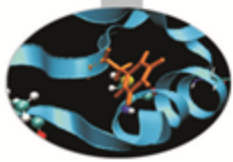
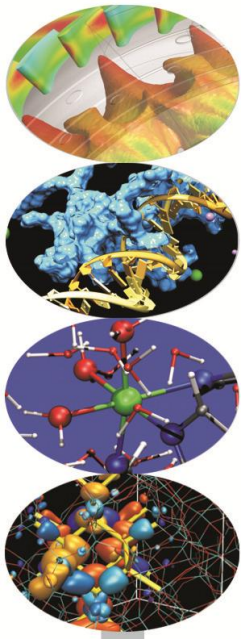
CINECA SCAI
SuperComputing Applications and Innovation



MPI
Laboratorio 3



Isabella Baccarelli
i.baccarelli@cineca.it

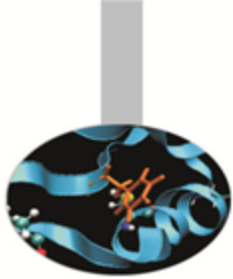
Cristiano Padrin
c.padrin@cineca.it

Mariella Ippolito
m.ippolito@cineca.it

Vittorio Ruggiero
v.ruggiero@cineca.it

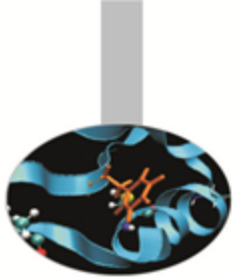
CINECA

Programma della 3° sessione di laboratorio



- Modalità di completamento della comunicazione
  - Uso della syncronous send, della *buffered send* e delle funzioni di gestione dei buffer (Esercizio 12)
- Uso delle funzioni di comunicazione non-blocking
  - *Non blocking circular shift* (Esercizio 13)
  - *Array Smoothing* (Esercizio 14)

Funzioni per il timing: MPI_Wtime e MPI_Wtick

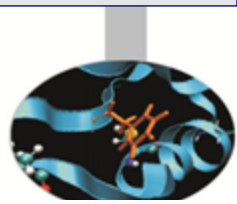


- È importante conoscere il tempo impiegato dal codice nelle singole parti per poter analizzare le performance
- `MPI_Wtime`: fornisce il numero floating-point di secondi intercorso tra due sue chiamate successive

```
double starttime, endtime;
starttime = MPI_Wtime();
.... stuff to be timed ...
endtime = MPI_Wtime();
printf("That took %f seconds\n",endtime-starttime);
```

- `MPI_Wtick`: ritorna la precisione di MPI_Wtime, cioè ritorna 10^{-3} se il contatore è incrementato ogni millesimo di secondo.
- NOTA: anche in FORTRAN sono funzioni

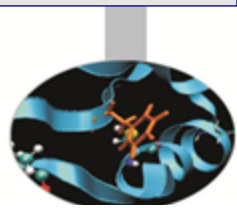
Syncronous Send blocking



- Modificare il codice dell'Esercizio 2, utilizzando la funzione `MPI_Ssend` per la spedizione di un *array* di *float*
- Misurare il tempo impiegato nella `MPI_Ssend` usando la funzione `MPI_Wtime`

```c
#include <stdio.h>
#include <mpi.h>
#define MSIZE 10

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

    MPI_Status status;
    int rank, size;
    int i, j;

    /* data to communicate */
    float  matrix[MSIZE];

    /* Start up MPI */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        for (i = 0; i < MSIZE; i++)
           matrix[i] = (float)i;
        MPI_Send(matrix, MSIZE, MPI_FLOAT, 1, 666, MPI_COMM_WORLD);
    } else if (rank ==1) {
        MPI_Recv(matrix, MSIZE, MPI_FLOAT, 0, 666, MPI_COMM_WORLD, &status);
        printf("\nProcess 1 receives the following array from process 0.\n");
        for (i = 0; i < MSIZE; i++)
           printf("%6.2f\n", matrix[i]);
    }

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

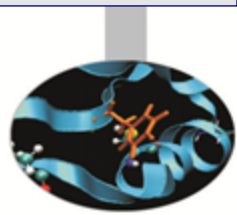
Send buffered blocking



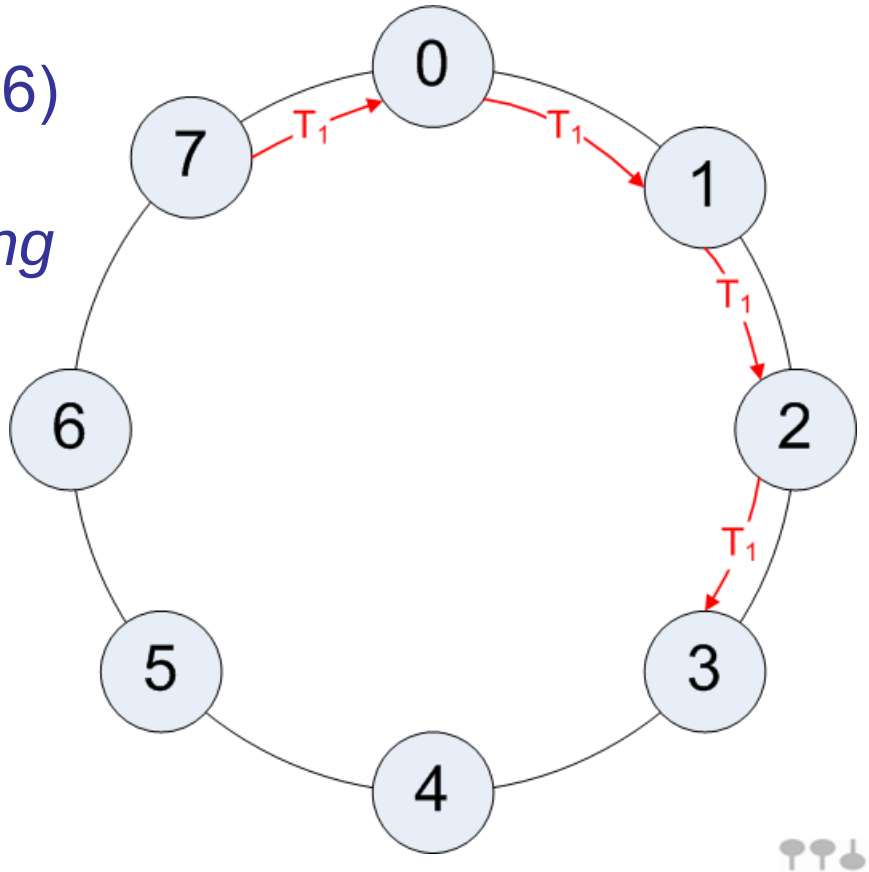
- Modificare l'Esercizio 2, utilizzando la funzione **`MPI_Bsend`** per spedire un *array* di *float*
- N.B.: la **`MPI_Bsend`** prevede la gestione diretta del *buffer* di comunicazione da parte del programmatore
- Misurare il tempo impiegato nella **`MPI_Bsend`** usando la funzione **`MPI_Wtime`**

```
#include <stdio.h>
#include <mpi.h>
#define MSIZE 10

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

    MPI_Status status;
    int rank, size;
    int i, j;

    /* data to communicate */
    float  matrix[MSIZE];

    /* Start up MPI */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        for (i = 0; i < MSIZE; i++)
           matrix[i] = (float)i;
        MPI_Send(matrix, MSIZE, MPI_FLOAT, 1, 666, MPI_COMM_WORLD);
    } else if (rank ==1) {
        MPI_Recv(matrix, MSIZE, MPI_FLOAT, 0, 666, MPI_COMM_WORLD, &status);
        printf("\nProcess 1 receives the following array from process 0.\n");
        for (i = 0; i < MSIZE; i++)
           printf("%6.2f\n", matrix[i]);
    }

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

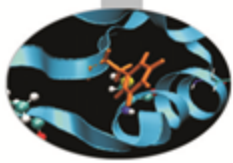
Circular Shift non-blocking



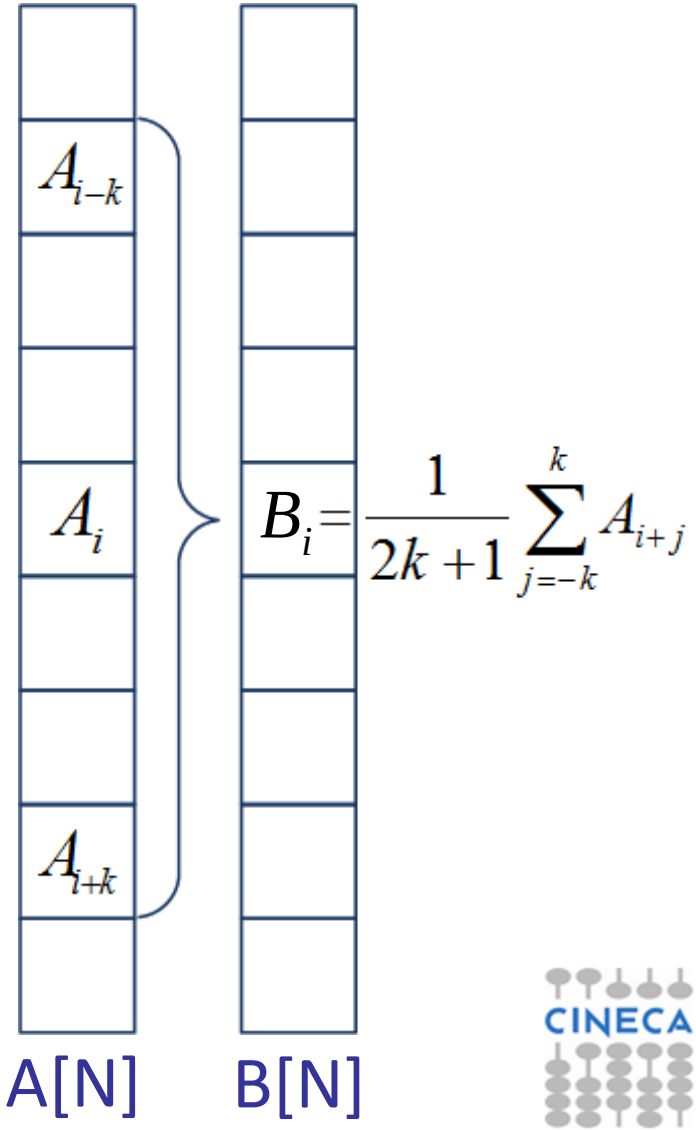
- Modificare il codice dello *shift* circolare periodico in "versione *naive*" (Esercizio 6) utilizzando le funzioni di comunicazione *non-blocking* per evitare la condizione *deadlock* per N=2000



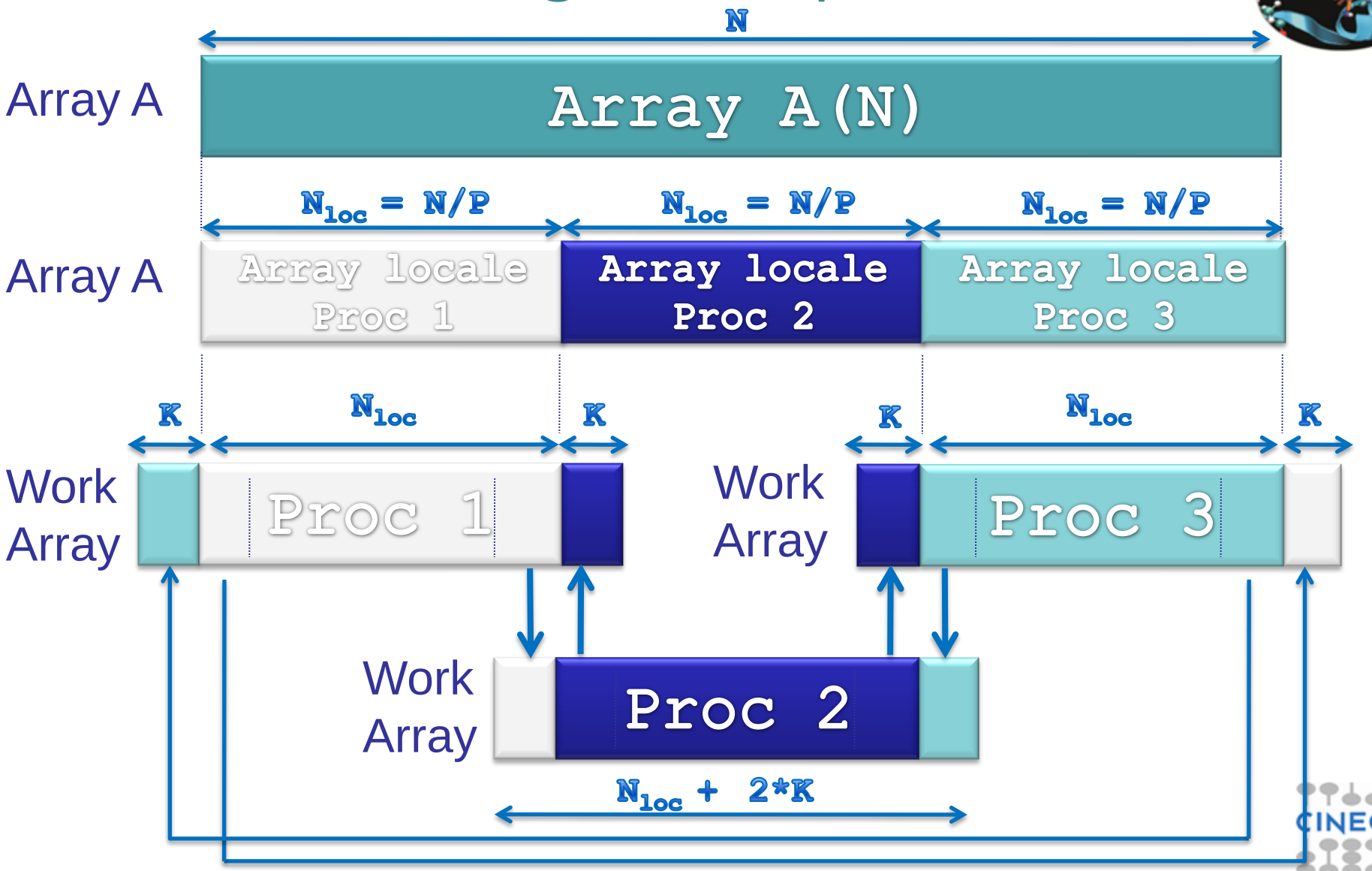
Array smoothing



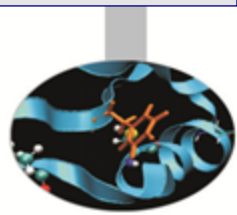
- dato un *array* A[N]
- stampare il vettore A
- per ITER volte:
  - calcolare un nuovo *array* B in cui ogni elemento sia uguale alla media aritmetica del suo valore e dei suoi K primi vicini al passo precedente
    - nota: l'array è periodico, quindi il primo e l'ultimo elemento di A sono considerati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