0.1 INTRODUZIONE

Al giorno d'oggi il calcolo parallelo consiste nell'esecuzione contemporanea del codice su più processori al fine di aumentare le prestazioni del sistema. In questo modo si superano molti dei limiti del calcolo sequenziale permettendo di aumentare le dimensioni dei problemi affrontabili e riducendo i tempi richiesti per risolverli.

La necessità di creare uno standard che garantisse la portabilità del codice tra le diverse architetture parallele (dai cluster di workstations ai supercalcolatori) spinse un gruppo di rivenditori, scienziati, programmatori e utenti a fondare, nei primi anni Novanta, l'MPI forum (www.mpi-forum.org). Si aprirono subito le discussioni e nacque così l'MPI (message-passing interface) e, come il nome stesso lascia intendere, si tratta di uno standard che permette di coordinare le attività dei processi attraverso lo scambio di messaggi. Non è tuttavia un linguaggio di programmazione, infatti definisce semplicemente termini,convenzioni e un'insieme di routine richiamabili da linguaggi come Fortran, C, C++ . Nulla comunque vieta di sfruttarle in altri linguaggi ma la nostra guida si occuperà della sola programmazione in linguaggio C attraverso l'uso delle MPICH2 (http://www-unix.mcs.anl.gov/mpi/mpich2/), delle librerie che offrono un'implementazione open-source, e quindi gratuita, dello standard MPI. Esse comprendono inoltre il software che gestisce i processi e le loro comunicazioni.

Questa guida, dunque, cerca di fornire una panoramica sulla programmazione parallela basata sul principio message-passing, mentre rimanda ad altro contesto (http://newk.alma.unibo.it/oscar/cmplx/complx03.htm) per una panoramica delle basi teoriche su cui si fondano i concetti di programmazione parallela. I temi che trattiamo sono facilmente comprensibili da chiunque abbia conoscenze di base riguardo C/C++ e non richiede una conoscenza particolare dello scambio di messaggi tra processi.

Capitolo 1

Programmazione di base

1.1 Let's get it started

Iniziamo subito il nostro viaggio all'interno della libreria MPI esaminando il più classico dei codici sin dai tempi del 'The C Programming Language' di Kerninghan e Ritchie's, ovvero un programma che stampi 'Hello, world!'

```
#include <mpi.h>
#include <stdio.h>

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

      MPI_Init(&argc,&argv);

      printf("Hello, world!\n");

      MPI_Finalize();

}
```

Le cose che saltano subito all'occhio sono tre:

- l'inclusione della libreria `mpi.h`, ovviamente obbligatoria per poter usufruire delle funzioni e delle macro MPI. Si consiglia vivamente di includerla prima di ogni altra libreria, in quanto un caso diverso può generare degli errori in fase di compilazione.
- É obbligatorio che il nostro programma MPI abbia un inizio ed una fine. L'inizio è dato dalla chiamata alla funzione `MPI_Init`

  ```
  int MPI_Init ( int * argc_ptr, char ** argv_ptr[] )
  ```

  che informa il sistema operativo che si sta eseguendo un programma MPI e gli consente di eseguire tutte le inizializzazioni necessarie al caso, mentre la conclusione del programma è data dalla chiamata alla funzione `MPI_Finalize ( void )`.

Un algoritmo parallelo che non necessita di scambio di messaggi si definisce come 'imbarazzantemente parallelo' (embarassingly parallel) e questo è il nostro caso: lo scambio di informazioni tra i processori/processi, infatti, è nullo.

Una volta compilato(link ad una pagina che insegna a compilare) e lanciato(link a pagina in cui si dice come lanciare) il programma, si otterrà a schermo una sequenza di 'Hello, world!' pari al numero di processi con cui è stato lanciato il programma. E questo è il nostro primo programma parallelo!

Una domanda sorge ora spontanea: come fa un processo a conoscere la propria identità? Nella MPI, i processi coinvolti nell'esecuzione di un programma parallelo sono identificati da una sequenza di numeri interi non negativi detti *rank*.

Se abbiamo un numero p di processi che eseguono un programma, i processi avranno allora un rank che va da 0 a $p-1$. La funzione MPI che ci viene incontro per risolvere questo problema à la `MPI_Comm_rank()`:

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

Questa funzione restituisce in result il rank del processo che l'ha chiamata. L'argomento *comm* è detto comunicatore (communicator), ed essenzialmente definisce un insieme di processi che possono comunicare insieme: un particolare comunicatore è `MPI_COMM_WORLD`, predefinito all'interno della MPI, che consiste nell'insieme di tutti i processi coinvolti nell'esecuzione del programma parallelo.

Rimanendo in quest'ottica, un'altra funzione estremamente utile risulta essere la `MPI_Comm_size()`

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

che restituisce in size il numero totale di processi che sono stati allocati. Diamo subito uno sguardo ad una applicazione pratica delle funzioni appena viste:

```
#include <mpi.h>
#include <stdio.h>

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

      int rank, totprocessi;

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

      printf("Hello world dal processo %d di %d\n", rank, totprocessi);

      MPI_Finalize();
}
```

Quando lanciamo il programma con 5 processi, l'output a schermo sarà:

```
Hello world dal processo 0 di 5
Hello world dal processo 1 di 5
Hello world dal processo 2 di 5
Hello world dal processo 3 di 5
Hello world dal processo 4 di 5
```

Va notato, a solo scopo illustrativo, che l'output a schermo potrà non essere sempre ordinato dato che più processi possono richiedere nello stesso momento la scrittura a video, ed è il sistema operativo che arbitrariamente ne sceglierà l'ordine.

Siamo pronti per una osservazione fondamentale: ogni processo implicato nell'esecuzione MPI esegue lo stesso identico binario compilato, dunque ogni processo riceve esattamente le stesse istruzioni da eseguire; in funzione del proprio rank, ogni processo eseguirà, mediante costrutti if o di altra natura, la parte di codice a lui assegnata.

1.2 Messaggiamo?

MPI, come abbiamo già detto, si fonda sul principio dello scambio di messaggi tra processi. É dunque necessario introdurre quanto prima le funzioni basilari di comunicazione tra processi, ovvero `MPI_Send()` ed `MPI_Recv()`. Com'è facile intuire, la prima funzione si occupa di mandare ad un processo delle informazioni, la seconda di riceverle; a tale scopo il sistema deve affiancare all'informazione da mandare o da ricevere una serie di parametri:

- il rank del processo che riceve
- il rank del processo che manda
- un tag utile a distinguere dati differenti provenienti dallo stesso processo
- un comunicatore

Questi parametri possono essere utilizzati dal ricevente per distinguere i diversi messaggi in arrivo. In particolare, il rank del processo che ha inviato l'informazione potrà essere utilizzato per distinguere i messaggi mandati da processi differenti, mentre il *tag*, un numero intero compreso tra 0 e 32767, potrà servire a distinguere messaggi differenti che hanno lo stesso processo mittente.

Per quanto riguarda il comunicatore, che come già anticipato rappresenta un gruppo di processi che possono scambiarsi messaggi, utilizzeremo prettamente `MPI_COMM_WORLD`, ovvero l'insieme dei processi che hanno già lanciato il programma nel momento in cui l'esecuzione dello stesso inizia.

1.2.1 MPI_Send ed MPI_Recv

Esaminiamo la sintassi delle due funzioni di comunicazione

`MPI_Send ed MPI_Recv:`

```
int MPI_Send ( void * message, int count, MPI_Datatype datatype,/
int dest, int tag, MPI_Comm comm )

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

Come fanno la maggior parte delle funzioni nelle librerie standard del C, anche le funzioni MPI restituiscono un codice intero di errore: generalmente ignoreremo questi valori di ritorno.

Il contenuto del messaggio è memorizzato nell'indirizzo puntato da `message` ed è un array contenente un certo numero `count` di elementi di tipo `datatype`. I corrispondenti tipi in C (se esistono) dei tipi predefiniti MPI sono elencati di seguito:

MPI_Datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned long int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Va notato che la quantità di spazio allocata per inviare il messaggio non deve essere per forza l'ammontare esatto di memoria allocata per ricevere il messaggio: l'importante è che il buffer destinato a ricevere i dati sia pari o superiore a quello che invierà. Nell'esecuzione di un programma parallelo ha senso pensare che non sia sempre possibile stabilire a priori la dimensione di un messaggio che verrà inviato, le MPI quindi ne consentono sempre la ricezione, purché sia disponibile lo spazio necessario per accoglierlo; in caso contrario, verrà segnalato un errore di overflow.

I parametri `source` e `dest` sono rappresentati invece dal rank del processo mittente o dal rank del processo destinatario. E' anche possibile utilizzare come parametro `source` una *wildcard* (un carattere jolly) rappresentata dalla costante predefinita `MPI_ANY_SOURCE` che consente al processo di rendersi disponibile per ricevere un messaggio da un qualsiasi processo contenuto in `comm`, invece che da uno in particolare.

Allo stesso modo è utilizzabile in `MPI_Recv` la wildcard `MPI_ANY_TAG` per il campo `tag`.

L'ultimo parametro della funzione `MPI_Recv`, `status`, punta ad un record contenente il rank del processo mittente (`source`) ed il tag, ed è utile nel caso in cui questi due campi fossero stati riempiti con le wildcard appena descritte.

Il programma seguente include le funzioni appena viste per realizzare una comunicazione "a cascata": il processo 0 si occupa di ricevere come input dall'utente un numero che verrà inviato al processo 1, il quale a sua volta lo riceverà e lo manderà al processo successivo, e questo avverrà di volta in volta fino all'avvenuta ricezione del numero da parte dell'ultimo processo.

```
#include <mpi.h>
#include <stdio.h>

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

```
            int id;
            int n;
            int num;
            MPI_Status status;

            MPI_Init(&argc, &argv);
            MPI_Comm_size(MPI_COMM_WORLD, &n);
            MPI_Comm_rank(MPI_COMM_WORLD, &id);
            MPI_Barrier(MPI_COMM_WORLD);

            if (id==0) {
                   printf("Inserire il numero da inviare: ");
                   scanf("%d", &num);
                   MPI_Send (&num,1,MPI_INT,id+1,0,MPI_COMM_WORLD);
                   printf("%d: ho inviato %d\n", id, num);
            } else {
                   MPI_Recv (&num,1,MPI_INT,id-1,0,MPI_COMM_WORLD,&status);
                   printf("%d: ho ricevuto %d\n", id, num);
                   if (id!=n-1) {
                          MPI_Send (&num,1,MPI_INT,id+1,0,MPI_COMM_WORLD);
                          printf("%d: ho inviato %d\n", id, num);
                   }
            }
            MPI_Finalize();
            return 0;
}
```

L'output di questo programma ci aiuta maggiormente a capirne la logica di funzionamento:

```
$ mpiexec -n 5 pippo
Inserire il numero da inviare: 100
0: Ho inviato 100
3: Ho ricevuto 100
3: Ho inviato 100
1: Ho ricevuto 100
1: Ho inviato 100
2: Ho ricevuto 100
2: Ho inviato 100
4: Ho ricevuto 100
```

Ricordando sempre che l'ordine con cui vengono soddisfatte le richieste di stampa a video da parte dei vari processi non è sempre attendibile e concorde con quello reale in quanto dipende dalle politiche di gestione dei processi del sistema operativo.

Una funzione che è stata introdotta in quest'ultimo programma ma non è ancora stata descritta è la funzione `MPI_Barrier()`:

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

la quale blocca l'esecuzione di ogni processo e la riprende nel momento in cui tutti i membri appartenenti a `comm` hanno chiamato la funzione. Essa dunque funge da strumento di sincronizzazione e può essere impiegata in più ambiti.

1.2.2 Il problema del Deadlock e come evitarlo

Un problema ricorrente, e non sempre facilmente individuabile nei programmi paralleli, è quello del `deadlock`. Si tratta di una situazione in cui 2 (o più) processi si bloccano a vicenda aspettando che uno esegua una certa azione che serve all'altro, e viceversa. Analizziamo insieme il seguente codice:

```
#include <mpi.h>
#include <stdio.h>

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

           int id;
           int n;
           int a[2], b[2];
           MPI_Status status;

           MPI_Init(&argc, &argv);
           MPI_Comm_size(MPI_COMM_WORLD, &n);
           MPI_Comm_rank(MPI_COMM_WORLD, &id);
           MPI_Barrier(MPI_COMM_WORLD);

           if (id==0) {
                  a[0]=3;
                  a[1]=5;
                  MPI_Recv (&b,2,MPI_INT,1,11,MPI_COMM_WORLD,&status);
                  MPI_Send (&a,2,MPI_INT,1,10,MPI_COMM_WORLD);
           } else if (id==1) {
                  a[0]=7;
                  a[1]=9;
                  MPI_Recv (&b,2,MPI_INT,0,10,MPI_COMM_WORLD,&status);
                  MPI_Send (&a,2,MPI_INT,0,11,MPI_COMM_WORLD);
           }
           printf("%d: b[0]=%d , b[1]=%d\n", id, b[0], b[1]);
           MPI_Finalize();
           return 0;
}
```

Se proviamo a eseguire questo programma (ha senso farlo solo con 2 processi) notiamo che nessuno dei due processi riesce a procedere. Entrambi si preparano a ricevere un messaggio dall'altro e qui rimangono bloccati. Questo accade perchè la funzione `MPI_Recv()`, come anche l'`MPI_Send()`, è bloccante. Ciò significa che il processo chiamante rimane in attesa del loro completamento. Per quanto riguarda `MPI_Send()` il completamento avviene quando i dati sono stati inviati e possono essere sovrascritti senza andare a modificare il messaggio. Questo tuttavia non significa che il messaggio è già stato mandato. Può accadere, in

particolare se di grandi dimensioni, che sia stato solamente copiato in un buffer e venga inviato successivamente. Il completamento della `MPI_Recv()`, invece, si ha quando i dati sono stati ricevuti e possono essere utilizzati

Per risolvere il problema la prima idea che ci viene in mente potrebbe essere quella di invertire le `MPI_Recv()` con le `MPI_Send()` in questo modo:

```
if (id==0) {
       a[0]=3;
       a[1]=5;
       MPI_Send (&a,2,MPI_INT,1,10,MPI_COMM_WORLD);
       MPI_Recv (&b,2,MPI_INT,1,11,MPI_COMM_WORLD,&status);
} else if (id==1) {
       a[0]=7;
       a[1]=9;
       MPI_Send (&a,2,MPI_INT,0,11,MPI_COMM_WORLD);
       MPI_Recv (&b,2,MPI_INT,0,10,MPI_COMM_WORLD,&status);
}
```

Questa soluzione, tuttavia, anche se corretta dal punto di vista logico, non sempre garantisce di evitare il deadlock. Poichè la comunicazione è realizzata attraverso un buffer dove l'`MPI_Send()` copia i dati da inviare, il programma funziona senza problemi solamente se questo buffer è in grado di contenerli tutti. In caso contrario si ha un deadlock: il mittente non può completare l'invio dato che il buffer è impegnato e il destinatario non può ricevere essendo a sua volta bloccato da una `MPI_Send()` non ancora completata.

A questo punto la soluzione che permette di evitare in ogni caso il deadlock risulta ovvia. É sufficiente invertire le funzioni di invio e ricezione in modo da renderle asimmetriche.

```
if (id==0) {
       a[0]=3;
       a[1]=5;
       MPI_Send (&a,2,MPI_INT,1,10,MPI_COMM_WORLD);
       MPI_Recv (&b,2,MPI_INT,1,11,MPI_COMM_WORLD,&status);
} else if (id==1) {
       a[0]=7;
       a[1]=9;
       MPI_Recv (&b,2,MPI_INT,0,10,MPI_COMM_WORLD,&status);
       MPI_Send (&a,2,MPI_INT,0,11,MPI_COMM_WORLD);
}
```

1.2.3 MPI_Sendrecv

La soluzione al deadlock appena vista non è comunque l'unica. Esiste ad esempio una funzione particolare che unifica il un'unica chiamata l'invio di un messaggio a un determinato processo e la ricezione di un'altro messaggio, proveniente da un'altro processo. É la `MPI_Sendrecv()`:

```
int MPI_Sendrecv ( void * sendbuf,
```

```
                int sendcount,
                MPI_Datatype senddatatype,
                int dest,
                int sendtag,
                void * recvbuf,
                int recvcount,
                MPI_Datatype recvtype,
                int source,
                int recvtag,
                MPI_Comm comm,
                MPI_Status *status )
```

Come si può vedere i parametri richiesti sono gli stessi di una `MPI_Send()` e di una `MPI_Recv()`. Anche in questo caso la funzione è bloccante ma rispetto alle due già viste in precedenza offre il vantaggio di lasciare al sottosistema di comunicazione il compito di controllare le dipendenze tra invii e ricezioni, evitando così il deadlock. Ecco come si può applicare all'esempio precedente:

```
#include <mpi.h>
#include <stdio.h>

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

           int id;
           int n;
           int a[2], b[2];
           MPI_Status status;

           MPI_Init(&argc, &argv);
           MPI_Comm_size(MPI_COMM_WORLD, &n);
           MPI_Comm_rank(MPI_COMM_WORLD, &id);
           MPI_Barrier(MPI_COMM_WORLD);

           if (id==0) {
                  a[0]=3;
                  a[1]=5;
                  MPI_Sendrecv (&a,2,MPI_INT,1,10,   &b,2,MPI_INT,1,11,/
                                MPI_COMM_WORLD,&status);
           } else if (id==1) {
                  a[0]=7;
                  a[1]=9;
                  MPI_Sendrecv (&a,2,MPI_INT,0,11,   &b,2,MPI_INT,0,10,/
                                MPI_COMM_WORLD,&status);
           }
           printf("%d: b[0]=%d , b[1]=%d\n", id, b[0], b[1]);
           MPI_Finalize();
           return 0;
}
```

1.2.4 Comunicazioni bloccanti e non bloccanti: MPI_Isend e MPI_Irecv

Come abbiamo potuto constatare, le funzioni viste finora sono tutte bloccanti. Ciò significa che il ritorno da una procedura avviene solamente al termine del processo di comunicazione. Esistono però routine che, dopo essere state invocate, ritornano subito il controllo al programma chiamante, permettendogli di svolgere nel frattempo altre operazioni. Queste funzioni si dicono non bloccanti e si usano per evitare tempi morti oppure i deadlock. Ne esistono diverse, ma quelle di maggiore importanza sono la `MPI_Isend()` e la `MPI_Irecv()`. Analizziamole insieme:

```
int MPI_Isend ( void * sendbuf, int count, MPI_Datatype datatype,/
int dest, int tag, MPI_Comm comm, MPI_Request * req )

int MPI_Irecv ( void * recvbuf, int count, MPI_Datatype datatype,/
int source, int tag, MPI_Comm comm, MPI_Request * req )
```

Come si può osservare hanno gli stessi parametri delle funzioni standard bloccanti a differenza del campo `MPI_Request` che serve per poter verificare quando il processo di comunicazione è completo. Per completamento di una comunicazione (*completion*) si intende il momento al partire dal quale si può accedere in modo sicuro alle locazioni di memoria usate nel trasferimento dei dati. Questo controllo spetta dunque al programmatore, dato che le funzioni non bloccanti non se ne occupano. Inoltre, rispetto alla send standard, tra i parametri dell'`MPI_Irecv()` non compare il campo *MPI_Status*.

Usando queste routine non bloccanti il nostro esempio diventa:

```
#include <mpi.h>
#include <stdio.h>

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

            int id;
            int n;
            int a[2], b[2];
            MPI_Request req1, req2;
            MPI_Status status;

            MPI_Init(&argc, &argv);
            MPI_Comm_size(MPI_COMM_WORLD, &n);
            MPI_Comm_rank(MPI_COMM_WORLD, &id);
            MPI_Barrier(MPI_COMM_WORLD);

            if (id==0) {
                   a[0]=3;
                   a[1]=5;
                   MPI_Isend (&a,2,MPI_INT,1,10,MPI_COMM_WORLD,&req1);
                   MPI_Irecv (&b,2,MPI_INT,1,11,MPI_COMM_WORLD,&req2);
            } else if (id==1) {
                   a[0]=7;
```

```
                a[1]=9;
                MPI_Irecv (&b,2,MPI_INT,0,10,MPI_COMM_WORLD,&req1);
                MPI_Isend (&a,2,MPI_INT,0,11,MPI_COMM_WORLD,&req2);
        }
        printf("%d: b[0]=%d , b[1]=%d\n", id, b[0], b[1]);
        MPI_Finalize();
        return 0;
}
```

Se proviamo a eseguire questo codice otteniamo un output di questo tipo:

```
0: b[0]=-1073745080 , b[1]=134795163
1: b[0]=-1073745080 , b[1]=134795163
```

Sicuramente non è ciò che ci si aspetta ma cerchiamo di capirne il motivo. Usando delle routine non bloccanti entrambi i processi chiamano le funzioni di comunicazione dopodichè continuano nell'esecuzione del codice. L'istruzione sucessiva che incontrano è quella di visualizzare il contenuto delle locazioni di memoria b[0] e b[1] dove dovrebbero ricevere i dati. In realtà questa operazione avviene prima che il processo di comunicazione sia completato perciò l'output risulta costituito dai valori di inizializzazione dei buffer. Ecco che torna in gioco quello che avevamo detto prima riguardo la verifica del completamento di una operazione. La funzione che ci permette di far questo è la `MPI_Wait()`:

```
int MPI_Wait ( MPI_Request *req, MPI_Status *status )
```

Essa pone il programma in stato di attesa finchè una send o una receive indicata dal parametro `MPI_Request()` ha termine. Per esempio una send non bloccante immediatamente seguita da una `MPI_Wait()` corrisponde a una send bloccante. Inseriamo ora questa nuova routine nel nostro programma:

```
if (id==0) {
                a[0]=3;
                a[1]=5;
                MPI_Isend (&a,2,MPI_INT,1,10,MPI_COMM_WORLD,&req1);
                MPI_Irecv (&b,2,MPI_INT,1,11,MPI_COMM_WORLD,&req2);
                MPI_Wait (&req1,&status);
                MPI_Wait (&req2,&status);
        } else if (id==1) {
                a[0]=7;
                a[1]=9;
                MPI_Irecv (&b,2,MPI_INT,0,10,MPI_COMM_WORLD,&req1);
                MPI_Isend (&a,2,MPI_INT,0,11,MPI_COMM_WORLD,&req2);
                MPI_Wait (&req1,&status);
                MPI_Wait (&req2,&status);
        }
```

Ora l'output è quello che ci aspettavamo e cioè:

```
0: b[0]=7 , b[1]=9
1: b[0]=3 , b[1]=5
```

1.2.5 Comunicazioni collettive

Durante lo sviluppo dei nostri programmi paralleli, potremmo trovarci spesso nella situazione di dover condividere tra più processi il valore di una certa variabile in fase di *run-time* oppure effettuare determinate operazioni su variabili che ogni processo mette a disposizione (con valori presumibilmente differenti): per risolvere situazioni di questo tipo dovremmo ricorrere ad esempio ad artigianali metodi di comunicazione ad albero (il processo 0 invia il dato ai processi 1 e 2, i quali si occuperanno di inviarli ai processi 3, 4, 5 e 6 e via così), se non fosse che le librerie MPI provvedono a fornire funzioni ideali per lo scambio o la fruizione di informazioni tra più processi, chiaramente ottimizzate per la macchina in cui vengono eseguite.

1.2.6 Il Broadcast

Un metodo di comunicazione che coinvolge tutti i processi appartenenti ad un comunicatore viene chiamato `comunicazione collettiva`. Di conseguenza, una comunicazione collettiva coinvolge generalmente più di due processi.

Chiamiamo *broadcast* la comunicazione collettiva in cui un singolo processo invia ad ogni altro processo lo stesso dato.

Nelle MPI la funzionalità di broadcast è offerta dalla funzione `MPI_Bcast()`:

```
int MPI_Bcast ( void * message, int count, MPI_Datatype datatype,/
int  root, MPI_Comm comm )
```

Questa funzione semplicemente manda le informazioni contenute in `message` dal processo `root` ad ogni altro processo appartenente al comunicatore `comm`: ogni processo deve però chiamarla con gli stessi valori di `root` e di `comm`. Com'è facile intuire, i parametri `count` e `datatype` hanno la stessa funzione che abbiamo descritto per le funzioni `MPI_Send()` ed `MPI_Recv()`, tenendo a mente però che una `MPI_Recv()` non può ricevere i dati inviati tramite un broadcast.

Nelle comunicazioni collettive, MPI obbliga a far si che `count` e `datatype` siano gli stessi per tutti i processi del comunicatore: questo perché in alcune comunicazioni collettive un singolo processo riceverà dati da tutti gli altri processi, ed in tal caso il programma solo per tener traccia dell'ammontare di dati ricevuti dovrebbe riservare un intero array.

Vediamo ora un esempio in cui viene usata la funzione di broadcast

```
#include <mpi.h>
#include <stdio.h>

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

           int id;
           int n;
           int var_to_share;

           MPI_Init(&argc, &argv);
           MPI_Comm_size(MPI_COMM_WORLD, &n);
           MPI_Comm_rank(MPI_COMM_WORLD, &id);
```

```
            if (id==0) {
                   var_to_share=5;
            }
            MPI_Bcast (&var_to_share,1,MPI_INT,0,MPI_COMM_WORLD);
            printf("Processo %d: Ho a disposizione il valore %d;\n", \
                                 id, var_to_share);
            MPI_Finalize();
            return 0;
}
```

Come si puó osservare il processo 0 modifica il valore della variabile che deve poi essere reso disponibile a tutti gli altri processi. Ognuno di questi esegue dunque la funzione MPI_Bcast con gli stessi identici parametri producendo il seguente output:

```
Processo 8: Ho a disposizione il valore 5;
Processo 9: Ho a disposizione il valore 5;
Processo 6: Ho a disposizione il valore 5;
Processo 5: Ho a disposizione il valore 5;
Processo 4: Ho a disposizione il valore 5;
Processo 7: Ho a disposizione il valore 5;
Processo 0: Ho a disposizione il valore 5;
Processo 1: Ho a disposizione il valore 5;
Processo 2: Ho a disposizione il valore 5;
Processo 3: Ho a disposizione il valore 5;
```

Si nota che il valore 5, disponibile inizialmente solo al processo 0 è ora disponibile a tutti i processi.

Indice