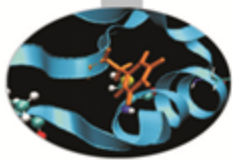
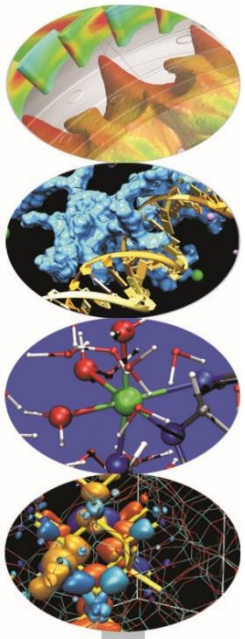
CINECA SCAI
SuperComputing Applications and Innovation



Calcolo Parallelo con MPI

(3° parte)

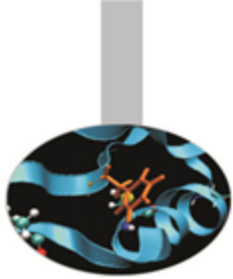


Claudia Truini
c.truini@cineca.it

Mariella Ippolito
m.ippolito@cineca.it

CINECA

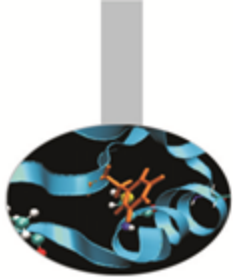
Calcolo parallelo con MPI (3° parte)



Introduzione alle comunicazioni collettive

Laboratorio n° 3

Pattern di comunicazione



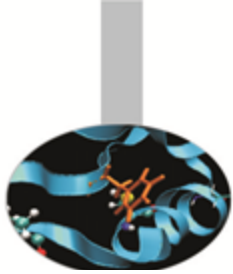
- Nella parallelizzazione di programmi reali, alcuni schemi di invio/ricezione del messaggio sono largamente diffusi:

  pattern di comunicazione

- I pattern di comunicazione possono essere di tipo
  - *point-to-point* , coinvolgono solo due processi
  - collettivi, coinvolgono più processi

- MPI mette a disposizione strumenti (funzioni MPI) per implementare alcuni pattern di comunicazione in modo corretto, robusto e semplice
  - il corretto funzionamento NON deve dipendere dal numero di processi

Introduzione alle comunicazioni collettive



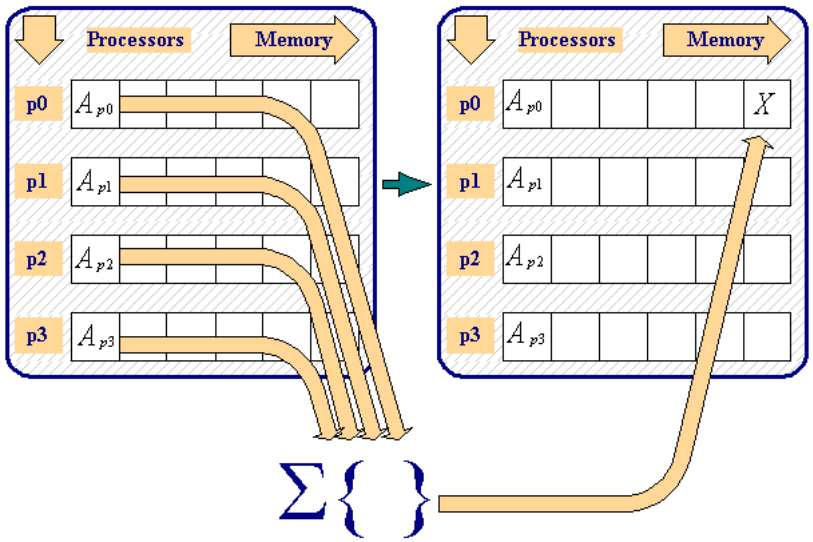
- Alcuni pattern di comunicazione prevedono il coinvolgimento di tutti i processi di un comunicatore
  - Esempio: il calcolo della somma dei contributi parziali dell'integrale per il calcolo del π

```
if (rank != 0) {

    /* slave processes send partial pi sum to master process 0 */
    MPI_Send(&pi, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);

} else {

    for (from = 1; from < size; from++) {

        printf("I have pi = %f\n", pi);

        /* master process receives partial pi sum from other processes */
            MPI_Recv(&sum, 1, MPI_DOUBLE, from, tag, MPI_COMM_WORLD, &status);

        printf(".. received %f from proc %d\n", sum, from);
            pi = pi + sum;

        fflush(stdout);
    }
}
```

- MPI mette a disposizione alcune funzioni che implementano questi pattern
  - Si evita così al programmatore l'onere e la complicazione di dover programmare questi pattern a partire da comunicazioni *point-to-point*
  - Sono implementati con gli algoritmi più efficaci

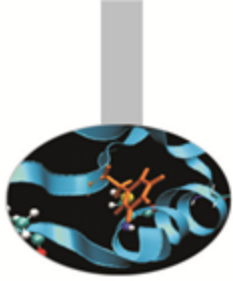
- È possibile catalogare queste funzioni, sulla base del/dei *sender* e del/dei *receiver*, in tre classi: ***all-to-one***, ***one-to-all***, ***all-to-all***.
  La divisione in classi ci permette di trovare facilmente la funzione cercata

REDUCE



- L'operazione di *REDUCE* consente di:
  - Raccogliere da ogni processo i dati provenienti dal *send buffer*
  - Ridurre i dati ad un solo valore attraverso un operatore (la somma in figura)
  - Salvare il risultato nel *receive buffer* del processo di destinazione, chiamato convenzionalmente *root* (p0 in figura)
- Appartiene alla classe *all-to-one*

Binding di MPI_Reduce



In C

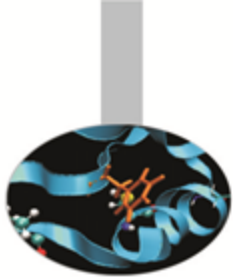
```
int MPI_Reduce(void* sbuf, void* rbuf, int count,
  MPI_Datatype dtype, MPI_Op op, int root, MPI_Comm comm)
```

In Fortran

```
MPI_REDUCE(SBUF, RBUF, COUNT, DTYPE, OP, ROOT, COMM, ERR)
```

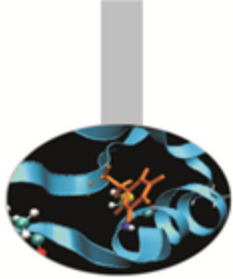
- [IN] **sbuf** è l'indirizzo del *send* buffer
- [OUT] **rbuf** è l'indirizzo del *receive* buffer
- [IN] **count** è di tipo `int` e contiene il numero di elementi del *send/receive* buffer
- [IN] **dtype** è di tipo `MPI_Datatype` e descrive il tipo di ogni elemento del *send/receive* buffer
- [IN] **op** è di tipo `MPI_Op` e referenzia l'operatore di reduce da utilizzare
- [IN] **root** è di tipo `int` e contiene il *rank* del processo *root* della reduce
- [IN] **comm** è di tipo `MPI_Comm` ed è il comunicatore cui appartengono i processi coinvolti nella reduce

Operatori di *Reduce*

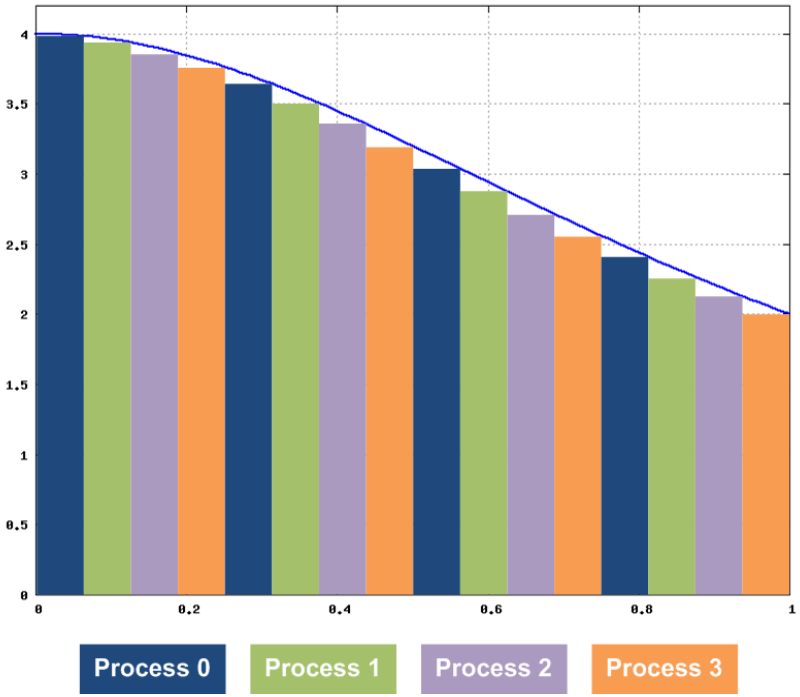


- Le principali operazioni di reduce predefinite sono
  - Massimo (MPI_MAX)
  - Minimo (MPI_MIN)
  - Somma (MPI_SUM)
  - Prodotto (MPI_PROD)
  - operazioni logiche (MPI_LAND, MPI_LOR, MPI_LXOR)
  - operazioni *bitwise* (MPI_BAND, MPI_BOR, MPI_BXOR)
- Gli operatori di reduce sono associativi e commutativi (almeno nella versione a precisione infinita)
- L'utente può definire operatori *ad-hoc* (MPI_Op_create)

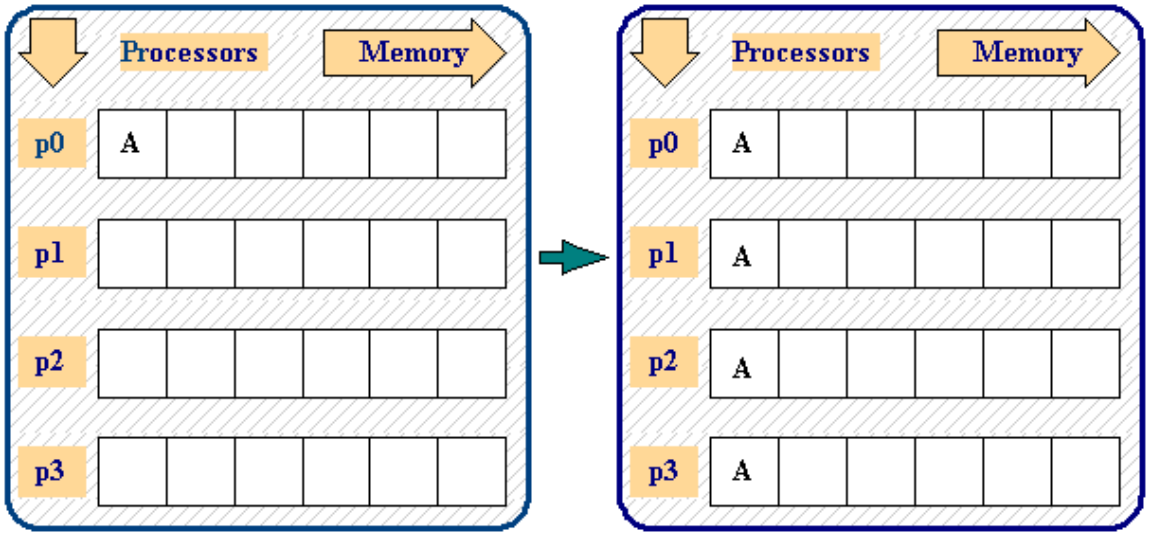
Calcolo di π con *reduce*



```c
#include <stdio.h>
#include "mpi.h"
#define INTERVALS 10000

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

    int rank, nprocs, tag;
    int i;
    int interval = INTERVALS;
    double x, dx, f, sum, pi;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    sum = 0.0; dx = 1.0 / (double) interval;

    /* each process computes integral */
    for (i = rank; i < interval; i = i+nprocs) {
        x = dx * ((double) (i - 0.5));
        f = 4.0 / (1.0 + x*x);
        sum = sum + f;
    }
    pi = dx*sum;
    sum = pi; /* using variable sum as sending buffer */

    MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0)
        printf("Computed PI %.24f\n", pi);

    /* Quit */
    MPI_Finalize();
    return 0;
}
```



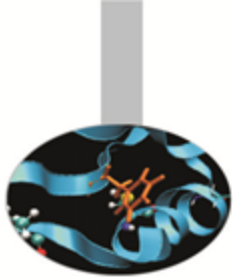
BROADCAST



- La funzionalità di *BROADCAST* consente di copiare dati dal *send* buffer del processo *root* (p0 nella figura) al *receive* buffer di tutti gli altri processi appartenenti al comunicatore utilizzato (processo *root* incluso)

- Appartiene alla classe *one-to-all*

Binding di `MPI_Bcast`



In C

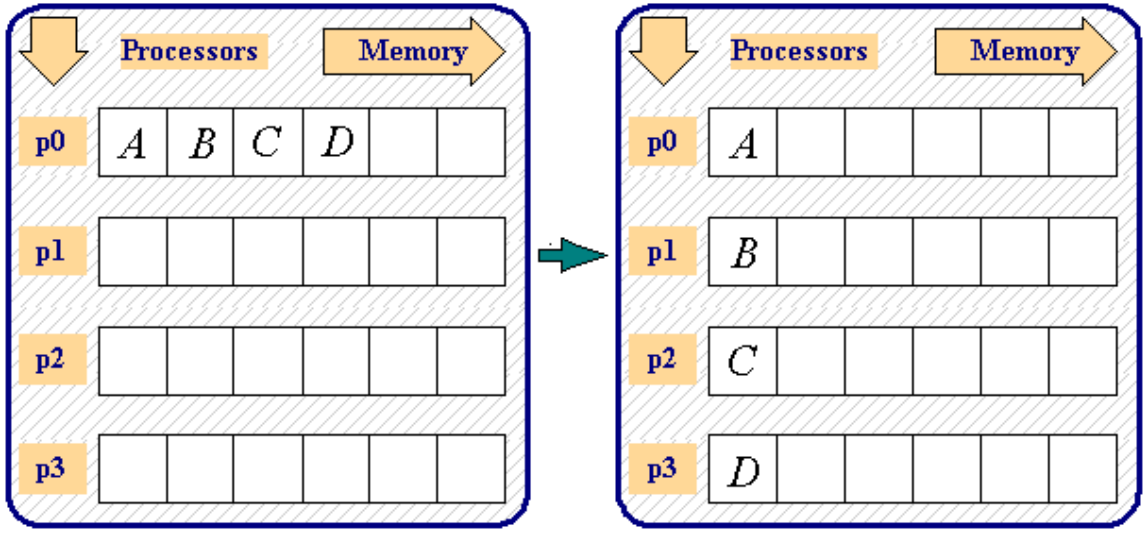
```
int MPI_Bcast(void* buf, int count, MPI_Datatype dtype,
              int root, MPI_Comm comm)
```

In Fortran

```
MPI_BCAST(BUF, COUNT, DTYPE, ROOT, COMM, ERR)
```

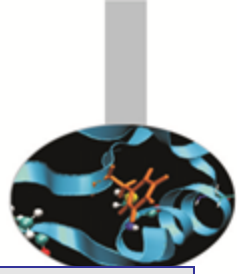
- [IN/OUT] **buf** è l'indirizzo del *send/receive buffer*
- [IN] **count** è di tipo `int` e contiene il numero di elementi del *buffer*
- [IN] **dtype** è di tipo `MPI_Datatype` e descrive il tipo di ogni elemento del *buffer*
- [IN] **root** è di tipo `int` e contiene il *rank* del processo *root* dell'operazione di *broadcast*
- [IN] **comm** è di tipo `MPI_Comm` ed è il comunicatore cui appartengono i processi coinvolti nell'operazione di *broadcast*

SCATTER



- Il processo *root* (p0 nella figura)
  - divide in N parti uguali un insieme di dati contigui in memoria
  - invia una parte ad ogni processo in ordine di *rank*
- Appartiene alla classe *one-to-all*

Binding di MPI_Scatter



In C

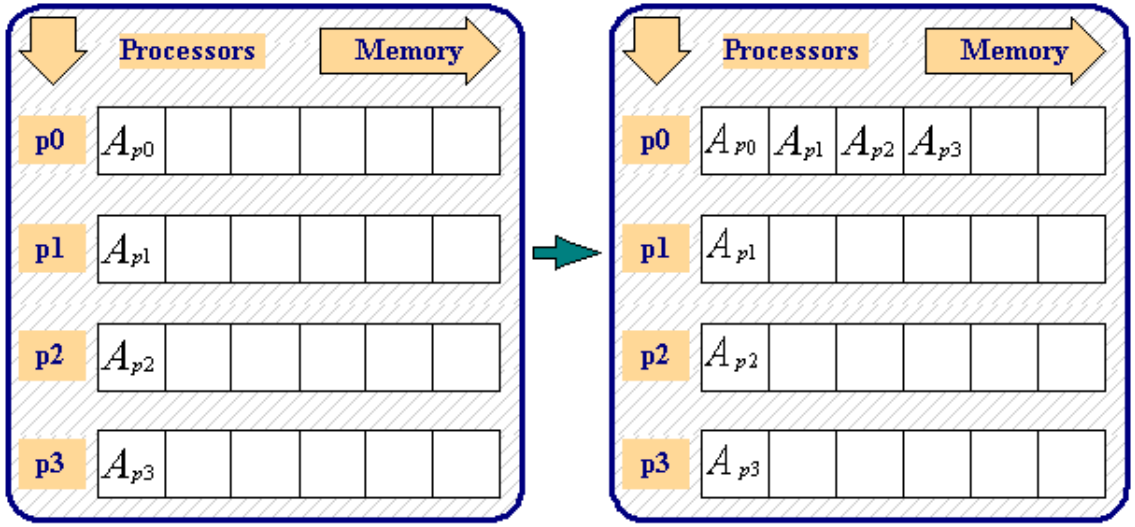
```
int MPI_Scatter(void* sbuf, int scount, MPI_Datatype s_dtype, void*
    rbuf, int rcount, MPI_Datatype r_dtype, int root, MPI_Comm comm)
```

In Fortran

```
MPI_SCATTER(SBUF, SCOUNT, S_DTYPE, RBUF, RCOUNT, R_DTYPE, ROOT,
            COMM, ERR)
```

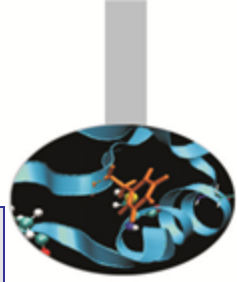
- [IN] **sbuf** è l'indirizzo del send buffer
- [IN] **scount**, di tipo `int`, contiene il numero di elementi spediti ad ogni processo
- [IN] **s_dtype**, di tipo `MPI_Datatype`, descrive il tipo di ogni elemento del *send buffer*
- [OUT] **rbuf** è l'indirizzo del *receive buffer*
- [IN] **rcount**, di tipo `int`, contiene il numero di elementi del *receive buffer*
- [IN] **r_dtype**, di tipo `MPI_Datatype`, descrive il tipo di ogni elemento del *receive buffer*
- [IN] **root**, di tipo `int`, contiene il *rank* del processo *root* della *scatter*
- [IN] **comm**, di tipo `MPI_Comm`, è il comunicatore cui appartengono i processi coinvolti nella *scatter*

GATHER



- Con la funzionalità *GATHER* ogni processo (incluso il *root*) invia il contenuto del proprio *send* buffer al processo *root*
- Il processo *root* riceve i dati e li ordina in funzione del *rank* del processo *sender*
- È l'inverso dell'operazione di scatter
- Appartiene alla classe *all-to-one*

Binding di `MPI_Gather`



In C

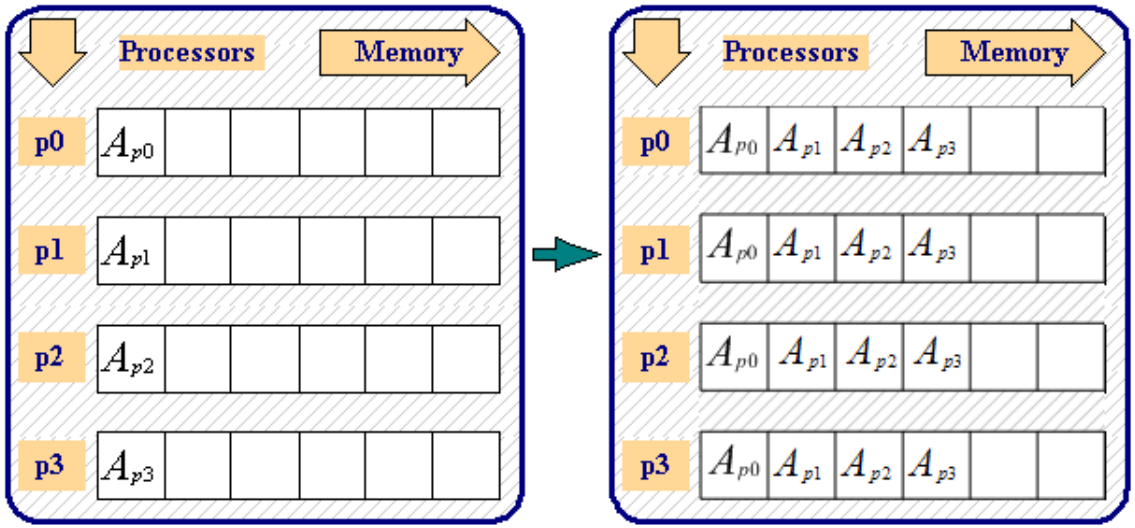
```
int MPI_Gather(void* sbuf, int scount, MPI_Datatype s_dtype,
               void* rbuf, int rcount, MPI_Datatype r_dtype,
               int root, MPI_Comm comm)
```

In Fortran

```
MPI_GATHER(SBUF, SCOUNT, S_DTYPE, RBUF, RCOUNT, R_DTYPE,
           ROOT, COMM, ERR)
```

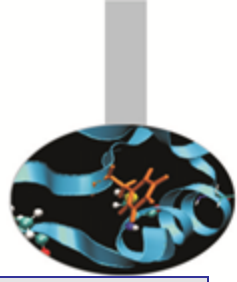
- [IN] **sbuf** è l'indirizzo del *send buffer*
- [IN] **scount** (`int`) contiene il numero di elementi del *send buffer*
- [IN] **s_dtype** (tipo `MPI_Datatype`) descrive il tipo di ogni elemento del *sbuf*
- [OUT] **rbuf** è l'indirizzo del *receive buffer*
- [IN] **rcount** (`int`) contiene il numero di elementi ricevuti da ogni processo
- [IN] **r_dtype** (tipo `MPI_Datatype`) descrive il tipo di ogni elemento del *rbuf*
- [IN] **root** (`int`) contiene il *rank* del processo *root* della *gather*
- [IN] **comm** (tipo `MPI_Comm`) è il comunicatore cui appartengono i processi coinvolti nella *gather*

ALLGATHER



- Di fatto è l'equivalente di un'operazione di *GATHER*, in cui il processo *root* dopo esegue una *BROADCAST*
- È molto più conveniente ed efficiente eseguire una operazione *ALLGATHER* piuttosto che la sequenza *GATHER+BROADCAST*
- Appartiene alla classe *all-to-all*

Binding di `MPI_ALLGather`



In C

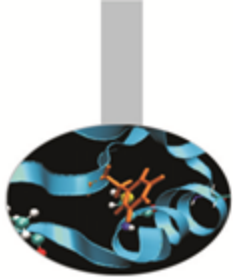
```
int MPI_Allgather(void* sbuf, int scount,MPI_Datatype s_dtype,
        void* rbuf, int rcount, MPI_Datatype r_dtype, MPI_Comm comm)
```

In Fortran

```
MPI_ALLGATHER(SBUF, SCOUNT, S_DTYPE, RBUF, RCOUNT, R_DTYPE, COMM,
              ERR)
```

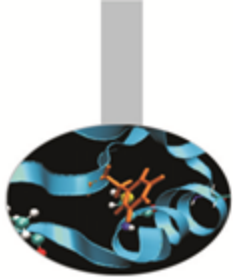
- [IN] **sbuf** è l'indirizzo del *send buffer*
- [IN] **scount** è di tipo `int` e contiene il numero di elementi del *send buffer*
- [IN] **s_dtype** è di tipo `MPI_Datatype` e descrive il tipo di ogni elemento del *send buffer*
- [OUT] **rbuf** è l'indirizzo del *receive buffer*
- [IN] **rcount** è di tipo `int` e contiene il numero di elementi del *receive buffer*
- [IN] **r_dtype** è di tipo `MPI_Datatype` e descrive il tipo di ogni elemento del *receive buffer*
- [IN] **comm** è di tipo `MPI_Comm` ed è il comunicatore cui appartengono i processi coinvolti nella ALL*Gather*

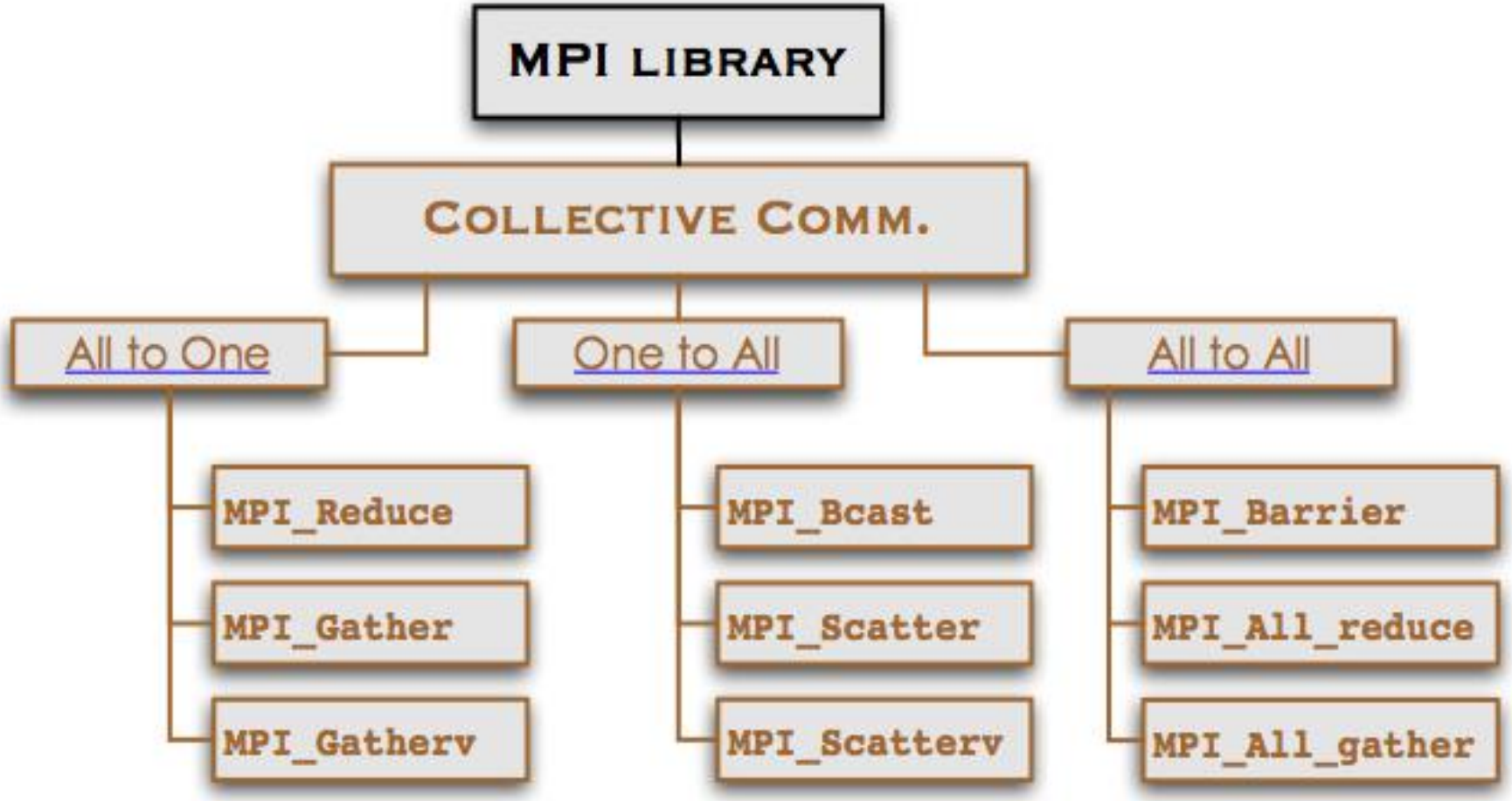
Altre comunicazioni collettive



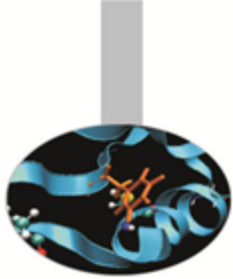
- **`MPI_BARRIER`**: L'esecuzione di ogni processo appartenente allo stesso comunicatore viene messa in pausa fino a quando tutti i processi non sono giunti a questa istruzione
- **`MPI_ALL_REDUCE`**: Il risultato della REDUCE viene comunicato a tutti i processi. È equivalente ad una REDUCE seguita da un BROADCAST
- **`MPI_SCATTERV`** e **`MPI_GATHERV`**: come SCATTER e GATHER, ma consentono di comunicare blocchi di dati di dimensione diversa
- In MPI 3.0 sono state inserite le operazioni **collettive non-blocking** (MPI_Ireduce, MPI_Ibcast, MPI_Iscatter, MPI_Iscatterv …) hanno un parametro aggiuntivo rispetto alle collettive blocking (MPI_request) necessario per il controllo del completamento

Overview: principali comunicazioni collettive





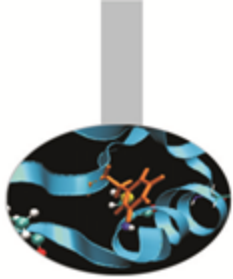
Calcolo parallelo con MPI (3° parte)



Introduzione alle comunicazioni collettive

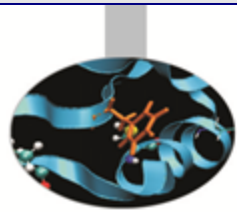
Laboratorio n° 3

Programma della 3° sessione di laboratorio

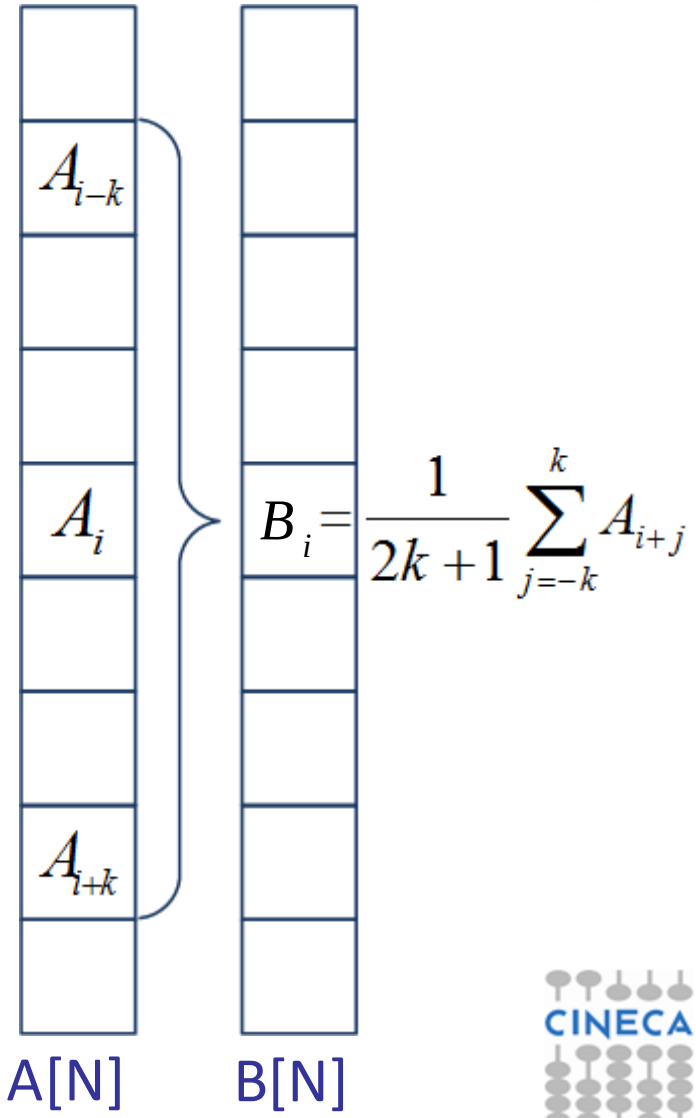


- Uso delle funzioni collettive per implementare pattern di comunicazione standard e sendrecv
  - Array Smoothing (Esercizio 8)
  - Calcolo di π con comunicazioni collettive (Eserc. 9)
  - Prodotto matrice-vettore (Esercizio 10)
  - Prodotto matrice-matrice (Esercizio 11)
- Uso delle funzioni di comunicazione non-blocking
  - Array Smoothing (Esercizio 14)

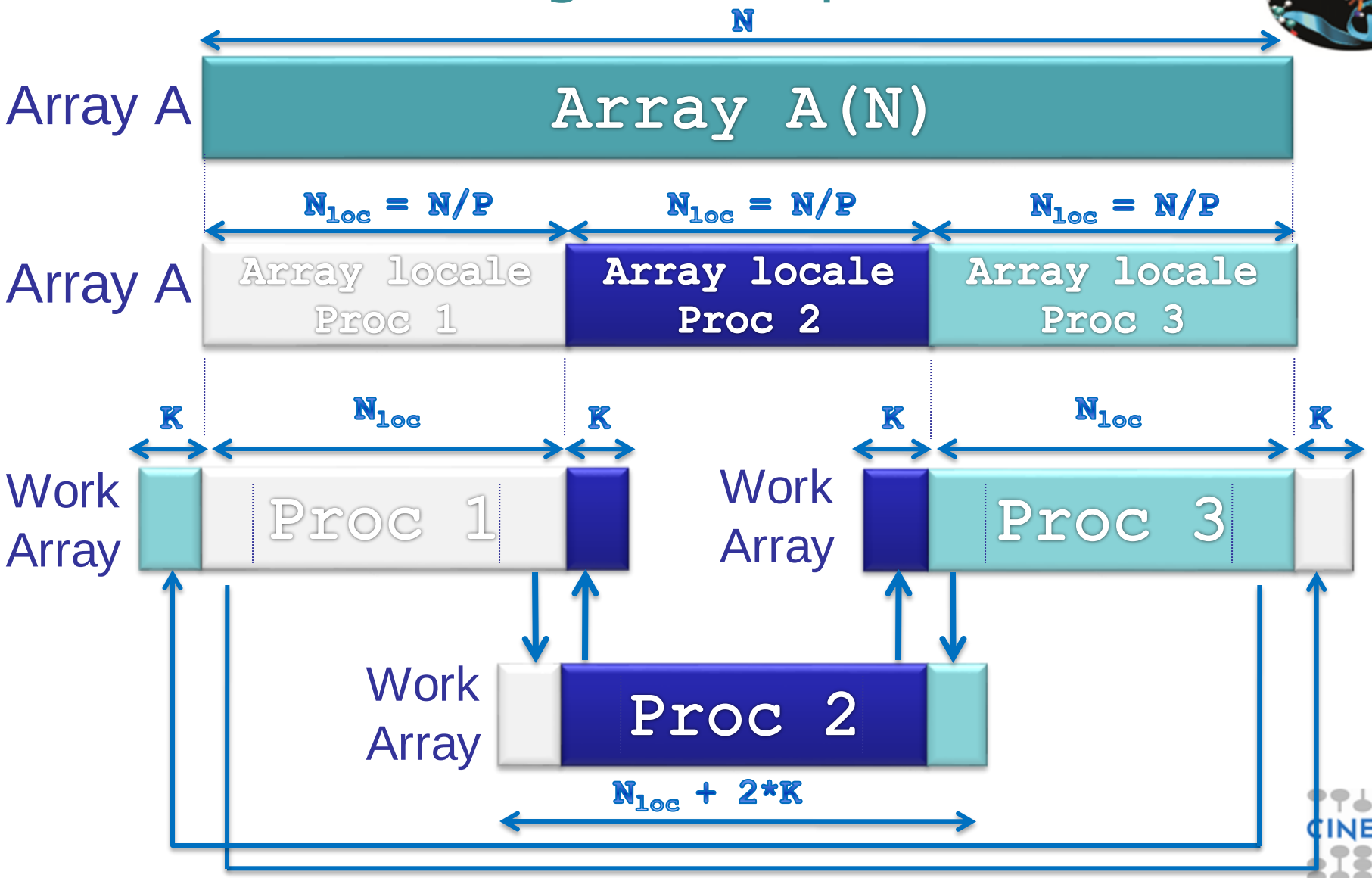
Array smoothing



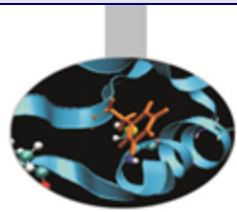
- Dato un *array* A[N]
  - inizializzare e stampare il vettore A
- per iter volte:
  - calcolare un nuovo *array* B in cui ogni elemento sia uguale alla media aritmetica del suo valore e dei suoi K primi vicini al passo precedente
    - nota: l'array è periodico, quindi il primo e l'ultimo elemento di A sono considerati primi vicini
  - stampare il vettore B
  - copiare B in A e continuare l'iterazione



Array smoothing: algoritmo parallelo

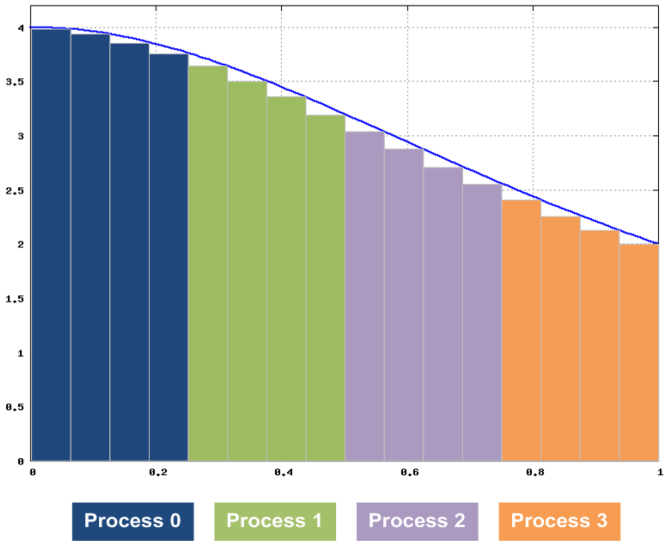


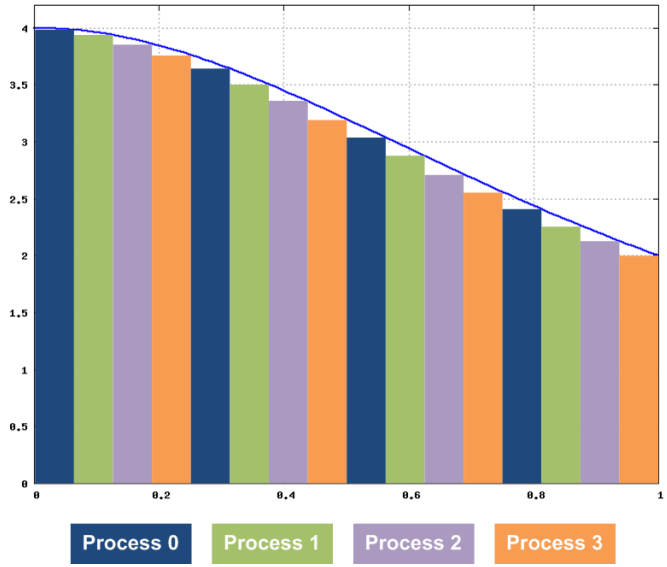
Array smoothing: algoritmo parallelo



- Il processo di *rank* 0
  - genera l'*array* globale di dimensione N , multiplo del numero P di processi
  - inizializza il vettore A con A[i] =i
  - distribuisce il vettore A ai P processi i quali riceveranno N_{loc} elementi nell'array locale (MPI_Scatter)
- Ciascun processo ad ogni passo di *smoothing*:
  - costruisce l'array di lavoro:
    - I primi K elementi dovranno ospitare la copia degli ultimi K elementi *dell'array* locale in carico al processo precedente (MPI_Sendrecv)
    - I successivi N_{loc} elementi dovranno ospitare la copia degli N_{loc} elementi dell'*array* locale in carico al processo stesso
    - Gli ultimi K elementi dovranno ospitare la copia dei primi K elementi del l'*array* locale in carico al processo di *rank* immediatamente superiore (MPI_Sendrecv)
  - Effettua lo *smoothing* degli N_{loc} elementi interni e scrive i nuovi elementi sull'array A
- Il processo di *rank* 0 ad ogni passo raccoglie (MPI_Gather) e stampa i risultati parziali

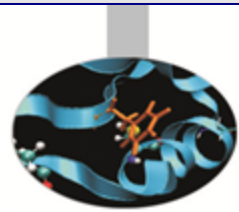
Calcolo di π con *reduction*: algoritmo parallelo

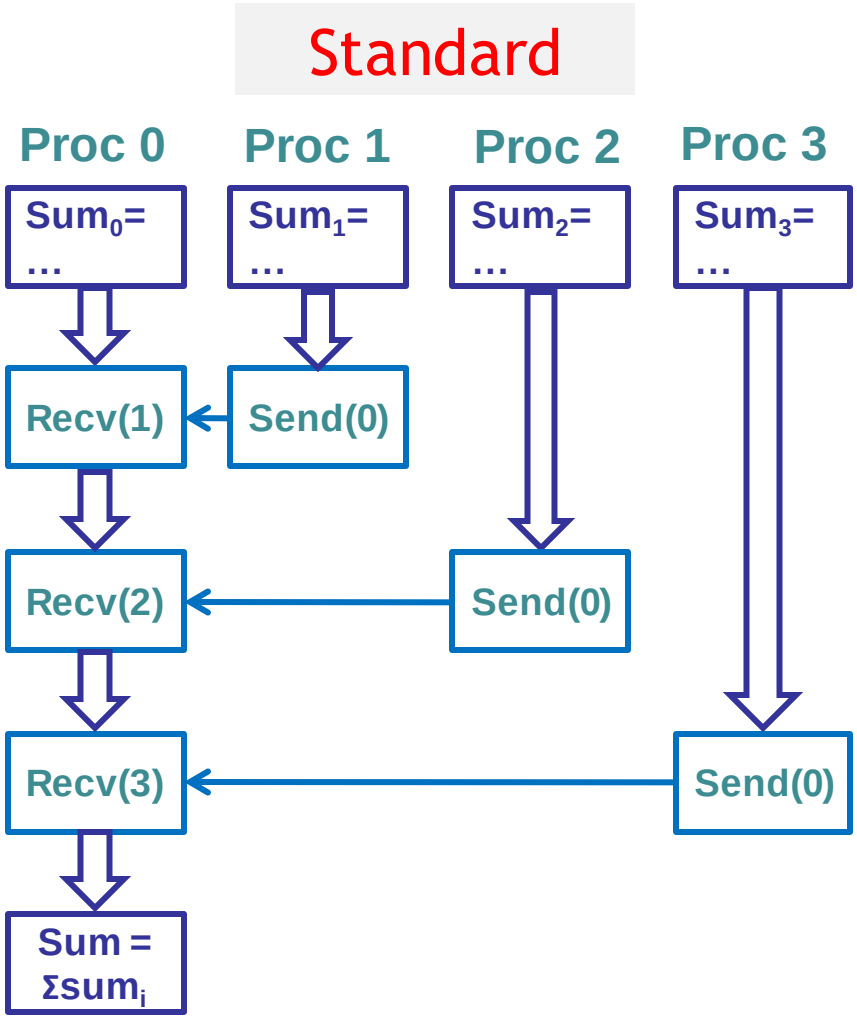


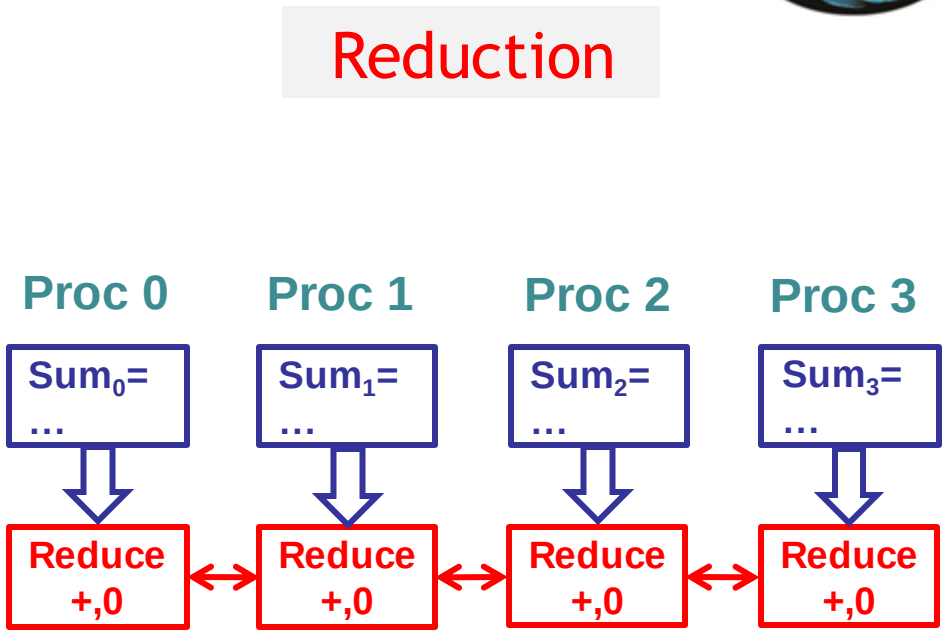


- Ogni processo calcola la somma parziale di propria competenza rispetto alla decomposizione scelta, come nel caso dell'esercizio 3
- Tutti i processi contribuiscono all'operazione di somma globale utilizzando la funzione di comunicazione collettiva `MPI_Reduce`

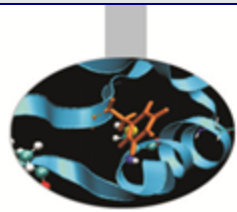
Calcolo di π in parallelo con *reduction*: *flowchart*







Prodotto Matrice-Vettore

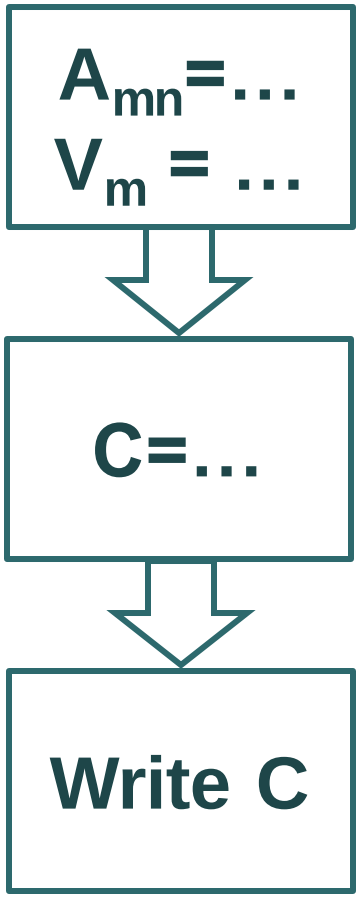


- Data una matrice A, di dimensione `size*size`, ed un vettore V di dimensione `size`, calcolare il prodotto C=A*V
- Ricordando che:

$$C_m = \sum_{n=1}^{size} A_{mn} V_n$$

- Nella versione parallela, per semplicità, assumiamo che `size` sia multiplo del numero di processi

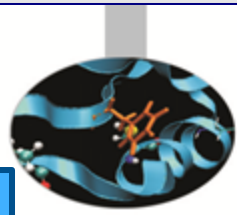
Prodotto matrice-vettore: algoritmo seriale



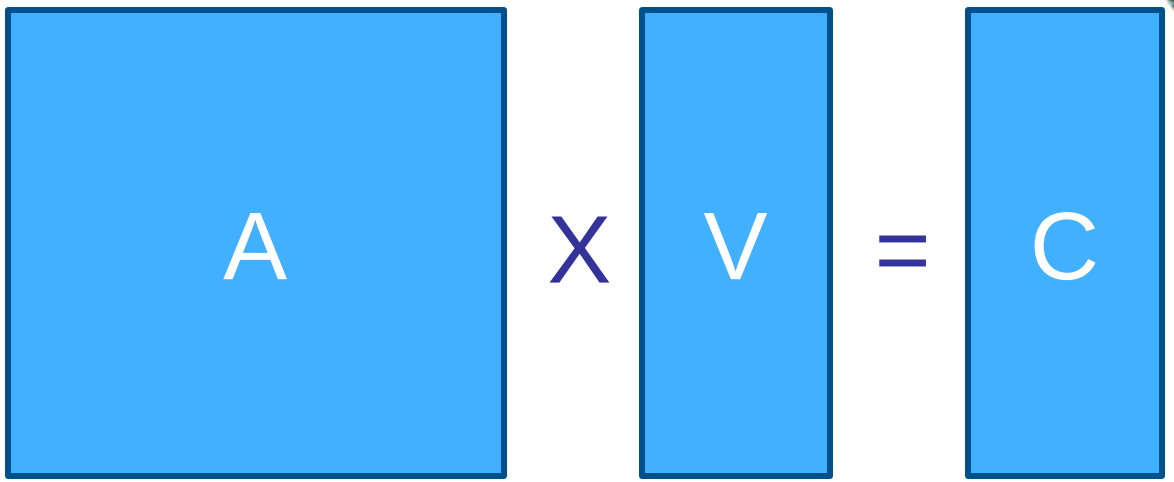
- Inizializzare gli array A e V
  - $A_{mn} = m+n$
  - $V_m = m$
- Core del calcolo
  - Loop esterno sull'indice m=1,size di riga della matrice A (e del vettore C)
  - Loop interno sull'indice n=1,size del vettore V

    $C_m = \Sigma_n (A_{mn} * V_n)$
- Scrittura del vettore C

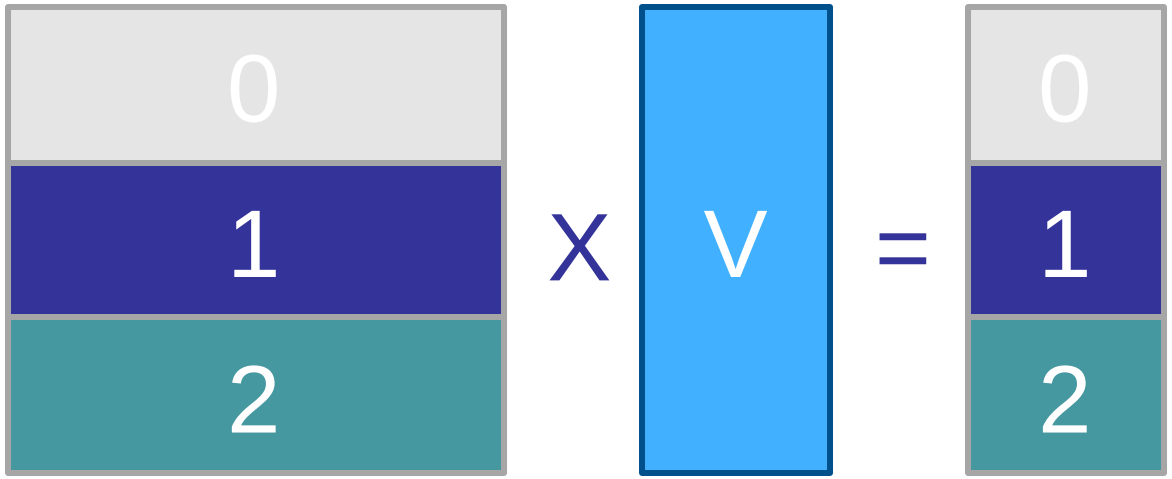
Prodotto matrice-vettore



Versione seriale

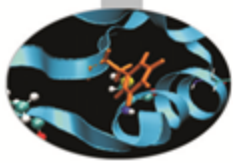


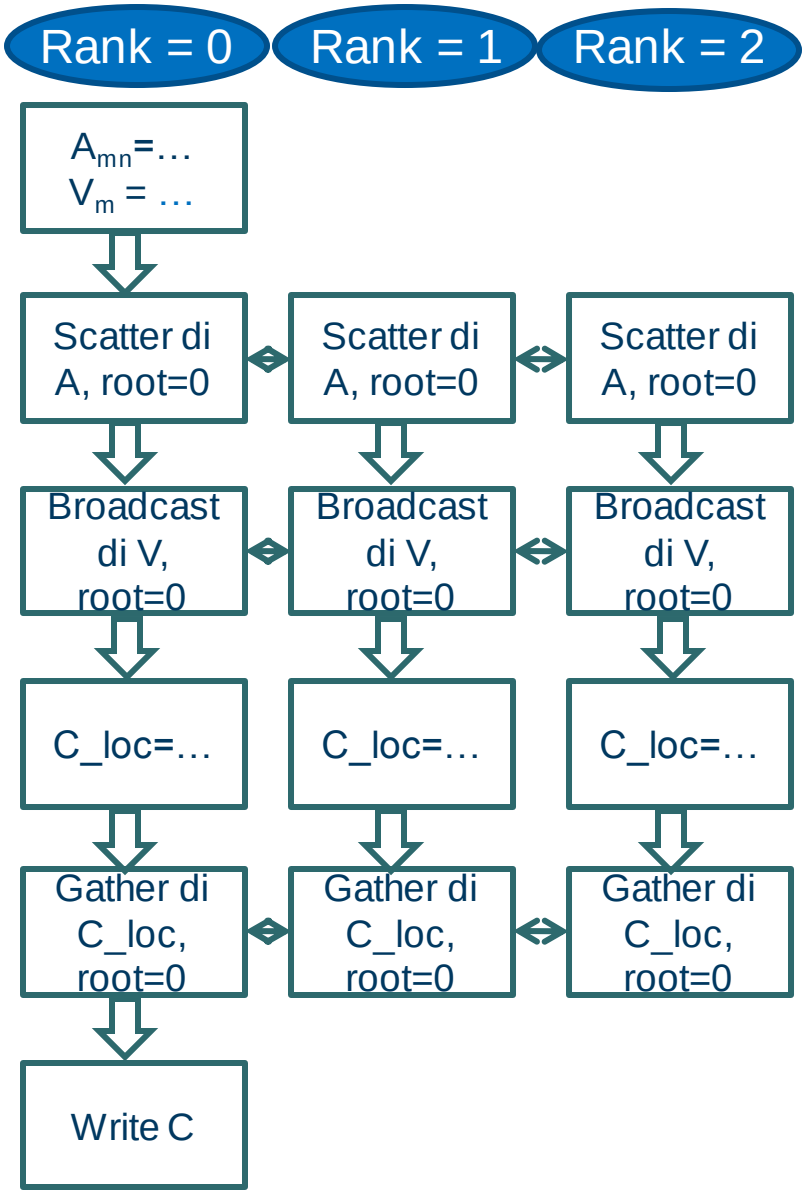
Versione parallela



N.B. Poiché in Fotran le matrici sono allocate per colonne, è necessario effettuare la trasposizione della matrice A

Prodotto matrice-vettore in parallelo



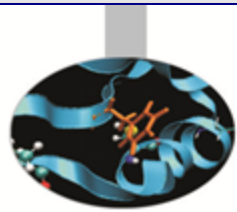


- Inizializzare gli array `A` e `V` sul solo processo master (rank = 0)
- Scatter della matrice `A`
  - Il processo master distribuisce a tutti i processi, se stesso incluso, un sotto-array (un set di righe contigue) di `A`
  - I vari processi raccolgono i sotto-array di `A` in array locali al processo (es. `A_loc`)
- Broadcast del vettore `V`
  - Il processo master distribuibuisce a tutti i processi, se stesso incluso, l'intero vettore `V`
- Core del calcolo sui soli elementi di matrice locali ad ogni processo (es. `A_loc` e `C_loc`)
  - Loop esterno sull'indice `m=1,size/nprocs`
  - Loop interno sull'indice `n=1,size`

  $C\_loc_m = \Sigma_n (A\_loc_{mn} * V_n)$
- Gather del vettore `C`
  - Il processo master (rank = 0) raccoglie gli elementi di matrice del vettore risultato `C` calcolate da ogni processo (`C_loc`)
- Scrittura del vettore `C` da parte del solo processo master

Prodotto Matrice-Matrice

esercizio facoltativo

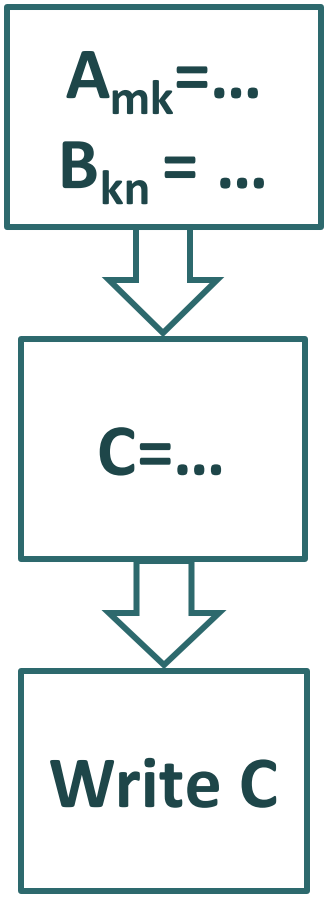


- Date due matrici A e B di dimensione size*size, calcolare il prodotto C=A*B
- Ricordando che:

$$C_{mn} = \sum_{k=1}^{size} A_{mk} B_{kn}$$

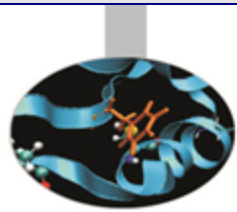
- La versione parallela andrà implementata assumendo che size sia multiplo del numero di processi

Prodotto matrice-matrice: algoritmo seriale

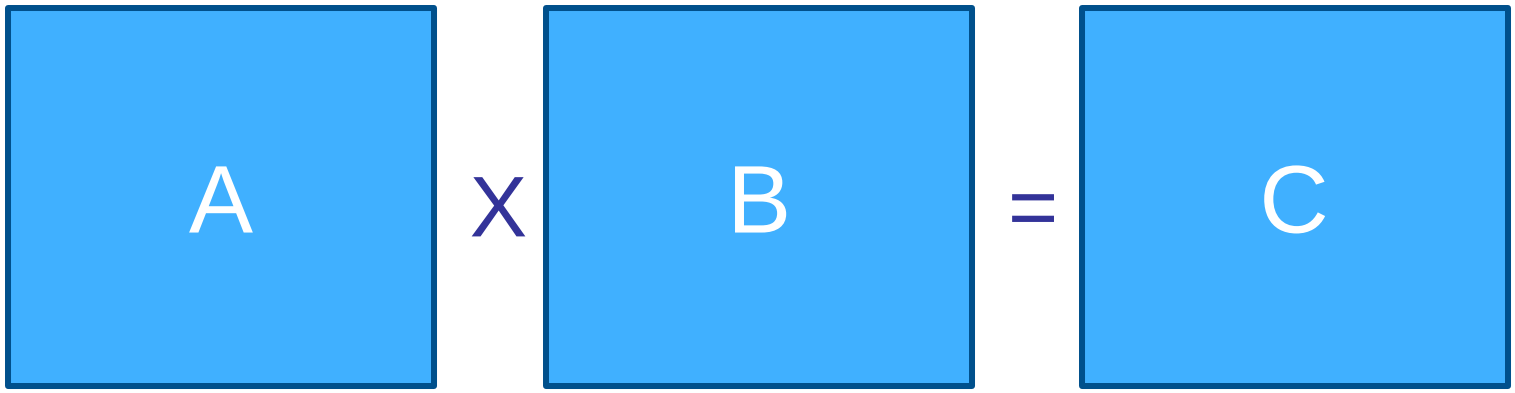


- Inizializzazione degli array A e B
  - A_{mk} = m+k
  - B_{kn} = n+k
- Core del calcolo
  - Loop esterno sull'indice m=1,size di riga della matrice A (e della matrice C)
  - Loop intermedio sull'indice n=1,size di colonna della matrice B (e della matrice C)
  - Loop interno sull'indice k=1,size di colonna della matrice A e di riga della matrice B
  - Calcolo del prodotto A_{mk}*B_{kn} ed accumulo su C_{mn}
- Scrittura della matrice C

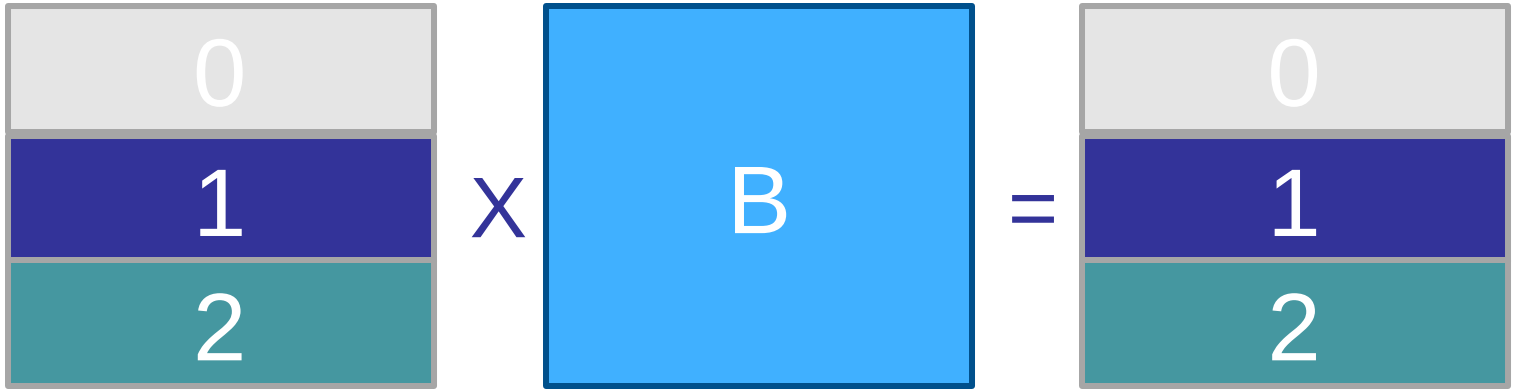
Prodotto matrice-matrice in C



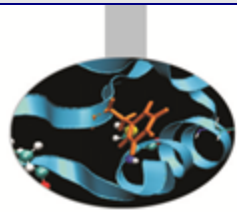
Versione seriale



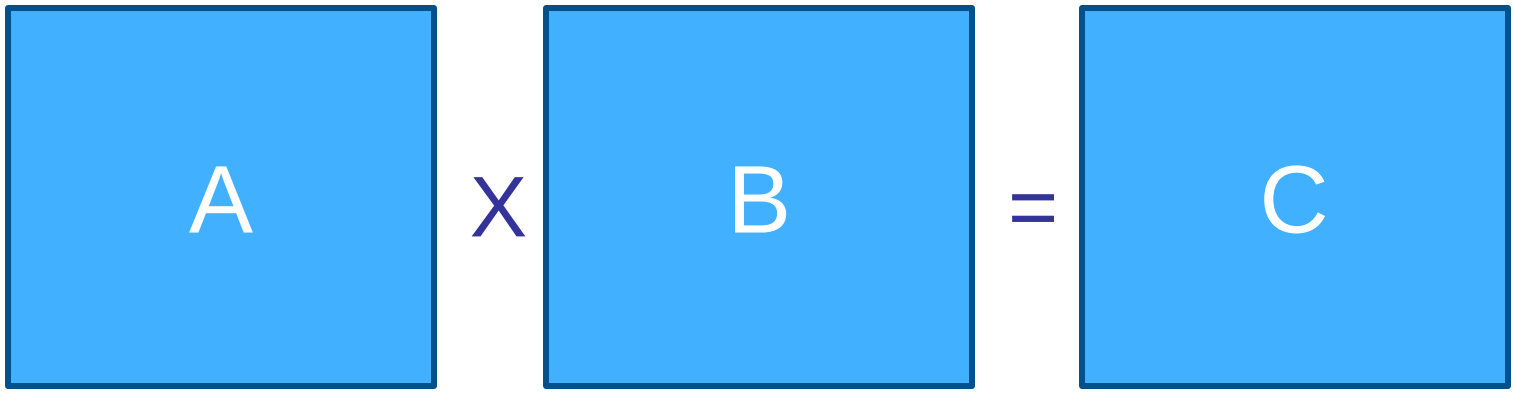
Versione parallela



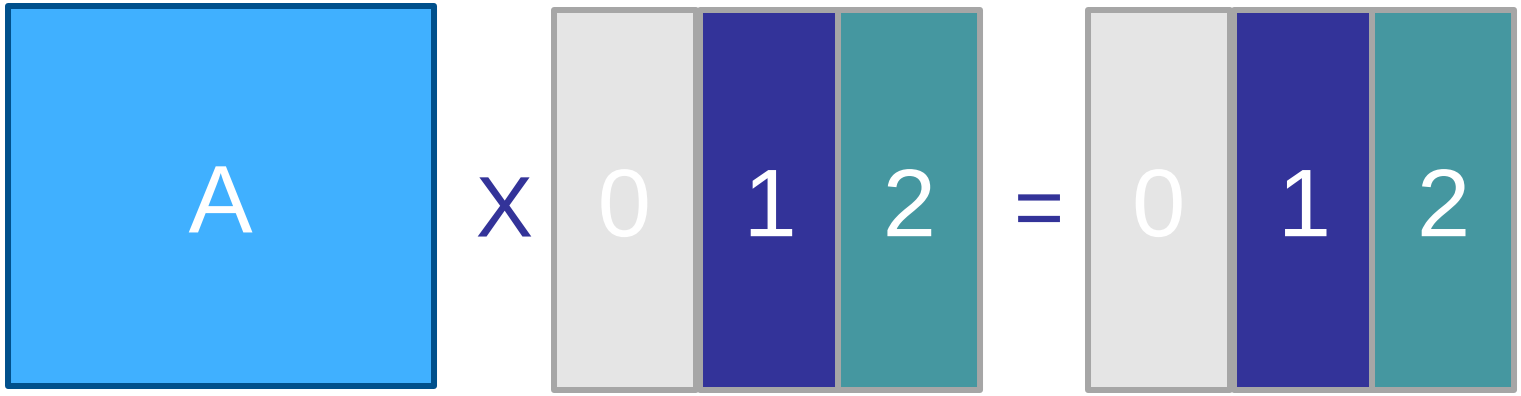
Prodotto matrice-matrice in Fortran



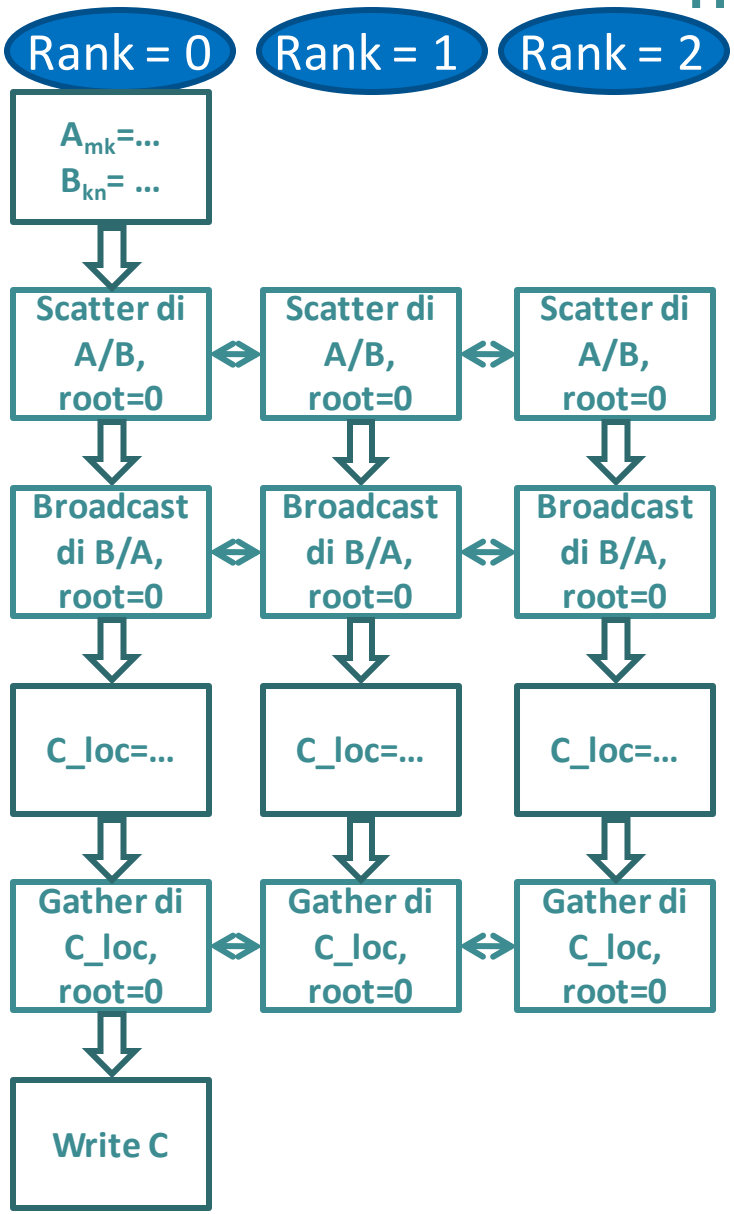
Versione seriale



Versione parallela

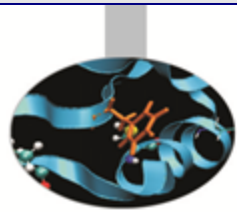


Prodotto matrice-matrice in parallelo

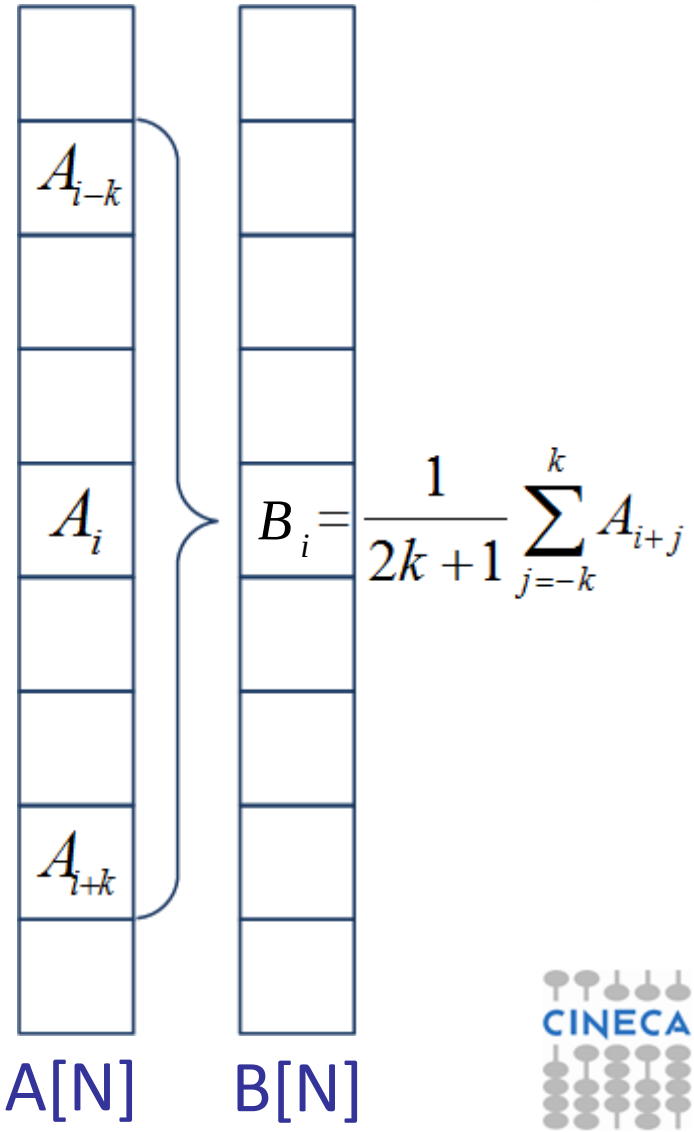


- Inizializzare le matrici `A` e `B` sul solo processo master (rank = 0)
- In C: Scatter della matrice `A` e Broadcast di `B` (viceversa in Fortran)
  - Il processo master distribuisce a tutti i processi, se stesso incluso, un sotto-array (un set di righe contigue) di `A` (B in Fortran)
  - Il processo root distribuisce a tutti i processi, se stesso incluso, l'intera matrice `B (A)`
  - I vari processi raccolgono i sotto-array di `A (B)` in un array locale al processo (es `A_loc`)
- Core del calcolo sui soli elementi di matrice locali ad ogni processo
  - Loop esterno sull'indice `m=1,size/nprocs`
  - Accumulo su C_loc_m
- Gather della matrice `C`
  - Il processo master (rank = 0) raccoglie gli elementi della matrice risultato C calcolate da ogni processo (C_local)
- Scrittura della matrice `C` da parte del solo processo master

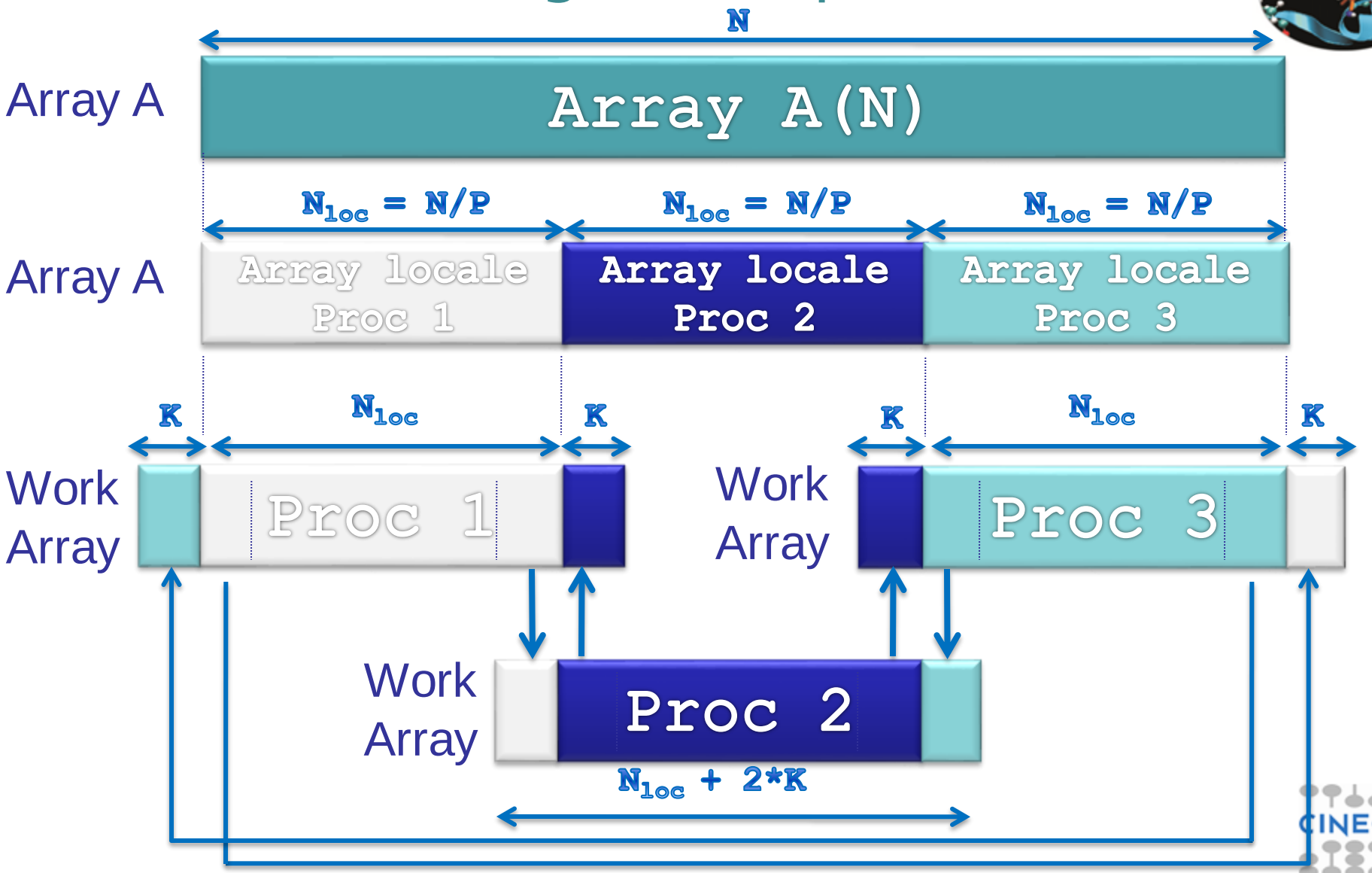
Array smoothing



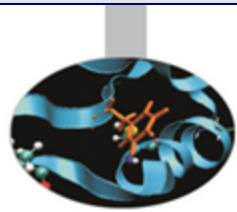
- dato un *array* A[N]
- stampare il vettore A
- per ITER volte:
  - calcolare un nuovo *array* B in cui ogni elemento sia uguale alla media aritmetica del suo valore e dei suoi K primi vicini al passo precedente
    - nota: l'array è periodico, quindi il primo e l'ultimo elemento di A sono considerati primi vicini
  - Stampare il vettore B
  - Copiare B in A e continuare l'iterazione



Array smoothing: algoritmo parallelo



Array smoothing: algoritmo parallelo



- Il processo di *rank* 0
  - genera l'*array* globale di dimensione N , divisibile per il numero P di processi
  - inizializza il vettore A con A[i] =i
  - distribuisce il vettore A ai P processi i quali riceveranno N_{loc} elementi nell'array locale
- Ciascun processo ad ogni passo di *smoothing*:
  - **Avvia** le opportune ***send non-blocking*** verso i propri processi primi vicini per spedire i suoi elementi
  - **Avvia** le opportune ***receive non-blocking*** dai propri processi primi vicini per ricevere gli elementi dei vicini
  - **Effettua lo *smoothing* dei soli elementi del vettore che non implicano la conoscenza di elementi di A in carico ad altri processi**
  - **Dopo l'avvenuta ricezione** degli elementi in carico ai processi vicini, **effettua lo *smoothing* dei rimanenti elementi del vettore**
- Il processo di *rank* 0 ad ogni passo raccoglie i risultati parziali e li stampa