

# Laboratorio di Calcolo Parallelo

## Lezione 2: Aspetti avanzati ed esempi in MPI

Francesco Versaci & Alberto Bertoldo

Università di Padova

6 maggio 2009



- 1 Aspetti avanzati di MPI
  - Deadlock
  - Comunicazioni punto-punto non bloccanti
  - Comunicazioni collettive
  - Misurazione delle prestazioni
  
- 2 Esempio: Moltiplicazione di matrici

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
    int me, np, q, sendto;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if (np%2==1) return 0;
    if (me%2==1) {sendto = me-1;}
    else {sendto = me+1;}
    MPI_Recv(&q, 1, MPI_INT, sendto, sendto, \
        MPI_COMM_WORLD, &status);
    MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
    printf("Sent_%d_to_proc_%d,_received_%d_from_proc_%d\n", \
        me, sendto, q, sendto);
    MPI_Finalize();
    return 0;
}
```



```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
    int me, np, q, sendto;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if (np%2==1) return 0;
    if (me%2==1) {sendto = me-1;}
    else {sendto = me+1;}
    MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
    MPI_Recv(&q, 1, MPI_INT, sendto, sendto, \
        MPI_COMM_WORLD, &status);
    printf("Sent_%d_to_proc_%d,_received_%d_from_proc_%d\n", \
        me, sendto, q, sendto);
    MPI_Finalize();
    return 0;
}
```



- Cambiare l'**ordine** delle chiamate
  - Pericoloso se le dipendenze coinvolgono più di 2 processi
  - Occorre avere ben chiaro il pattern di comunicazione
  - Non aumenta la complessità del programma
- Usare le chiamate **non bloccanti**
  - Aumenta la complessità del programma
  - Può anche aumentare l'efficienza
- Usare la modalità **bidirezionale**
  - Non è sempre applicabile
  - Rende più leggibile e semplice il codice
- Usare la modalità **buffered**
  - Occorre gestire i buffer
  - Comportamento prevedibile
  - Non aumenta di molto la complessità

# Deadlock

Evitarlo cambiando l'ordine delle chiamate

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
    int me, np, q, sendto;
    MPI_Status st;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if (np%2==1) return 0;
    if (me%2==1) {sendto = me-1;}
    else {sendto = me+1;}
    if (me%2 == 0) {
        MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
        MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD, &st);
    } else {
        MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD, &st);
        MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
    }
    printf("Sent %d to proc %d, received %d from proc %d\n", \
        me, sendto, q, sendto);
    MPI_Finalize();
    return 0;
}
```



## Spedire un messaggio

```
int MPI_Isend(void* buf, int count, \
             MPI_Datatype datatype, int dest, int tag, \
             MPI_Comm comm, MPI_Request *request)
```

- Inizializza il processo di spedizione
- Ritorna quando la richiesta di spedizione è stata registrata
- `buf` non può essere sovrascritto finché la richiesta è **pendente**
- Occorre controllare lo stato della richiesta di spedizione
- Può corrispondere a una ricezione bloccante

## Note sull'efficienza

- Aumenta la **sovrapposizione** tra computazione e comunicazione
- Il grado di sovrapposizione dipende dall'implementazione e dallo hardware



## Ricevere un messaggio

```
int MPI_Irecv(void* buf, int count, \  
MPI_Datatype datatype, int source, \  
int tag, MPI_Comm comm, MPI_Request *request)
```

- Inizializza il processo di ricezione (recv posting)
- Ritorna quando la richiesta di ricezione è stata registrata
- `buf` non può essere usato finché la richiesta è pendente
- Occorre controllare lo stato della richiesta di ricezione
- Può corrispondere a una spedizione bloccante

## Note sull'efficienza

È consigliabile effettuare il posting **ASAP**:

- Può sbloccare un mittente

## Controllare lo stato

```
int MPI_Test(MPI_Request *request, \
             int *flag, MPI_Status *status)
```

- Ritorna subito dopo aver controllato lo stato
- `flag = true` se l'operazione è stata completata e:
  - `send`: `buf` può essere aggiornato
  - `recv`: `buf` contiene i dati ricevuti

## Aspettare la conclusione

```
int MPI_Wait(MPI_Request *request, MPI_Status *st)
```

- Ritorna quando l'operazione è conclusa

## Note sull'efficienza

È consigliabile effettuare la wait **ALAP**

## Stato di richieste multiple

```
int MPI_Waitany(int count, MPI_Request *array_req, \
               int *index, MPI_Status *st)
```

```
int MPI_Waitall(int count, MPI_Request *array_req, \
               MPI_Status *array_st)
```

```
int MPI_Waitsome(int incount, MPI_Request *array_req, \
                 int *outcount, int *array_ind, MPI_Status *array_st)
```

N.B.: Esistono primitive analoghe per il **test**

**ANY**: una **sol**a operazione tra quelle in sospeso

**ALL**: **tutte** le operazioni in sospeso

**SOME**: **almeno** una operazione tra quelle in sospeso



# Comunicazioni punto-punto

Esempio 3: Nearest neighbor exchange in ring topology

```
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[4];

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

prev = (rank-1)%numtasks;
next = (rank+1)%numtasks;

MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);
MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

    { do some work }

MPI_Waitall(4, reqs, stats);

MPI_Finalize();
```



## Comunicazione tra **più** processi MPI

### Tutti o nessuno

- **Tutti** i processi del comunicatore sono coinvolti
- Le primitive sono tutte **bloccanti** (nel senso che abbiamo visto)
- Se un processo non partecipa?

Tipi di comunicazioni collettive:

**Sincronizzazione:** barrier

**Trasferimento dati:** broadcast, scatter, gather, all-to-all

**Operazioni di riduzione:** sum, max, min, ...

### Sincronizzare i processi

```
int MPI_Barrier(MPI_Comm comm)
```

- Blocca il chiamante finché **tutti** i processi effettuano la chiamata

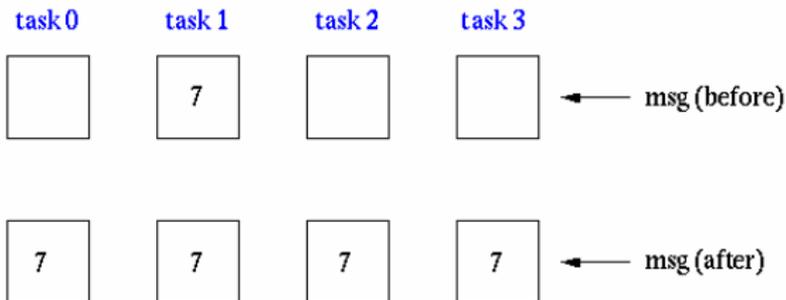
### Replicare i dati

```
int MPI_Bcast(void* buffer, int count, \
             MPI_Datatype datatype, int root, MPI_Comm comm)
```

- Replica il contenuto di `buffer` da `root` a tutti gli altri processi
- Non implica necessariamente la sincronizzazione

```
count = 1;
source = 1;
MPI_Bcast(&msg, count, MPI_INT, source, MPI_COMM_WORLD);
```

broadcast originates in task 1



### Distribuire i dati

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

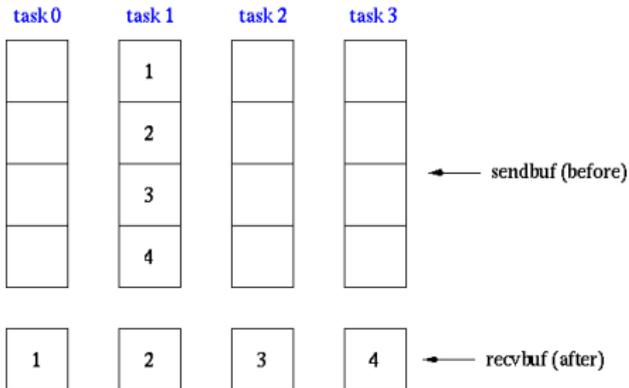
```
sendcnt = 1;
```

```
recvcnt = 1;
```

```
src = 1;
```

```
MPI_Scatter(sendbuf, sendcnt, MPI_INT, \
recvbuf, recvcnt, MPI_INT, \
src, MPI_COMM_WORLD);
```

task 1 contains the message to be scattered



### Raccogliere i dati

```
int MPI_Gather(void* sendbuf, int sendcount, \
MPI_Datatype sendtype, void* recvbuf, int recvcount, \
MPI_Datatype recvttype, int root, MPI_Comm comm)
```

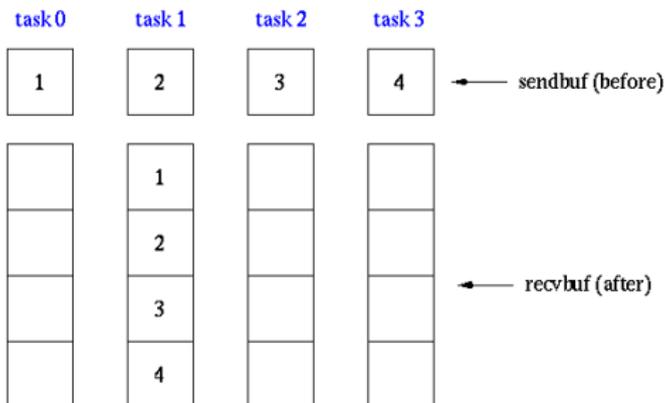
sendcnt = 1;

recvcnt = 1;

src = 1;

messages will be gathered in task 1

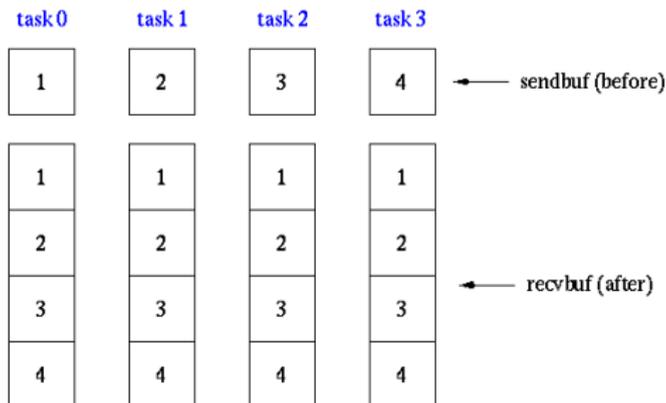
```
MPI_Gather(sendbuf, sendcnt, MPI_INT,
recvbuf, recvcnt, MPI_INT,
src, MPI_COMM_WORLD);
```



### Raccogliere i dati + replicazione

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

```
sendcnt = 1;  
recvcnt = 1;  
MPI_Allgather(sendbuf, sendcnt, MPI_INT,  
recvbuf, recvcnt, MPI_INT,  
MPI_COMM_WORLD);
```

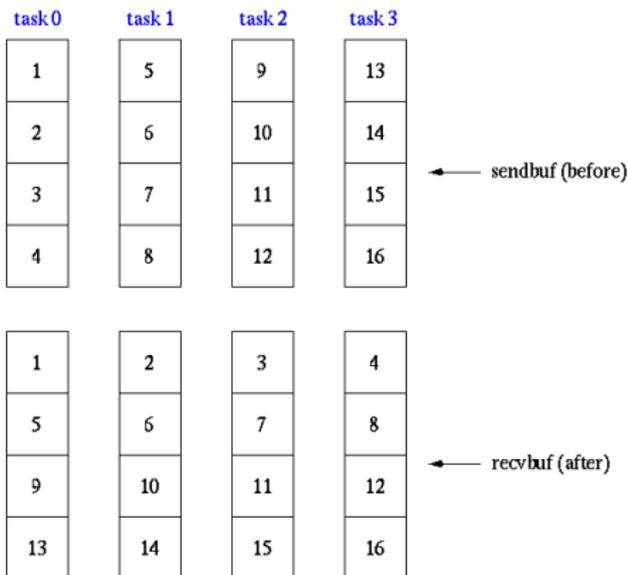


### Scambio da tutti a tutti

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

```
sendcnt = 1;  
recvcnt = 1;
```

```
MPI_Alltoall(sendbuf, sendcnt, MPI_INT,  
             recvbuf, recvcnt, MPI_INT,  
             MPI_COMM_WORLD);
```



### Operazioni di riduzione

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

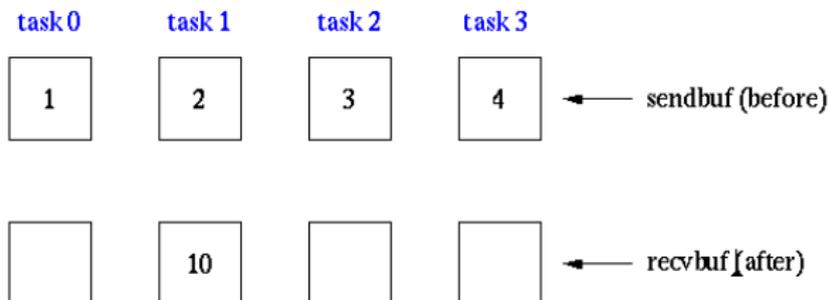
- Operazioni predefinite: `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, `MPI_PROD`, ...
- Operazioni definite dall'utente

`count = 1;`

`dest = 1;`

result will be placed in task 1

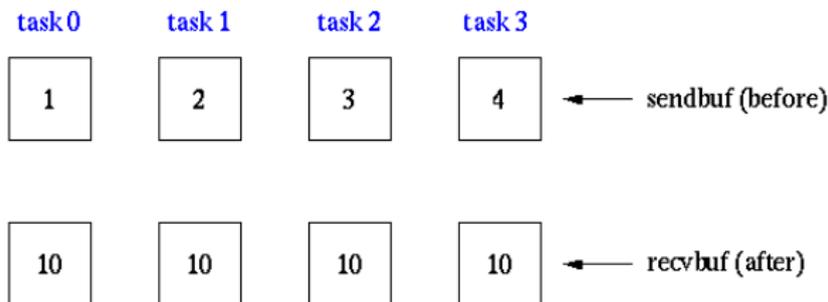
```
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,
           dest, MPI_COMM_WORLD);
```



## Operazioni di riduzione + replicazione

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

```
count = 1;
MPI_Allreduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,
    MPI_COMM_WORLD);
```



```
#define SIZE 4
int numtasks, rank, sendcount, recvcount, source;
float sendbuf[SIZE][SIZE] = {
    {1.0, 2.0, 3.0, 4.0}, {5.0, 6.0, 7.0, 8.0},
    {9.0, 10.0, 11.0, 12.0}, {13.0, 14.0, 15.0, 16.0}  };
float recvbuf[SIZE];
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
if (numtasks == SIZE) {
    source = 1;
    sendcount = recvcount = SIZE;
    MPI_Scatter(sendbuf, sendcount, MPI_FLOAT, recvbuf, recvcount,
               MPI_FLOAT, source, MPI_COMM_WORLD);
    printf("rank=%d Results: %f %f %f %f\n", rank, recvbuf[0],
           recvbuf[1], recvbuf[2], recvbuf[3]);
}
else printf("Must specify %d procs. Terminating.\n", SIZE);
MPI_Finalize();
```



## Misurare il tempo di esecuzione

**double** MPI\_Wtime ( **void** )

- I timer standard (es. POSIX) non sono adeguati per programmi MPI:
  - hanno accuratezza insufficiente
  - non sono portabili
- Ritorna il **tempo locale** (in secondi) trascorso da un istante prefissato
- Non ha senso in termini assoluti
- Occorre misurare intervalli di tempo come **differenza**

## Accuratezza della misurazione

**double** MPI\_Wtick ( **void** )

- Restituisce la risoluzione del timer in secondi



# Misurazione delle prestazioni

## Esempio 5

```
double t0 , t1 , time ;  
...  
t0 = MPI_Wtime () ;  
if ( myproc == 0 ) {  
    MPI_Send ( a , size , MPI_DOUBLE , 1 , 0 , MPI_COMM_WORLD ) ;  
    MPI_Recv ( b , size , MPI_DOUBLE , 1 , 0 , MPI_COMM_WORLD , &st ) ;  
} else {  
    MPI_Recv ( b , size , MPI_DOUBLE , 0 , 0 , MPI_COMM_WORLD , &st ) ;  
  
    /* do something */  
  
    MPI_Send ( b , size , MPI_DOUBLE , 0 , 0 , MPI_COMM_WORLD ) ;  
}  
t1 = MPI_Wtime () ;  
time = 1.e6 * ( t1 - t0 ) ;  
printf ( "That took %f useconds\n" , time ) ;
```



- Lo standard MPI-1 prevede più di 100 primitive!!!
  - Lo standard MPI-2 ne prevede un numero ancor maggiore
  - Molte implementazioni MPI prevedono primitive aggiuntive
- 
- Altre modalità di comunicazione punto-punto
  - ... e le loro versioni non bloccanti
  - Comunicazioni collettive vettoriali
  - Tipi di dato derivati
  - Operazioni di riduzione derivate
  - Gestione dei comunicatori
  - Topologie virtuali
  - ...

- 1 Aspetti avanzati di MPI
  - Deadlock
  - Comunicazioni punto-punto non bloccanti
  - Comunicazioni collettive
  - Misurazione delle prestazioni
  
- 2 Esempio: Moltiplicazione di matrici

Siano date tre matrici dense  $\mathbf{A}$ ,  $\mathbf{B}$  e  $\mathbf{C} \in \mathbb{R}^{m \times m}$

Vogliamo calcolare  $\mathbf{C} \leftarrow \mathbf{A} \times \mathbf{B}$  utilizzando  $n$  processi di calcolo (supponiamo  $m$  divisibile per  $n$ )

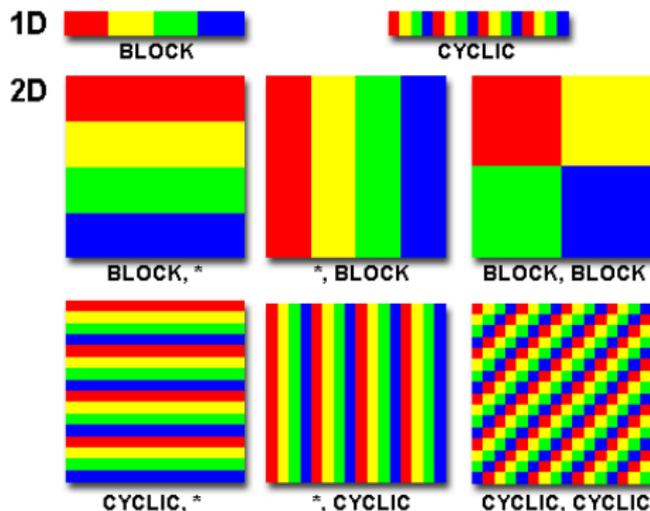
Occorre definire:

- 1 Il formato di input/output
- 2 L'algoritmo parallelo:
  - Come distribuire i dati tra i processi
  - Cosa fa ciascun processo
  - Quali dati occorre scambiare

La scelta di come distribuire i dati tra i processi dipende da:

- Eventuali **vincoli** sul formato in input/output
- Proprietà dell'algoritmo
- **Tradeoff** tra comunicazione e replicazione
- Semplicità implementativa

Esempi di distribuzione di matrici tra 4 processi:



## Distribuzione 2D a blocchi su righe di $\mathbf{A}$ e $\mathbf{C}$ (con replicazione)

- Il processo  $p$  ottiene  $\mathbf{A}(p) \in \mathbb{R}^{\frac{m}{n} \times m}$
- La matrice  $\mathbf{B}$  viene **replicata**
- Calcola  $\mathbf{C}(p) \leftarrow \mathbf{A}(p) \times \mathbf{B}$

## Master $p = 0$

- 1 Legge  $\mathbf{A}$  e  $\mathbf{B}$  da un file
- 2 Distribuisce  $\mathbf{A}(p)$  al processo  $p$
- 3 Replica  $\mathbf{B}$  in ciascun processo
- 4 Calcola  $\mathbf{C}(0) \leftarrow \mathbf{A}(0) \times \mathbf{B}$
- 5 Raccoglie  $\mathbf{C}(p)$  dal processo  $p$
- 6 Scrive  $\mathbf{C}$  in un file

## Distribuzione 2D a blocchi su righe di $\mathbf{A}$ e $\mathbf{C}$ (con replicazione)

- Il processo  $p$  ottiene  $\mathbf{A}(p) \in \mathbb{R}^{\frac{m}{n} \times m}$
- La matrice  $\mathbf{B}$  viene **replicata**
- Calcola  $\mathbf{C}(p) \leftarrow \mathbf{A}(p) \times \mathbf{B}$

## Master $p = 0$

- 1 Legge  $\mathbf{A}$  e  $\mathbf{B}$  da un file
- 2 Distribuisce  $\mathbf{A}(p)$  al processo  $p$  (**MPI\_Scatter**)
- 3 Replica  $\mathbf{B}$  in ciascun processo (**MPI\_Bcast**)
- 4 Calcola  $\mathbf{C}(0) \leftarrow \mathbf{A}(0) \times \mathbf{B}$
- 5 Raccoglie  $\mathbf{C}(p)$  dal processo  $p$  (**MPI\_Gather**)
- 6 Scrive  $\mathbf{C}$  in un file

## Distribuzione 2D a blocchi su righe di $\mathbf{A}$ e $\mathbf{C}$ (con replicazione)

- Il processo  $p$  ottiene  $\mathbf{A}(p) \in \mathbb{R}^{\frac{m}{n} \times m}$
- La matrice  $\mathbf{B}$  viene **replicata**
- Calcola  $\mathbf{C}(p) \leftarrow \mathbf{A}(p) \times \mathbf{B}$

## Slave $p > 0$

- 1 ...
- 2 Ottiene  $\mathbf{A}(p)$  distribuito dal master
- 3 Ottiene  $\mathbf{B}$  replicato dal master
- 4 Calcola  $\mathbf{C}(p) \leftarrow \mathbf{A}(p) \times \mathbf{B}$
- 5 Invia  $\mathbf{C}(p)$  al master che lo raccoglie
- 6 ...

## Distribuzione 2D a blocchi su righe di $\mathbf{A}$ e $\mathbf{C}$ (con replicazione)

- Il processo  $p$  ottiene  $\mathbf{A}(p) \in \mathbb{R}^{\frac{m}{n} \times m}$
- La matrice  $\mathbf{B}$  viene **replicata**
- Calcola  $\mathbf{C}(p) \leftarrow \mathbf{A}(p) \times \mathbf{B}$

## Slave $p > 0$

- 1 ...
- 2 Ottiene  $\mathbf{A}(p)$  distribuito dal master (**MPI\_Scatter**)
- 3 Ottiene  $\mathbf{B}$  replicato dal master (**MPI\_Bcast**)
- 4 Calcola  $\mathbf{C}(p) \leftarrow \mathbf{A}(p) \times \mathbf{B}$
- 5 Invia  $\mathbf{C}(p)$  al master che lo raccoglie (**MPI\_Gather**)
- 6 ...

## Distribuzione 2D a blocchi su righe di $\mathbf{A}$ , $\mathbf{B}$ e $\mathbf{C}$

- Il processo  $p$  ottiene  $\mathbf{A}(p), \mathbf{B}(p) \in \mathbb{R}^{\frac{m}{n} \times m}$
- Calcola  $\mathbf{C}(p) \leftarrow \sum_q \mathbf{A}(p, q) \times \mathbf{B}(q)$
- Il processo  $p$  necessita di  $\mathbf{B}(q)$  durante il calcolo di  $\mathbf{C}(p), \forall q \neq p$

## Per ogni $q = 0 \dots n - 1$

- 1 Se  $p = q$  distribuisco  $\mathbf{B}(p)$  a  $\forall q \neq p$
- 2 Se  $p \neq q$  ottengo  $\mathbf{B}(q)$  da  $q$
- 3 Calcola  $\mathbf{C}(p) \leftarrow \mathbf{C}(p) + \mathbf{A}(p, q) \times \mathbf{B}(q)$

Notare che ogni processo memorizza solo 2 blocchi di  $\mathbf{B}$

## Distribuzione 2D a blocchi su righe di $A$ , $B$ e $C$

- Il processo  $p$  ottiene  $\mathbf{A}(p), \mathbf{B}(p) \in \mathbb{R}^{\frac{m}{n} \times m}$
- Calcola  $\mathbf{C}(p) \leftarrow \sum_q \mathbf{A}(p, q) \times \mathbf{B}(q)$
- Il processo  $p$  necessita di  $\mathbf{B}(q)$  durante il calcolo di  $\mathbf{C}(p), \forall q \neq p$

## Per ogni $q = 0 \dots n - 1$

- 1 Se  $p = q$  distribuisco  $\mathbf{B}(p)$  a  $\forall q \neq p$  (**MPI\_Bcast**)
- 2 Se  $p \neq q$  ottengo  $\mathbf{B}(q)$  da  $q$  (**MPI\_Bcast**)
- 3 Calcola  $\mathbf{C}(p) \leftarrow \mathbf{C}(p) + \mathbf{A}(p, q) \times \mathbf{B}(q)$

Notare che ogni processo memorizza solo 2 blocchi di  $B$



### Distribuzione 2D a blocchi su righe e colonne di $A$ , $B$ e $C$

- Assumiamo  $n = s^2$  e  $n$  è divisibile per  $s$
- I processi sono disposti (virtualmente) in una griglia 2D
- Il processo  $(i, j)$  ottiene  $A(i, j), B(i, j) \in \mathbb{R}^{\frac{m}{s} \times \frac{m}{s}}$
- Calcola  $C(i, j) \leftarrow \sum_k A(i, k) \times B(k, j)$

### Inizializzazione:

- 1 Il processo  $(i, j)$  ruota ciclicamente a sx la riga  $i$  di  $A$  di  $i$  posizioni
- 2 Il processo  $(i, j)$  ruota ciclicamente in su la colonna  $j$  di  $B$  di  $j$  posizioni

### NOTA

Le rotazioni sovrascrivono  $A(i, j)$  e  $B(i, j)$

### Distribuzione 2D a blocchi su righe e colonne di $A$ , $B$ e $C$

- Assumiamo  $n = s^2$  e  $n$  è divisibile per  $s$
- I processi sono disposti (virtualmente) in una griglia 2D
- Il processo  $(i, j)$  ottiene  $A(i, j), B(i, j) \in \mathbb{R}^{\frac{m}{s} \times \frac{m}{s}}$
- Calcola  $C(i, j) \leftarrow \sum_k A(i, k) \times B(k, j)$

### Inizializzazione:

- 1 Il processo  $(i, j)$  ruota ciclicamente a sx la riga  $i$  di  $A$  di  $i$  posizioni (**MPI\_Send** e **MPI\_Recv**)
- 2 Il processo  $(i, j)$  ruota ciclicamente in su la colonna  $j$  di  $B$  di  $j$  posizioni (**MPI\_Send** e **MPI\_Recv**)

### NOTA

Le rotazioni sovrascrivono  $A(i, j)$  e  $B(i, j)$

### Distribuzione 2D a blocchi su righe e colonne di $\mathbf{A}$ , $\mathbf{B}$ e $\mathbf{C}$

- Assumiamo  $n = s^2$  e  $n$  è divisibile per  $s$
- I processi sono disposti (virtualmente) in una griglia 2D
- Il processo  $(i, j)$  ottiene  $\mathbf{A}(i, j), \mathbf{B}(i, j) \in \mathbb{R}^{\frac{m}{s} \times \frac{m}{s}}$
- Calcola  $\mathbf{C}(i, j) \leftarrow \sum_k \mathbf{A}(i, k) \times \mathbf{B}(k, j)$

### Per $s$ volte il processo $(i, j)$ ripete:

- 1 Calcola  $\mathbf{C}(i, j) \leftarrow \mathbf{C}(i, j) + \mathbf{A}(i, j) \times \mathbf{B}(i, j)$
- 2 Ruota ciclicamente a sx la riga  $i$  di  $\mathbf{A}$  di 1 posizione
- 3 Ruota ciclicamente in su la colonna  $j$  di  $\mathbf{B}$  di 1 posizione

### NOTA

Le rotazioni sovrascrivono  $\mathbf{A}(i, j)$  e  $\mathbf{B}(i, j)$

# Moltiplicazione di matrici

Esempio: Algoritmo di Cannon

Distribuzione iniziale:

$A(0,0)$	$A(0,1)$	$A(0,2)$		$B(0,0)$	$B(0,1)$	$B(0,2)$
$A(1,0)$	$A(1,1)$	$A(1,2)$		$B(1,0)$	$B(1,1)$	$B(1,2)$
$A(2,0)$	$A(2,1)$	$A(2,2)$		$B(2,0)$	$B(2,1)$	$B(2,2)$

Dopo l'inizializzazione:

$A(0,0)$	$A(0,1)$	$A(0,2)$		$B(0,0)$	$B(1,1)$	$B(2,2)$
$A(1,1)$	$A(1,2)$	$A(1,0)$		$B(1,0)$	$B(2,1)$	$B(0,2)$
$A(2,2)$	$A(2,0)$	$A(2,1)$		$B(2,0)$	$B(0,1)$	$B(1,2)$



# Moltiplicazione di matrici

Esempio: Algoritmo di Cannon

Step 1:

A(0,0)	A(0,1)	A(0,2)			B(0,0)	B(1,1)	B(2,2)
A(1,1)	A(1,2)	A(1,0)			B(1,0)	B(2,1)	B(0,2)
A(2,2)	A(2,0)	A(2,1)			B(2,0)	B(0,1)	B(1,2)

Step 2:

A(0,1)	A(0,2)	A(0,0)			B(1,0)	B(2,1)	B(0,2)
A(1,2)	A(1,0)	A(1,1)			B(2,0)	B(0,1)	B(1,2)
A(2,0)	A(2,1)	A(2,2)			B(0,0)	B(1,1)	B(2,2)

Step 3:

A(0,2)	A(0,0)	A(0,1)			B(2,0)	B(0,1)	B(1,2)
A(1,0)	A(1,1)	A(1,2)			B(0,0)	B(1,1)	B(2,2)
A(2,1)	A(2,2)	A(2,0)			B(1,0)	B(2,1)	B(0,2)

-  **Per lo standard:**  
`http://www.mpi-forum.org/docs`
-  **Tutorial:**  
`https://computing.llnl.gov/tutorials/mpi`
-  **Per l'implementazione IBM:**  
`http://www.dei.unipd.it/~addetto/manuali\_online/index.html`
-  **In particolare:**  
`http://www.redbooks.ibm.com/redbooks/pdfs/sg245380.pdf`