

# Calcolo Parallelo – Progetto

## *Lezione 2 – MPI*

Michele Schimd

`schimdmi@dei.unipd.it`

2 maggio 2019



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA



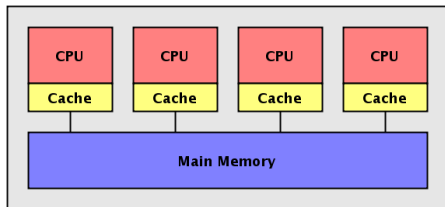
DIPARTIMENTO  
DI INGEGNERIA  
DELL'INFORMAZIONE

- 1 Programmazione parallela
  - Modelli di memoria
  - Modelli di programmazione

- 2 Introduzione a MPI
  - Ambiente di esecuzione
  - Comunicazioni punto-punto

- 3 Aspetti avanzati di MPI
  - Deadlock
  - Comunicazioni punto-punto non bloccanti
  - Riepilogo comunicazioni punto-punto
  - Comunicazioni collettive
  - Misurazione delle prestazioni

# Architettura a memoria condivisa



## Caratteristiche:

- Spazio di indirizzamento globale
- Processori uguali = SMP
- Può essere UMA or NUMA
- Esempi:
  - Server multiprocessore

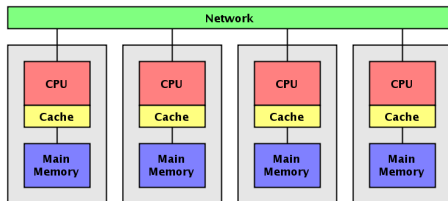
## Pro

- Facilità di programmazione (multithreading/inter-process communications)
- Condivisione dati veloce
- Il SO può controllare il bilanciamento del carico

## Contro

- Collo di bottiglia tra CPU e memoria
- Scalabilità limitata
- Coerenza della cache

# Architetture a memoria distribuita



## Caratteristiche:

- Spazio di indirizzamento locale
- Richiede una rete di interconnessione
- Esempi:
  - Cluster

## Pro

- Alta scalabilità (n. di CPU e memoria)
- Costi limitati (es. Beowulf)

## Contro

- Difficili da programmare
- Occorre distribuire i dati
- Costo delle comunicazioni

## Modelli di programmazione parallela più comuni:

- Shared-memory (memoria condivisa)
  - Multithreading
  - Message passing (**scambio di messaggi**)
  - Data parallel
  - Ibridi
- 
- **Astrazione** delle architetture di memoria
  - Possono (teoricamente) essere realizzati su qualsiasi architettura di memoria:
    - Modello a memoria condivisa su macchine a memoria distribuita
    - Modello a scambio di messaggi su macchine a memoria condivisa (es. MPI)

### Single-Program Multiple-Data

I task:

- Eseguono tutti lo **stesso programma**
- Possono eseguire istruzioni diverse
- Possono usare dati diversi
- Comunicano tra loro e si sincronizzano

Adatto per applicazioni parallele regolari

### Multiple-Program Multiple-Data

- I task eseguono programmi **diversi**
- Adatto per applicazioni parallele irregolari, e.g. Master/Worker

### Single-Program Multiple-Thread

- Modello realizzato da `CUDA` per GPU
- Simile al SPMD

- 1 Programmazione parallela
  - Modelli di memoria
  - Modelli di programmazione

- 2 **Introduzione a MPI**
  - Ambiente di esecuzione
  - Comunicazioni punto-punto

- 3 Aspetti avanzati di MPI
  - Deadlock
  - Comunicazioni punto-punto non bloccanti
  - Riepilogo comunicazioni punto-punto
  - Comunicazioni collettive
  - Misurazione delle prestazioni

## MPI = Message Passing Interface

Con il termine MPI si intendono tre cose:

- 1 **Paradigma** di programmazione a scambio di messaggi
- 2 Insieme di **specifiche** che definiscono una API
- 3 **Libreria** per la programmazione di applicazioni parallele

In realtà sarebbe corretto dire che:

- 1 Message passing è il paradigma di programmazione
- 2 **MPI è una API** (protocollo di comunicazione e semantica delle operazioni)
  - Non è uno standard *de iure*
  - È lo standard *de facto* per la programmazione message passing
  - Se ne occupa un apposito forum: <http://www.mpi-forum.org>
- 3 Esistono varie implementazioni di MPI, es: OpenMPI, MPICH, IBM MPL, ecc.



# Perché MPI?

## Pro

- Indipendenza dall'implementazione
  - Librerie proprietarie, es. IBM MPI
  - Open Source, es. OpenMPI
- Indipendenza dall'architettura di calcolo e di rete
  - Shared-memory, NUMA, distributed memory, multithreading, ecc.
  - Little-endian, big-endian, ecc...
  - Infiniband, TCP, Myrinet, ecc...
- Indipendenza dal linguaggio
  - C/C++, Java, Fortran, ecc...

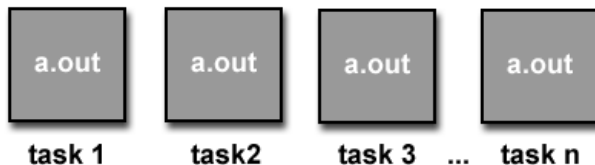
⇒ **Portabilità, scalabilità, efficienza**

## Contro

- Parallelismo troppo a basso livello
  - Controllo diretto su come, quando e tra chi scambiare messaggi
- Difficoltà di utilizzo
  - Sincronizzazione, deadlock

## Versioni principali dello standard

- MPI-1.3 (Detta anche MPI-1)
  - Gestione dell'ambiente di esecuzione
  - Comunicazioni punto-punto
  - Comunicazioni collettive
  - Ambiente di esecuzione statico
- MPI-2.2 (Detta anche MPI-2)
  - I/O parallelo
  - Ambiente di esecuzione dinamico
  - Accesso alla memoria remota
- MPI-3
  - Approvato nel settembre 2012, non ancora sviluppato
  - Operazioni collettive non bloccanti
  - Neighborhood collectives
  - Creazione di gruppi da parte dei processi a run-time



- Il sistema run-time lancia  $n$  **processi**
- Ogni processo:
  - esegue una copia di `a.out` indipendentemente
  - ha il suo spazio di memoria locale
  - **può** essere mappato su un processore diverso
  - ha una sua identità
- `a.out` **può** essere un programma MPI

Ora ci occupiamo solo di programmi MPI

### Definizione

Un **comunicatore** è un oggetto che definisce un gruppo di processi MPI possono comunicare tra di loro

- Ha un nome che lo identifica
  - Ha una dimensione (numero di processi)
  - Ogni processo può essere identificato univocamente
  - I processi sono tra loro equivalenti
- 
- Esiste il comunicatore di default `MPI_COMM_WORLD`
  - Comprende gli  $n$  processi lanciati dal sistema run-time
  - I processi sono identificati con un intero da 0 a  $n - 1$
  - Possono essere definiti altri comunicatori, anche non disgiunti

## Inizializzare MPI

```
int MPI_Init(int *argc, char ***argv)
```

- Deve essere chiamata da tutti i processi **prima** di ogni altra chiamata MPI

## Terminare MPI

```
int MPI_Finalize(void)
```

## Abortire in caso d'errore MPI

```
int MPI_Abort(MPI_Comm comm, int error)
```

- termina tutti i processi associati a un certo comunicatore.

Dato un comunicatore `comm` (per noi è sempre `MPI_COMM_WORLD`):

## Quanti siamo?

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

- `size` è la dimensione del comunicatore

## Chi sono io?

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- `rank` è l'identificatore del processo corrente (da 0 a `size-1`)

```
/* C Example */
#include <stdio.h>
#include <mpi.h>

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

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello_world_from_process_%d_of_%d\n", rank, size);
    MPI_Finalize();
    return 0;
}
```

# Gestione dell'ambiente di esecuzione

## Alcune note sull'Esempio 1

- `MPI_COMM_WORLD` è definito in `mpi.h`
- Ogni comando è eseguito da **ogni** processo **indipendentemente**
- La compilazione produce un unico eseguibile
- Il sistema run-time dipende dall'implementazione e definisce:
  - Se e come si replica l'eseguibile nei nodi di calcolo
  - Come si lanciano i processi
  - Come vengono gestiti lo standard output/error

### Esempio di esecuzione:

```
$ mpcc hello.c
$ poe ./a.out -procs 4 <-- Sul Power 7 in AIX
Hello world from process 1 of 4
Hello world from process 0 of 4 <-- NOTARE L'ORDINE
Hello world from process 2 of 4
Hello world from process 3 of 4
```

Se usate OpenMPI l'esecuzione si fa con `mpirun`

```
$ mpirun -np 4 a.out
```



Quando si esegui un programma parallelo in modalità interattiva

- Il runtime deve sapere quali sono i nodi paralleli su cui può eseguire i processi.
- Nel power 7 esiste un solo nodo che ospita i processi.
- In ogni caso bisogna specificare un nodo di calcolo per ogni processo
- Quindi (in esecuzione interattiva) è necessario predisporre un file `host.list` con tante righe uguali a `power7a` quanti sono i nodi di calcoli (limitative a 32 processi quindi 32 linee).

Se vi dimenticate questo file vi verrà visualizzato un messaggio del tipo

```
ERROR: 0031 808  Hostfile or pool must be used to request
```

## Comunicazione tra **due** processi MPI

- Copia di dati tra spazi di memoria distinti
- Permette di identificare univocamente i messaggi
- Usa tipi di dati propri
- Le primitive possono essere **bloccanti** o **non bloccanti**
- Varie **modalità** di comunicazione:

**Standard:** buffering e sincronizzazione automatici

**Buffered:** buffering utente intermedio

**Sincrona:** rendezvous stretto

**Pronta:** spedizione immediata (no handshaking)

I messaggi sono composti da **due** parti:

**Intestazione:** permette di identificare univocamente il messaggio

**Dati:** ciò che si vuole scambiare tra i processi

# Comunicazioni punto-punto

## Messaggi

L'intestazione è composta da:

**Sorgente:** è l'identificatore (nel comunicatore) del processo che invia il messaggio

**Destinazione:** è l'identificatore (nel comunicatore) del processo che deve ricevere il messaggio

**Tag:** identifica il messaggio (intero da 0 a `MPI_TAG_UB`)

**Comunicatore:** è il contesto di comunicazione (già visto)

I dati sono composti da:

**Tipo:** tipo di dati MPI (tabella)

**Lunghezza:** come numero di  
elementi

**Buffer:** array di elementi consecutivi  
nella memoria utente

MPI datatype	C datatype
<code>MPI_CHAR</code>	signed char
<code>MPI_SHORT</code>	signed short int
<code>MPI_INT</code>	signed int
<code>MPI_LONG</code>	signed long int
<code>MPI_UNSIGNED_CHAR</code>	unsigned char
<code>MPI_UNSIGNED_SHORT</code>	unsigned short int
<code>MPI_UNSIGNED</code>	unsigned int
<code>MPI_UNSIGNED_LONG</code>	unsigned long int
<code>MPI_FLOAT</code>	float
<code>MPI_DOUBLE</code>	double
<code>MPI_LONG_DOUBLE</code>	long double
<code>MPI_BYTE</code>	
<code>MPI_PACKED</code>	

### Spedire un messaggio

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

- Ritorna quando il messaggio è stato copiato dal sistema
- Quando ritorna `buf` può essere sovrascritto
- Non è detto che quando ritorna il messaggio sia stato ricevuto!!!
- Potrebbe essere nella **memoria di sistema** del mittente

### Note sul buffering in spedizione

- Dipende dall'implementazione di MPI
- Può essere controllato dall'utente (lo vedremo prossimamente)
- **Pro:** disaccoppia send e recv, elimina la sincronizzazione
- **Contro:** memoria aggiuntiva, tempo per la copia

### Ricevere un messaggio

```
int MPI_Recv(void* buf, int count, \
             MPI_Datatype datatype, \
             int source, int tag, \
             MPI_Comm comm, MPI_Status *status)
```

- Ritorna quando il messaggio è stato ricevuto nel buffer nello spazio utente
- `count` deve essere  $\geq$  della lunghezza del msg ricevuto
- `source` può essere `MPI_ANY_SOURCE`
- Il `tag` identifica quale messaggio da `source` si vuole ricevere
- `tag` può essere `MPI_ANY_TAG`
- `status` contiene informazioni aggiuntive

### Attenzione

I tipi di dato devono coincidere con quelli della send corrispondente!

# Comunicazioni punto-punto

## Esempio 2

```
#include <mpi.h>
#include <stdio.h>
int main( int argc, char *argv[])
{
    int rank, buf;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        buf = 123456;
        MPI_Send(&buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("%d_has_sent_%d_to_%d\n", rank, buf, 1);
    }
    else if (rank == 1) {
        MPI_Recv(&buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        printf("%d_has_received_%d_from_%d\n", rank, buf, 0);
    }
    MPI_Finalize();
    return 0;
}
```

# Comunicazioni punto-punto

Alcune note sull'Esempio 2

- Stesso codice eseguibile, ma...
- ... processi diversi eseguono parti di codice diverse
- Solo  $p_0$  inizializza `buf`
- $p_1$  ottiene il valore 123456 in `buf` solo dopo aver ricevuto il messaggio da  $p_0$

## Esempio di esecuzione:

```
$ mpcc blocking.c
$ poe ./a.out -procs 8
0 has sent 123456 to 1
1 has received 123456 from 0
```

Cosa fanno i processi  $p_2 \dots p_7$  ?

```
#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;
}
```

```
#include <mpi.h>
#include <unistd.h>
...
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &me);
if (me!=0)
    MPI_Send(&me, 1, MPI_INT, 0, me, MPI_COMM_WORLD);
else {
    sleep(5);
    for (int i=1; i<np; i++) {
        MPI_Recv(&q, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, \
            MPI_COMM_WORLD, &status);
        printf("Received %d from proc %d\n", \
            q, status.MPI_SOURCE);
    }
}
MPI_Finalize();
return 0;
}
```

- Tutti i processi sono in attesa di un messaggio con un Receive bloccante (e nessuno può inviare messaggi quindi non possono sbloccarsi a vicenda)
- Supposizioni errate sul modo di gestire le code di messaggi da parte del sistema runtime
  - Mentre un processo sta compiendo del lavoro e non è in ascolto, potrebbe ricevere messaggi.
  - In genere c'è un buffer per accoglierli temporaneamente.
  - Il comportamento del sistema non è specificato da MPI, è lasciato all'implementazione.
  - Buffer potrebbe esserci o no, contenere uno o più messaggi, avere *eviction policies* diverse...
- Errori di programmazione (destinazioni errate nel Send o sorgenti errate nel Receive, tag diversi in coppie Send/Receive che devono comunicare)

- 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 (bisogna controllare manualmente lo stato delle operazioni)
  - Può anche aumentare l'efficienza (possono eventualmente fare altro lavoro mentre prima di mettermi in attesa)
- Ostrich Algorithm

- 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 (bisogna controllare manualmente lo stato delle operazioni)
  - Può anche aumentare l'efficienza (possono eventualmente fare altro lavoro mentre prima di mettermi in attesa)
- Ostrich Algorithm    **cercate sempre di gestire le potenziali situazioni di deadlock note**

# 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;
}
```

- Implementazioni diverse del meccanismo di bufferizzazione
  - Leggere le specifiche dell'implementazione, in genere meglio non assumere che il buffer sia molto grande
- Ricezione *in order*
  - No se chiamate in thread diversi

## 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 **sola** 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();
```

### ● Asynchronous

- **Blocking send** (`MPI_Send`): ritorna quando è possibile sovrascrivere il buffer coi dati da inviare. Nota che i dati potrebbero essere nella memoria di sistema locale e non ancora realmente inviati.
- **Blocking receive** (`MPI_Recv`): ritorna quando il buffer di ricezione contiene dati validi.
- **Non-blocking send/receive** (`MPI_Isend` / `MPI_Irecv`): ritornano subito, non è detto che i dati siano già stati letti/scritti. È necessario controllare o mettersi in attesa programmaticamente (`MPI_Wait`/`MPI_Test`).

### ● Synchronous

- **Synchronous blocking send** (`MPI_Ssend`): ritorna solo quando la destinazione inizia a ricevere.
- **Synchronous nonblocking send** (`MPI_Issend`): ritorna subito ma devo usare `MPI_Wait`/`MPI_Test` per monitorare se i dati sono arrivati al ricevitore.

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? Nuovo gruppo...
- Solo tipi di dati MPI predefiniti (~solo array di tipi omogenei)

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

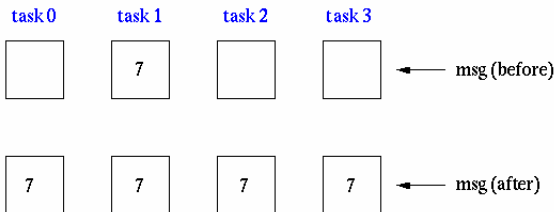
### Replicare i dati

```
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

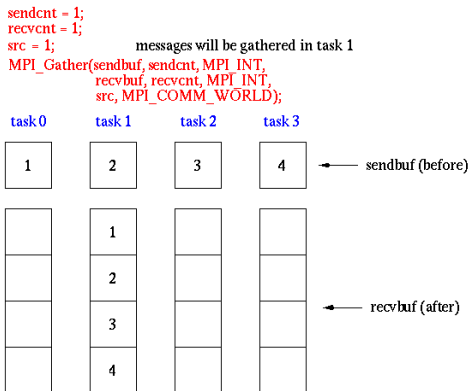






### Raccogliere i dati

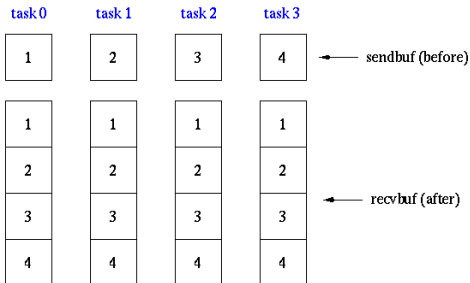
```
int MPI_Gather(void* sendbuf, int sendcount, \
MPI_Datatype sendtype, void* recvbuf, int recvcount, \
MPI_Datatype recvtype, int root, MPI_Comm comm)
```



### 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);
```

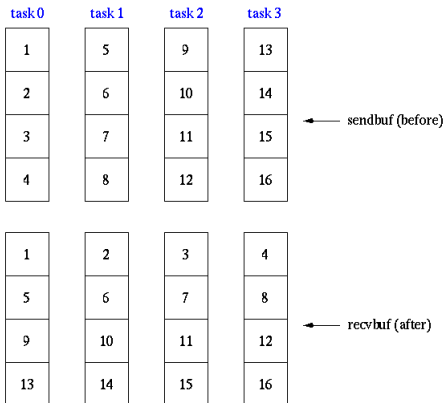


### Scambio da tutti a tutti

```
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);`

task 0

task 1

task 2

task 3



← sendbuf (before)

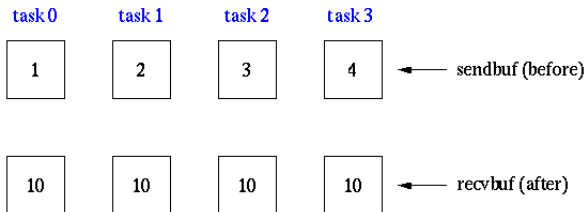


← recvbuf (after)

## 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 = 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;
    if (rank==source) {
        sendbuf = loadSquareMatrix(SIZE);
    }
    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 dal 01/01/1970
- 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 (es. 0.000001)



# 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);
```

# C'è altro?

- 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
  - ...

-  **Per lo standard:**  
<http://www.mpi-forum.org/docs>
-  **Tutorial:**  
<https://computing.llnl.gov/tutorials/mpi>
-  **Ottimizzazione su Power7:**  
<http://www.redbooks.ibm.com/abstracts/sg248079.html>  
<http://www.redbooks.ibm.com/redbooks/pdfs/sg245380.pdf>