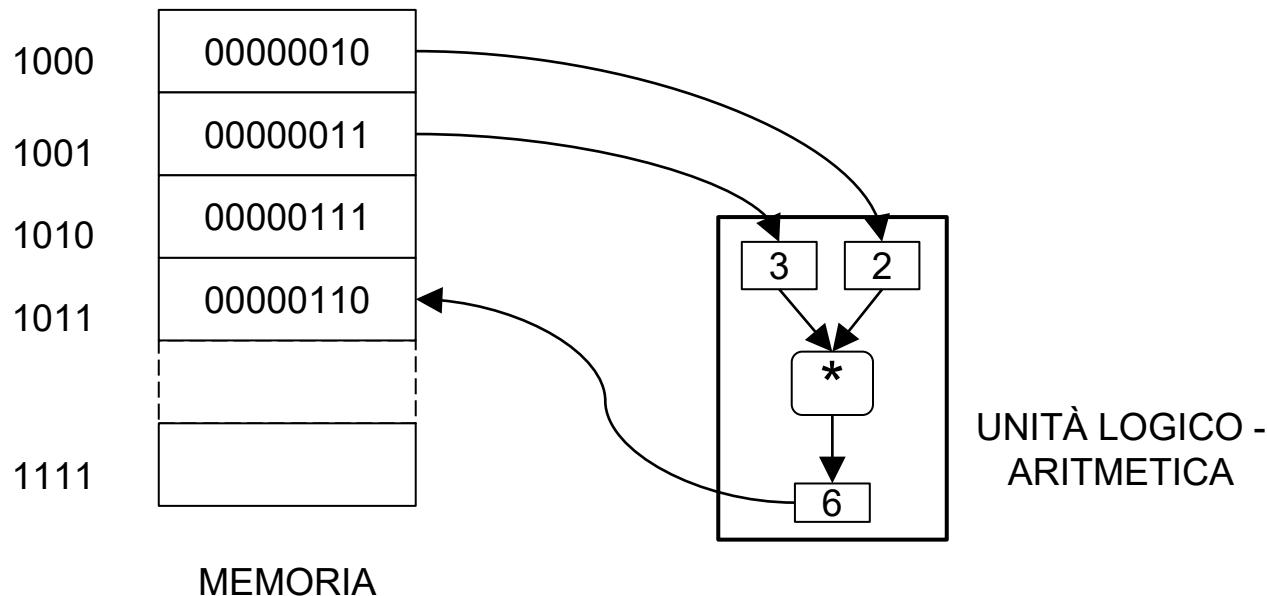
1000	00000010	2
1001	00000011	3
1010	00000111	7
1011		
1111		

MEMORIA



# ESEMPIO

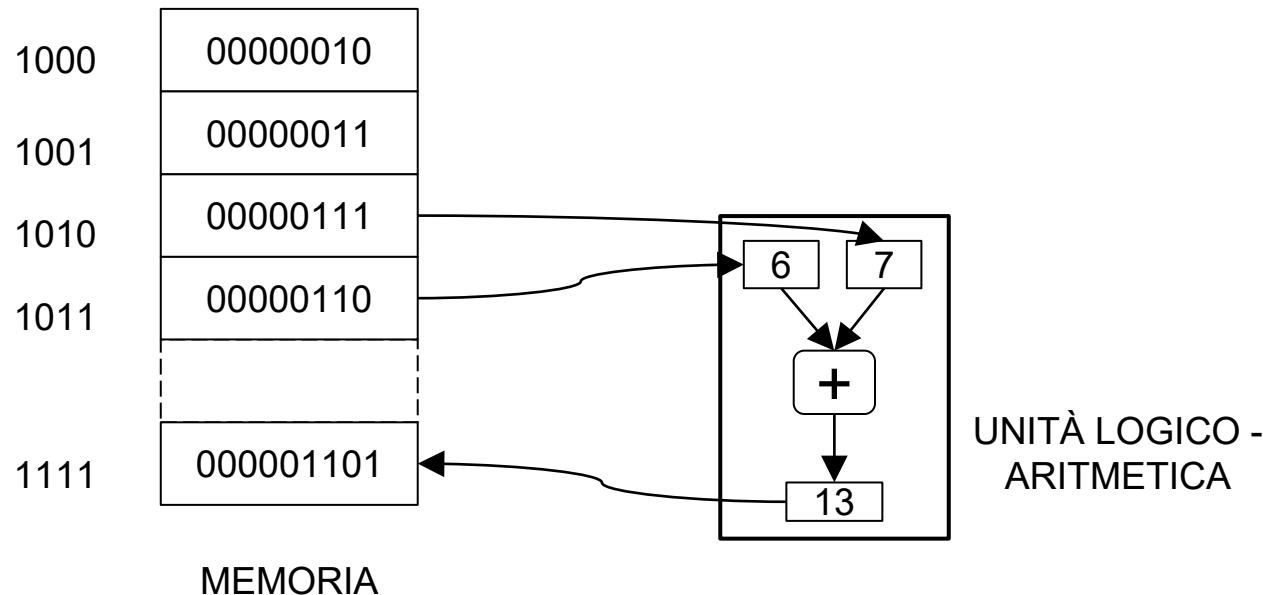
1° FASE: Calcolo di  $2 \times 3$  e memorizzazione del risultato nella locazione di memoria con indirizzo 1011





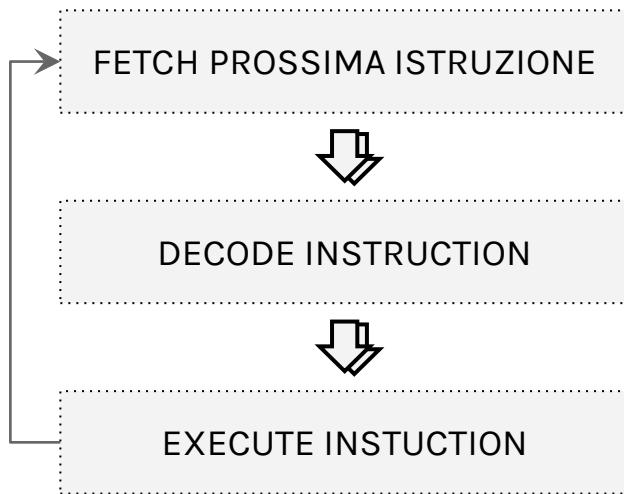
## ESEMPIO

2° FASE: calcolo di  $6 + 7$  e memorizzazione del risultato nella locazione di memoria con indirizzo 1111





# CICLO DI ESECUZIONE DI ISTRUZIONI DELLA CPU



Possiamo schematizzare il ciclo di istruzioni eseguito dalla CPU con l'**Instruction Fetch Decode Cycle**



# CICLO DI ESECUZIONE DELLE ISTRUZIONI

Instruction Fetch Execute Cycle descrive le fasi che un processore generalmente effettua per l'esecuzione del ciclo di istruzioni

- ▶ Fetch Cycle
- ▶ Decode Cycle
- ▶ Execute Cycle



## FETCH CYCLE

La CPU estrae le istruzioni da eseguire dalla memoria e le prepara per l'esecuzione.



## DECODE CYCLE

In questa fase si **decodifica** l'istruzione estratta al passo precedente. Vengono estratti gli **operandi** per eseguire l'istruzione.



## EXECUTE CYCLE

In questa fase la CPU **esegue** le istruzioni che sono state estratte e decodificate. Alla fine il risultato viene **salvato** in memoria.



## COMPILAZIONE CON LIVELLO 0

Compiliamo il sorgente prima in questo modo:

```
$gcc -o nome_eseguibile nome_sorgente.c
```

e successivamente in questo modo:

```
$gcc -o nome_eseguibile -O1 nome_sorgente.c
```

L'opzione -O1 attiva il livello 1 di ottimizzazione del compilatore gcc

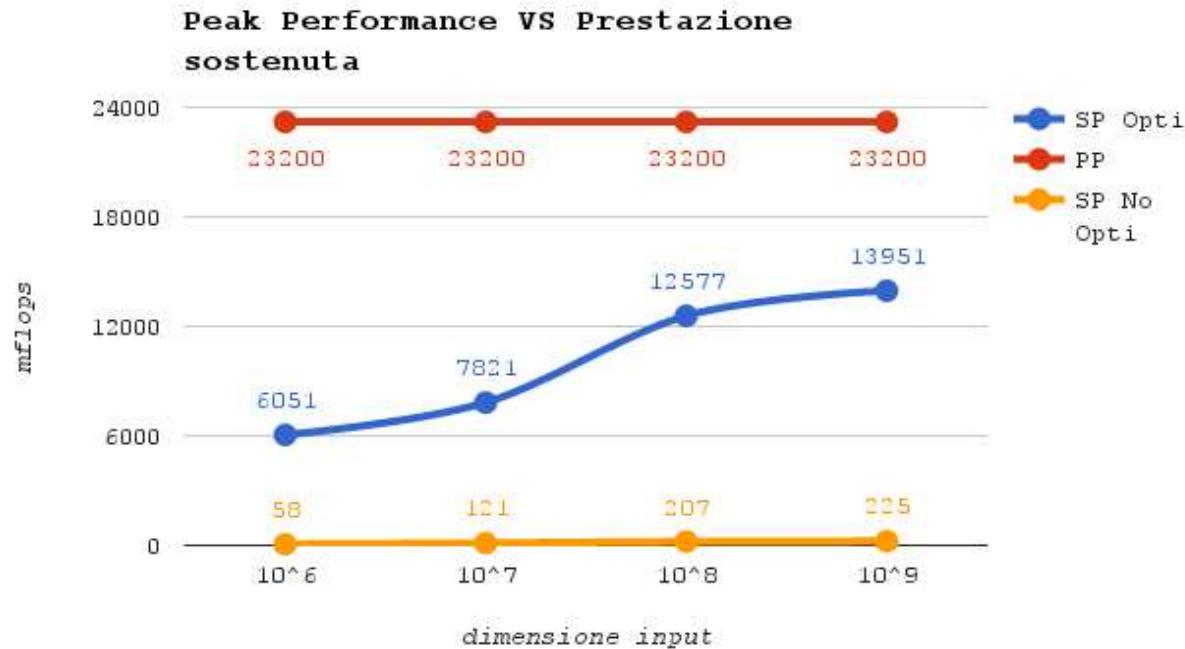


# COSA OTTENIAMO (processore Intel i5 -3470s)?

INPUT	Tempo di esecuzione senza O (secondi)	MFLOPS	Tempo di esecuzione (secondi)	MFLOPS	Peak Performance (MFLOPS)
$10^6$	0.017	58	0.004660	6051	<b>23200</b>
$10^7$	0.082	121	0.028827	7821	<b>23200</b>
$10^8$	0.482	207	0.257179	12577	<b>23200</b>
$10^9$	4.440	225	5.485245	13951	<b>23200</b>



# CONFRONTO PEAK - SOSTENUTA





# QUALI SONO I FATTORI CHE INFLUENZANO LE PRESTAZIONI?

Fissato l'algoritmo i fattori sono:

- ▶ Processore e tecnologia hardware;
- ▶ Set di istruzioni;
- ▶ Compilatore.

Quindi gli elementi da tenere in considerazione sono molteplici!