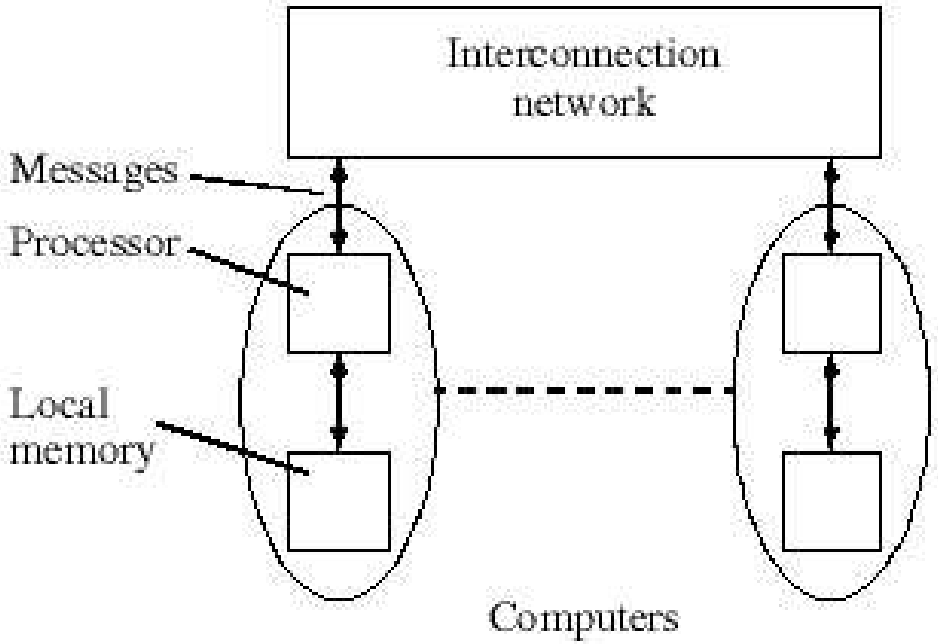
Calcolo su macchine a memoria distribuita

MPI

Architetture *Message–Passing*

Calcolatori distinti connessi attraverso un *network* di comunicazione.

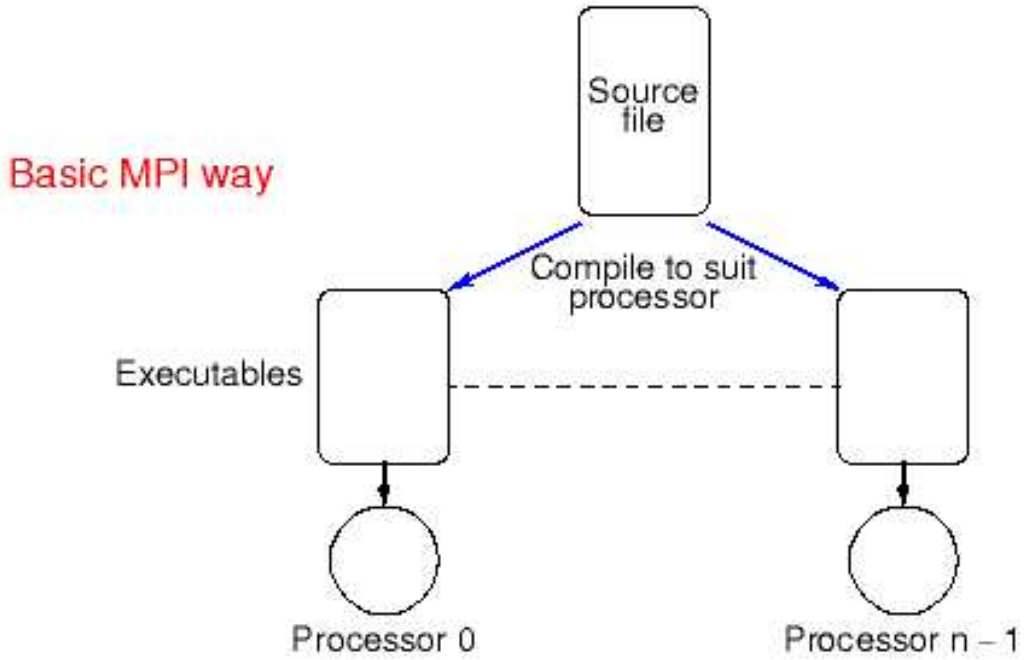


Message–Passing:concetto

- Paradigma sequenziale:
  - singolo processore
  - accede alla sua memoria
  - Il programma viene eseguito su quel processore.
- Paradigma *Message–Passing:*
  - più processori (più sequenziali insieme)
  - ognuno accede alla propria memoria
  - il programma viene eseguito su ogni processore
  - I processori si scambiano informazioni solo attraverso messaggi.

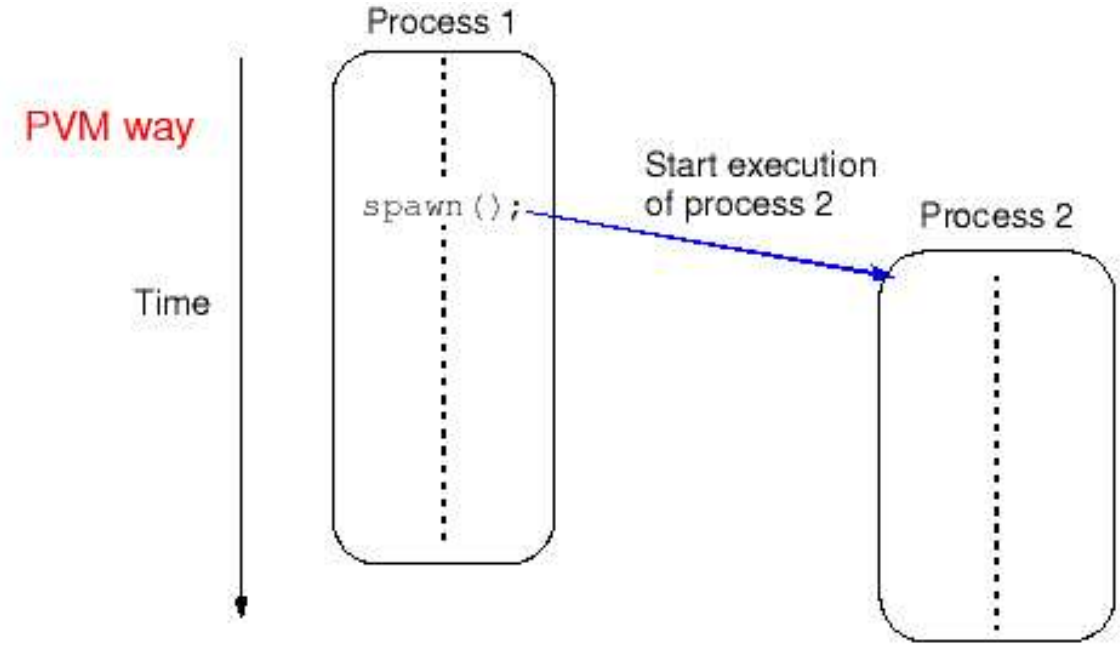
Il modello *SPMD*

- Single Program Multiple Data.
- Tutti i processori eseguono lo stesso programma.
- Processori differenti eseguono parti differenti (tramite istruzioni).
- Tutti gli eseguibili partono insieme (creazione dei processi statica).



Il modello *MPMD*

- Multiple Program Multiple Data.
- Programmi diversi sono assegnati a processori differenti.
- Un processore (*master*) esegue il processo ed istanzia gli altri processi (*slave*).
- Gli eseguibili possono partire in momenti diversi (creazione dei processi dinamica).



Messaggi: cosa sono e cosa contengono?

Scambio di informazioni tra i processori.
Contengono i valori dei dati ed inoltre...

- Quale processore sta inviando i dati.
- Dove il processore sta inviando i dati.
- Che tipo di dati sta inviando.
- Quanti dati sta inviando.
- Quale(i) processore(i) sta(nno) ricevendo i dati.
- Dove saranno posizionati i dati ricevuti.
- Quanti dati il processore si aspetta di ricevere.
- ...

Comunicazioni: tra chi?

- Punto a punto
  Coinvolti solo due processori: uno invia l'altro riceve.
  Solo questi due processori devono conoscere le caratteristice del messaggio.

- Collettive
  Coinvolge tutti i processori appartenenti ad uno specifico gruppo.

Punto a punto Sincrono

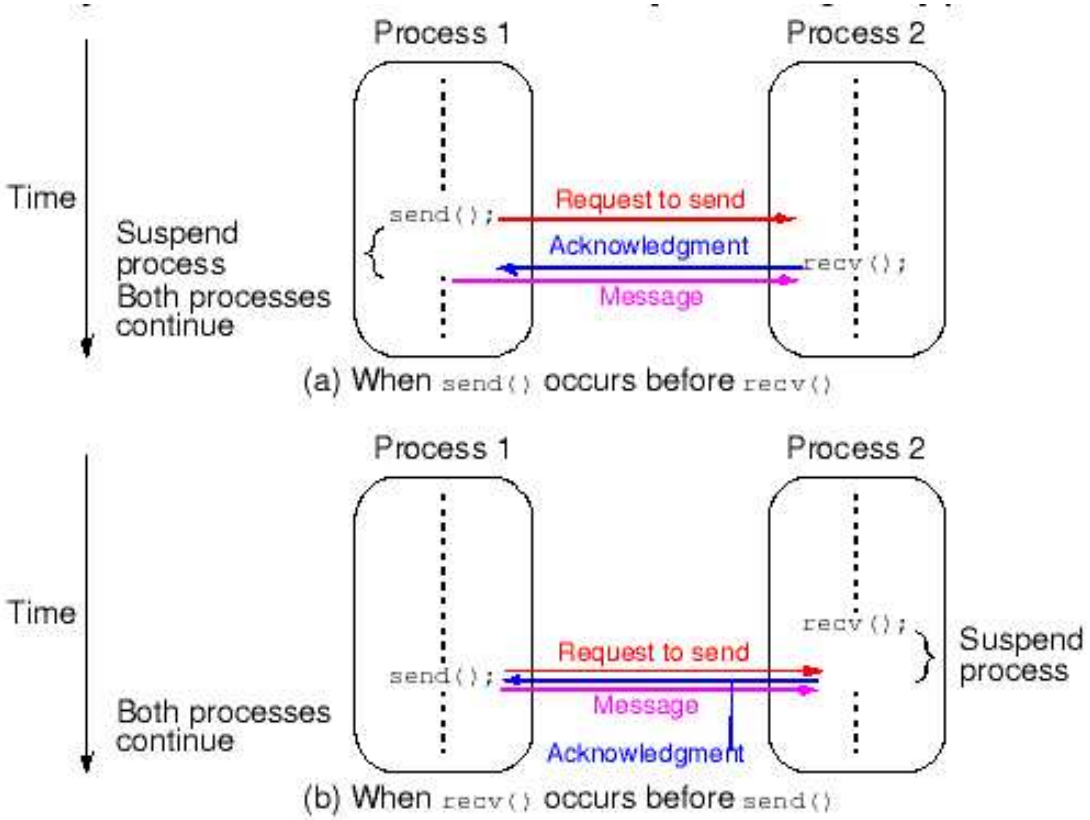
I processi coinvolti devono essere sincronizzati nel tempo e nello spazio.

- Sincronizzazione nel tempo: entrambi i processi devono esseri pronti prima che avvenga la trasmissione.
- Sincronizzazione nello spazio: richiede la disponibilità di un canale di trasmissione tra la chi invia e chi riceve.

Ad esempio il telefono.

Il processo aspetta che l'operazione sia terminata:
aspetta che il messaggio è arrivato a destinazione.

Send e *Receive* Sincroni: due possibilità



Punto a punto Asincrono

I Processi coinvolti non sono sincronizzati nel tempo o nello spazio.

- I messaggi sono "bufferizzati" fino a che l'operazione non viene eseguita.

Ad esempio l' e-mail.

Il processo non aspetta che l'operazione sia terminata:
spedito il messaggio passa all'azione successiva.

Devono essere usate con cura.

Punto a punto *Blocking* e *Non–Blocking*

Come si comporta il processo che esegue l'operazione?

- *Blocking*
  - Invio: si blocca fino a che il buffer dei dati trasmessi non è stato svuotato.
  - Ricezione: si blocca fino a che il messaggio non è arrivato.

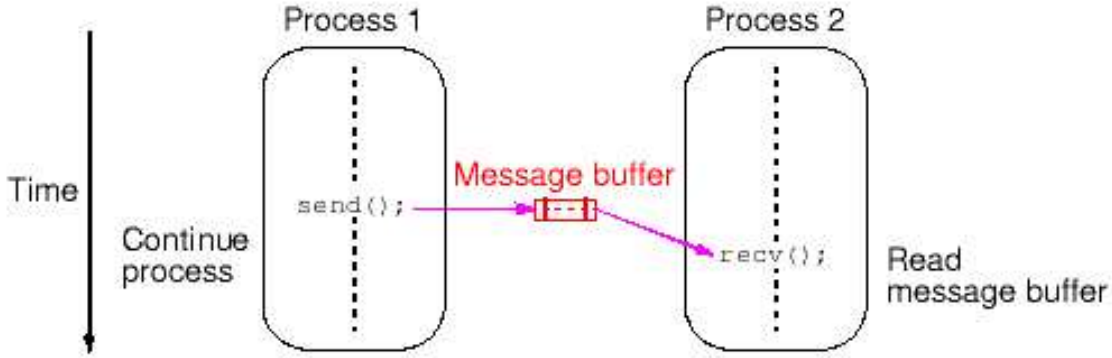
- *Non–Blocking*
  - Invio: inizia ma non la completa.
  - Ricezione: inizia ma non la completa.

  Occorre verificare il completamento delle operazioni.
  Si usa per sovrapporre operazioni di comunicazione a quelle di calcolo.

Blocking **e** *Non–Blocking*

Come è possibile che routines di *message–passing* possano passare all'azione successiva senza che il trasferimento del messaggio sia completato?

- È necessario un Message Buffer tra il processo mittente e quello ricevente:



MPI

Message Passing Interface

Cos'è MPI?

- Una libreria (non un linguaggio di programmazione).
- Un insieme di funzioni (per Fortran [77,90],C,C++) che implementano uno standard per il *message–passing*, in particolare:
  - un modello astratto per il *message–passing*
  - non un particolare compilatore
  - non un prodotto specifico.
- Paradigma di programmazione consolidato.
- Chiaro.
- Interfacciato in modo efficiente con qualunque *hardware* e svariate applicazioni.
- Sistemi forniti dai venditori di *hardware* non sono *portabili*.

Principali caratteristiche

- Modularità.
- Accesso a prestazioni di picco (dei processori).
- Portabilità.
- Eterogeneità dei dati.
- Topologie e sottogruppi di processori.
- *Tools* di misurazione delle prestazioni.

A cosa e a chi serve MPI?

- A consentire lo sviluppo di librerie parallele.
- A fornire un accesso ad *hardware* parallelo sempre più avanzato per:
  - l'utente finale
  - lo scrittore di librerie di calcolo parallele
  - lo scrittore di *software* parallelo

Un po' di storia di MPI

- Supercomputing 92 (Novembre): definizione dello *standard*.
- Supercomputing 93: una prima bozza dello *standard*.
- Due mesi di commenti aperti al pubblico.
- Maggio 94: pronta la versione finale.
- Attualmente: disponibile in versione *public-domain* e *vendor*

Chi ha definito lo *standard* MPI?

Un'ampia partecipazione

- Venditori di *hardware*
  - IBM, Intel, Meiko, Cray, Convex,...
- Scrittori di librerie parallele
  - PVM, Express, Linda,...
- Specialisti in campo applicativo e consulenti di supercalcolo

Dove si può usare MPI?

- Una varietà di macchine connesse con una rete di comunicazione:
  - parallele
  - sequenziali
  - collezioni di macchine eterogenee.

MPI o no?

- Quando usare MPI?
  - se si vuole scrivere un codice parallelo *portabile*
  - se si vuole scrivere una libreria parallela
  - se si ha a che fare con relazioni tra i dati irregolari o dinamiche che non rientrano in un modello di tipo *data parallel* (HPF)
- Quando non usare MPI?
  - se si può usare HPF o OpenMP
  - se non si ha bisogno del calcolo parallelo
  - se si possono utilizzare librerie di calcolo parallelo (magari scritte in MPI)

Cosa contiene MPI?

- Il concetto di *comunicatore*: definisce un gruppo di processi che è abilitato alla comunicazione.
- Il concetto di *thread safety* su architetture a memoria condivisa: il valore di una variabile rimane consistente anche quando vi è accesso concorrente da parte di più *threads*.
- Comunicazioni *punto–a–punto*(*blocking*, *non-blocking*, *sincrone*,...).
- Comunicazioni collettive.
- Topologie dei processori (ad esempio una maglia di processori 2D).

Cosa non contiene MPI?

- Operazioni esplicite *shared memory*.
- Gestione dei processi.
- Trasferimento remoto della memoria.
- Supporto esplicito per i *threads*.

MPI Fortran (C)

- Tutto quello che riguarda MPI ha un prefisso *MPI_*.
- Tutte le routine MPI hanno come ultimo argomento un codice (intero), se viene eseguita correttamente il suo valore è MPI_SUCCESS.
- Tutte le costanti MPI e le dichiarazioni di tipo sono contenute in mpif.h (mpi.h) che deve sempre essere incluso nel programma e nelle routine che contengono istruzioni MPI.
- Dove possibile MPI segue l'ANSI standard.

MPI com'è ?

Grande

Lo standard contiene circa 125 funzioni, ma ...

Piccolo

... non di tutte le loro funzionalità si ha realmente bisogno;
un codice completo con message passing lo posso avere usando solamente 6 funzioni:

MPI_INIT
MPI_COMM_SIZE
MPI_COMM_RANK
MPI_SEND
MPI_RECV
MPI_FINALIZE.

MPI: i gruppi

- Un gruppo è un insieme ordinato di processi.
- I processi paralleli MPI sono organizzati in gruppi.
- Ogni processo è associato ad un intero (*rank*), unico all'interno del gruppo.
- I *rank* sono contigui e compresi nel range da 0 a N-1, dove N è la taglia del gruppo.
- Il gruppo (iniziale) di default include tutti i processi paralleli.

MPI: i comunicatori

- Un *comunicatore* è un oggetto che rappresenta unicamente un dominio di comunicazione.
- Esistono due tipi:
  - Intracomunicatore: per comunicazioni all'interno di un gruppo.
  - Intercomunicatore: per comunicazioni tra due gruppi disgiunti.
- Le applicazioni MPI iniziano con un (intra)comunicatore di default MPI_COMM_WORLD che include tutti i processi.

MPI: domande preliminari

Come posso iniziare ad usare MPI?

Come posso sapere quanti processori sto usando ?

Come posso sapere quale processo sono?

Come posso concludere l'uso di MPI?

Come posso iniziare ad usare MPI?

CALL MPI_INIT(IERROR)

- Tutti i programmi MPI devono contenere una sua chiamata.
- Deve essere chiamata prima di qualsiasi altra routine MPI.
- Deve essere chiamata una sola volta; chiamate seguenti sono errori.

Quanti processori sto usando?

CALL MPI_COMM_SIZE(COMM,SIZE,IERROR)

COMM (input) nome del comunicatore.

SIZE (output) numero di processi nel gruppo del comunicatore COMM.

Per default MPI_COMM_WORLD fornisce il numero di processori disponibili.

Quale processore sono io?

CALL MPI_RANK (COMM,RANK,IERROR)

COMM (input) nome del comunicatore.

RANK (output) rango del processo nel gruppo del comunicatore COMM.

Viene usata insieme a MPI_COMM_SIZE.

Come posso concludere l'uso di MPI?

CALL MPI_FINALIZE(IERROR)

- Termina l'ambiente MPI.
- Dopo non posso essere più chiamate routine MPI.
- Verificare che tutte le comunicazioni pendenti siano terminate.

Compilazione ed esecuzione

```
% mpicc first.c -o first
% mpif77 firstf.f -o first
% mpif90 firstf.f -o first

% mpirun -np 2 first
```

mpirun non fa parte dello standard;

MPI standard non contiene comandi di esecuzione.

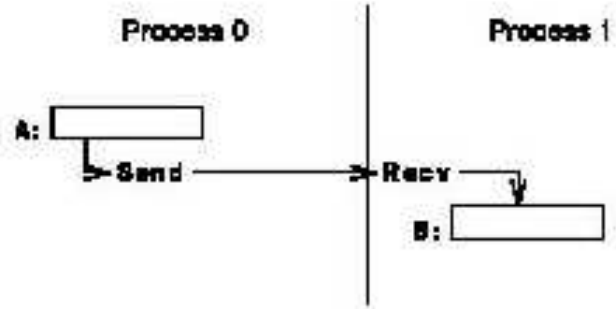
Esempio uso funzioni principali MPI

```
! Program first.f
program first
implicit none
include "mpif.h"
integer :: ierr,rank,size
call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD,size,ierr)
call MPI_Comm_rank(MPI_COMM_WORLD,rank,ierr)
write(*,*) "process", rank, "of", size
call MPI_Finalize(ierr)
end program first
```

```
process 0 of 2
process 1 of 2
```

Spedizione e ricezione in MPI

- A quale processore spedisco un dato?
- cosa spedisco?
- come fa il ricevente ad identificare il messaggio?



Spedizione(blocking) dei dati

CALL MPI_SEND(buf, count, datatype,dest, tag, comm,ierr)

buf (input) nome del dato inviato
count (input) numero di elementi del dato inviato
datatype (input) tipo di dato inviato
dest (input) processore a cui inviato il dato
tag (input) tag del messaggio inviato
comm (input) comunicatore usato per inviare il messaggio
ierr (output)

esempio

```
REAL COORD(100)
...

CALL MPI_SEND(COORD,100,MPI_REAL,2,0,MPI_COMM_WORLD,IERR)
```

Tag?

- È posto dentro il messaggio.
- È usato per distinguere tra differenti messaggi da spedire: corrispondenza univoca tra chi invia e chi riceve.
- numero intero il cui valore è compreso in un intervallo dipendente dall'architettura.
- Se non richiesta alcuna distinzione si usa come *tag* la cosiddetta wild card: il processore ricevente corrisponde a un qualunque processore che spedisce.

Quali tipi di dati posso utilizzare con MPI?

MPI datatype	Fotran datatype	Dimensioni
MPI_INTEGER,MPI_INTEGER4	INTEGER	4 byte
MPI_REAL,MPI_REAL4	REAL	4 byte
MPI_DOUBLE_PRECISION,MPI_REAL8	DOUBLE PRECISION	8 byte
MPI_COMPLEX, MPI_COMPLEX8	COMPLEX	8 byte
MPI_LOGICAL, MPI_LOGICAL4	LOGICAL	4 byte
MPI_CHARACTER	CHARACTER	1 byte

Ricezione(blocking) dei dati

CALL MPI_RECV(buf, count, datatype,source,tag, comm,status,ierr)

buf (input) nome del dato da ricevuto
count (input) numero di elementi del dato ricevuto
datatype (input) tipo di dato ricevuto
source (input) processore da cui ho ricevuto il dato
tag (input) tag del messaggio
comm (input) comunicatore usato per ricevere il messaggio
status(output) vettore di interi di dimensione MPI_STATUS_SIZE
ierr (output)

Ricezione(blocking) dei dati:considerazioni

- Il valore ricevuto di *count* deve essere uguale alle dimensioni di *buffer*:
  - se maggiore $\longrightarrow$ errore overflow
  - se minore $\longrightarrow$ modificate solo locazioni memoria in mumero uguale a *count*.
- Per effettuare l'invio devo specificare il processo ricevente, l'operazione di ricezione può essere accettata da un arbitrario processo che ha spedito.
  - qualsiasi tag è accettabile? tag=MPI_ANY_TAG
  - qualsiasi source è accettabile? source=MPI_ANY_SOURCE.
- status contiene:
  - status(MPI_TAG) tag del messagio ricevuto.
  - status(MPI_SOURCE) source del messagio ricevuto.
  - status(MPI_ERROR) codice di errore del messagio ricevuto.
- e se voglio la lunghezza del messaggio? uso la funzione MPI_GET_COUNT.

Un semplice esempio Fortran

```
program main
implicit none
include 'mpif.h'
integer rank, size, to, from, tag, countin,countout,, i, ierr
integer src, dest integer st_source, st_tag, st_count
integer status(MPI_STATUS_SIZE),st_source,dest
real     data(100)

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )

print *, 'Process ', rank, ' of ', size, ' is alive'
dest = size - 1
src = 0
if (rank .eq. src) then
  to = dest
  countin = 6
  tag = 2001
 do  i=1, 6
     data(i) = i
 end do
```

```
      call MPI_SEND( data, countin, MPI_REAL, to,
     +                tag, MPI_COMM_WORLD, ierr )

    else if (rank .eq. dest) then
         tag = MPI_ANY_TAG
         countout = 6
         from = MPI_ANY_SOURCE

      call MPI_RECV(data, countout, MPI_REAL, from,
     +              tag, MPI_COMM_WORLD, status, ierr )
      call MPI_GET_COUNT( status, MPI_REAL,
     +                    st_count, ierr )

        st_source = status(MPI_SOURCE)
        st_tag    = status(MPI_TAG)
        print *, 'Status info: source = ', st_source,
     +           ' tag = ', st_tag, ' count = ', st_count
          print *, rank, ' received', (data(i),i=1,6)
    endif

    call MPI_FINALIZE( ierr )

    end
```

Un semplice esempio Fortran:OUTPUT

countin=6, countout=6

Process 0 of 2 is alive

Process 1 of 2 is alive

Status info: source = 0 tag = 2001 count = 6

1 received 1.000000000 2.000000000 3.000000000 4.000000000 5.000000000 6.000000000

countin=5, countout=6

Process 0 of 2 is alive

Process 1 of 2 is alive

Status info: source = 0 tag = 2001 count = 5

1 received 1.000000000 2.000000000 3.000000000 4.000000000 5.000000000 0.0000000000E+00

countin=10, countout=6

Process 1 of 2 is alive

Process 0 of 2 is alive

ERROR: 0032-117 User pack or receive buffer is too small (24) in MPI_Recv, task 1 ...

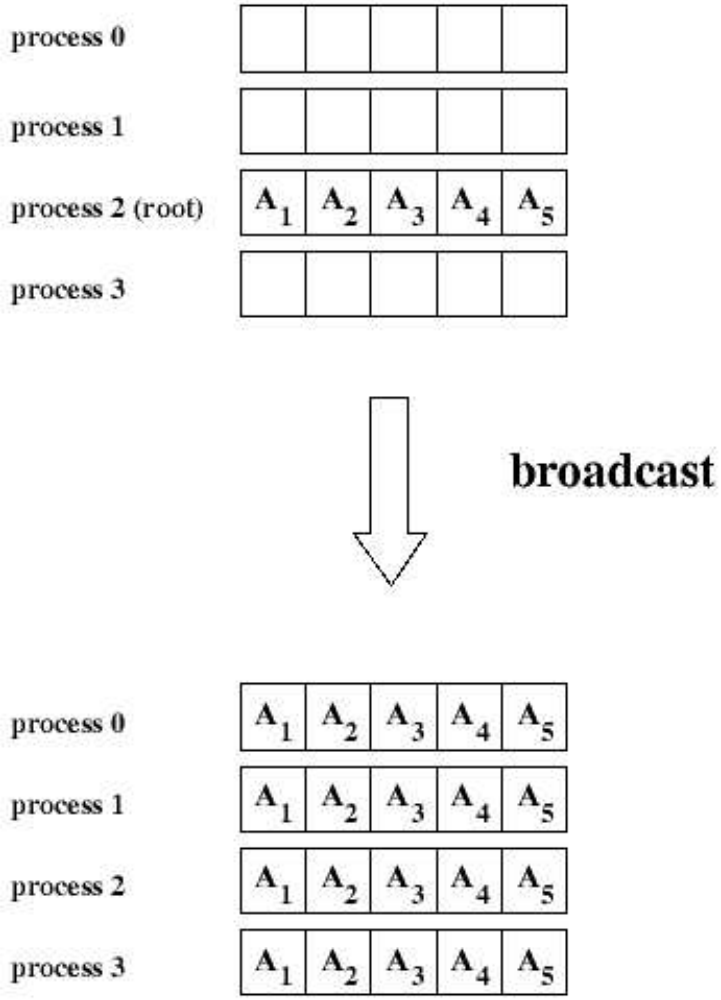
Comunicazioni collettive

- Al momento esistono solo in versione *blocking*.
- Vengono eseguite in modo che tutti i processi nel gruppo chiamano la routine di comunicazione con argomenti coincidenti.
- MPI garantisce che tutti i messaggi ad esse collegati non vengono confusi con quelli delle comunicazione *punto a punto*.
- Hanno tutte come argomento un comunicatore che identifica il gruppo di processi partecipanti.
- Non hanno la *tag* come argomento del messaggio.

Comunicazioni Collettive:funzioni principali

Funzione	Operazione
MPI_BCAST	Un processore spedisce dati ad altri processi appartenenti ad un gruppo
MPI_SCATTER	Il processore *root* comunica i dati a tutti i processi appartenti ad un gruppo
MPI_GATHER MPI_ALLGATHER	Vengono comunicati ad uno o a tutti i processi appartenenti ad un gruppo i dati posseduti da diversi processi
MPI_REDUCE MPI_ALLREDUCE	Il risultato di una operazione eseguita sui dati posseduti da diversi processi viene comunicata ad uno o a tutti i processi appartenti ad un gruppo
MPI_BARRIER	L'esecuzione di ogni processore appartenente ad un gruppo viene messa in pausa fino a che tutti i processi di quel gruppo non sono giunti a questa istruzione

Comunicazioni Collettive disponibili



MPI_BCAST

MPI_BCAST (Buffer, Count, Datatype, Root, Comm, Ierror)

Buffer (input) nome del dato spedito

Count (input) numero di elementi del dato spedito

Datatype (input) tipo di dato spedito

Root (input) rank del processore che esegue il broadcast

Comm (input) nome del comunicatore usato per l'operazione

Ierror(input)

Deve essere chiamata da tutti i membri di un gruppo usando lo stesso argomento per Comm e Root.

Esempio MPI_BCAST

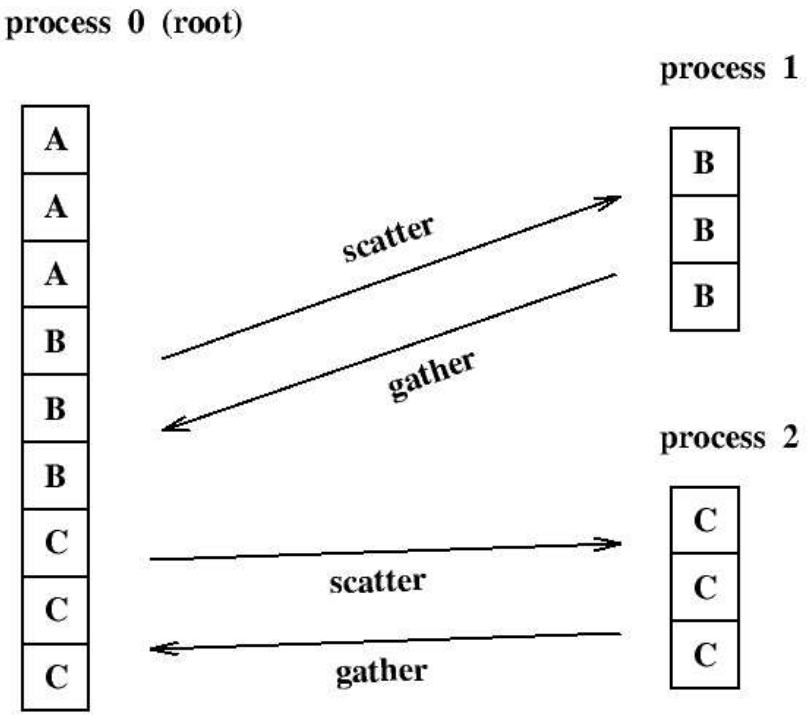
```
program esempioBCAST
include 'mpif.h'
integer nprocs, mype, ierr, i ,imesg(3)

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, mype, ierr)

  if(mype.eq.0) then
      do i=1,3
        imesg(i) = i
      enddo
  else
      do i=1,3
        imesg(i) = 0
      enddo
  endif
 write(6,*) mype,' prima:',imesg(1), imesg(2), imesg(3)
```

```
      call MPI_BCAST(imesg,3,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)

      write(6,*) mype,' dopo:',imesg(1), imesg(2), imesg(3)

      call MPI_FINALIZE(ierr)

      end
```

2proc:

```
0  prima: 1 2 3
0  dopo: 1 2 3
1  prima: 0 0 0
1  dopo: 1 2 3
```

Comunicazioni Collettive disponibili



MPI_GATHER

MPI_GATHER (Sendbuf,Sendcount,Sendtype,Recvbuf,Recvcount,Rectype,Root,Comm,Ierr)

Sendbuf (input) punto di partenza del dato inviato

Sendcount(input) numero di elementi del dato inviato

Sendtype (input) tipo del dato inviato

Recvbuf (output) indirizzo del dato ricevuto (significativo solo per Root)

Recvcount (intero) numeri di elementi del dato ricevuto da ogni singolo processore

(significativo solo per root)

Rectype (intero) tipo del dato ricevuto

Root (intero) rank del processore ricevente

Comm (intero) nome del comunicatore usato

Recvbuf e Recvcount vengono ignorati dai processori che non sono root

Root e Comm devono avere valori identici in tutti i processori

Recvcount indica il numero di elementi che il processore Root ha ricevuto da ogni processore, non il totale di elementi ricevuti.

Esempio MPI_GATHER

```
 program esempioGATHER
 include 'mpif.h'
 integer nprocs, mype, ierr, isend, i
 integer irecv
 dimension irecv(3)

   call MPI_INIT(ierr)
   call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
   call MPI_COMM_RANK(MPI_COMM_WORLD, mype, ierr)

 if(nprocs.gt.3) stop 'N. proc.>3'
    do i=1,3
    irecv(i) = 0
    enddo
    isend = mype + 1

   call MPI_GATHER(isend,1,MPI_INTEGER,
+                  irecv,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
```

```
      if(mype.eq.0) then
         write(6,*) 'irecv: ', irecv(1), irecv(2), irecv(3)
      endif

        call MPI_FINALIZE(ierr)

      end
```

2proc

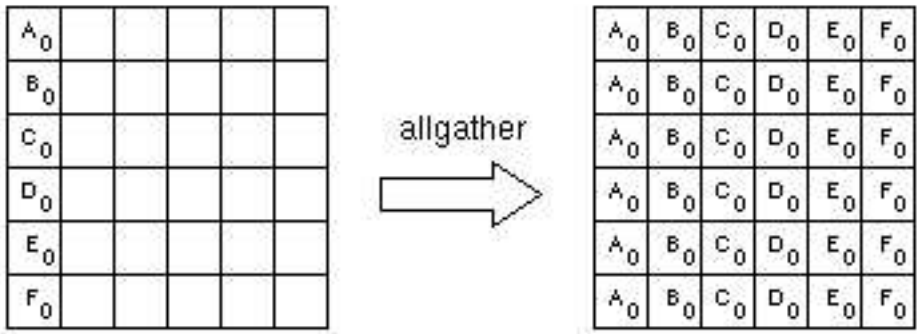
irecv: 1 2 0

3proc

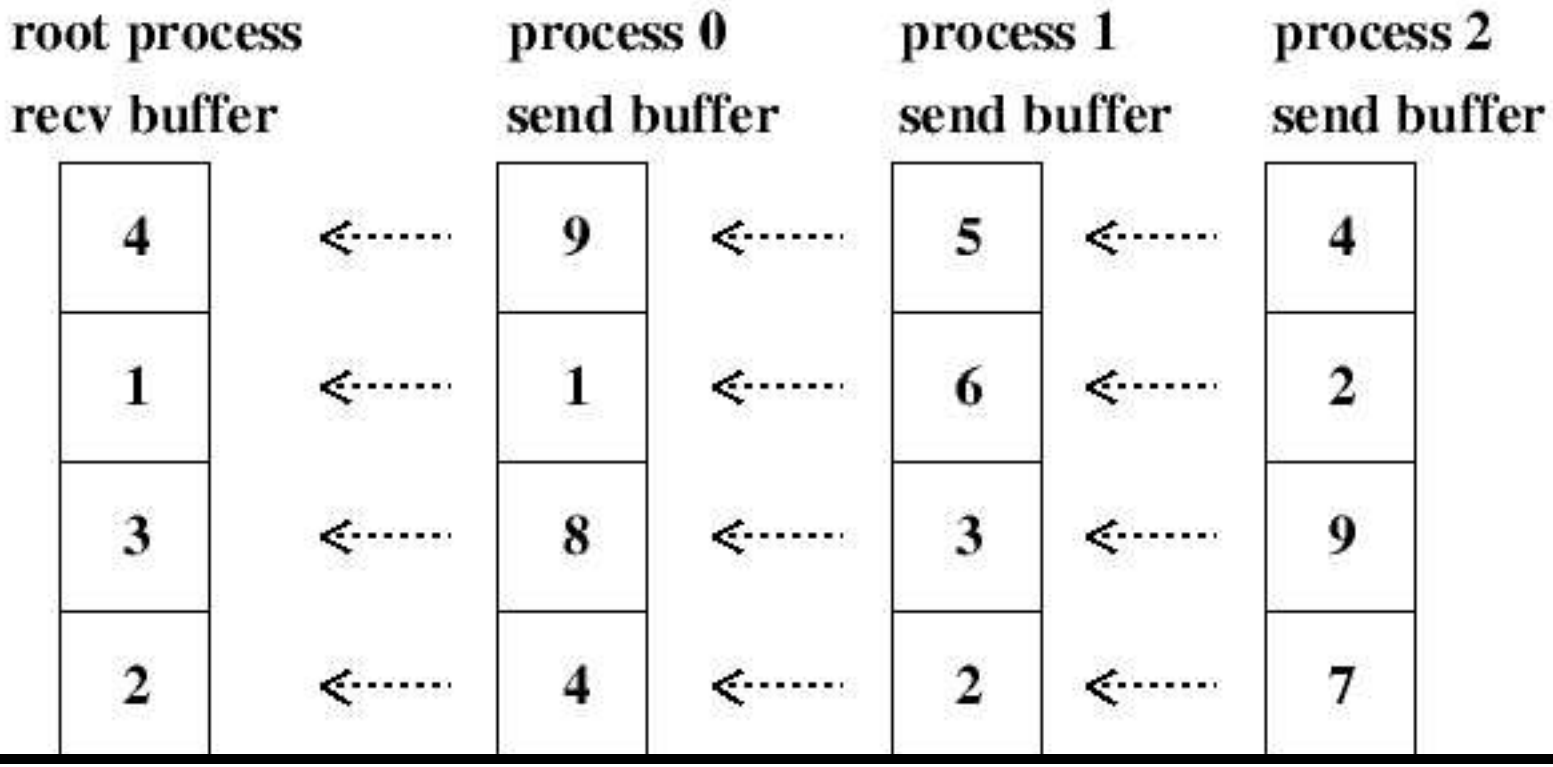
irecv: 1 2 3

Vengono inseriti in Recvbuf secondo l'ordine del rank.

Comunicazioni Collettive disponibili



an example of MPI_REDUCE using MPI_MIN



Esempio MPI_REDUCE

MPI_REDUCE (Sendbuf,Recvbuf,Count,Datatype,Op,Root,Comm,Ierr)
Sendbuf (input) nome del dato inviato
Recvbuf (input) nome che assume il dato ricevuto
Count (input) numero di elementi del dato inviato e ricevuto
Datatype (input) tipo del dato inviato e ricevuto
Op (input) operazione di riduzione effettuata sui dati ricevuti
Root (input) rank del processore che esegue la riduzione
Comm (input) comunicatore usato

Sendbuf e Recvbuf hanno lo stesso numero di elementi dello stesso tipo

Esempio MPI_REDUCE

```
      program esempioREDUCE
      include 'mpif.h'
      integer nprocs, mype, ierr, i
      integer isend, irecv
      dimension isend(2), irecv(2)

         call MPI_INIT(ierr)
         call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
         call MPI_COMM_RANK(MPI_COMM_WORLD, mype, ierr)

        do i=1,2
          isend(i) = mype + i
        enddo

         call MPI_REDUCE(isend(1),irecv(1),1,MPI_INTEGER,
     +                   MPI_SUM,0,MPI_COMM_WORLD,ierr)
         call MPI_REDUCE(isend(2),irecv(2),1,MPI_INTEGER,
     +                   MPI_PROD,0,MPI_COMM_WORLD,ierr)
```

```
      if(mype.eq.0) then
        write(6,*) 'irecv: ', irecv(1), irecv(2)
      endif

    call MPI_FINALIZE(ierr)
  end
```

2 processori

irecv: 3 6

4 processori

irecv: 10 120

Altre routines collettive in MPI

Esistono altre *routines*:

- Tutte le *routines* il cui nome inizia per All spediscono il risultato dell'operazione a tutti i processi coinvolti nella comunicazione.
- Tutte le *routines* il cui nome termina con v consentono di avere *chunks* dei dati di taglia diversa (l'argomento della chiamata è un vettore non uno scalare).
- Allreduce, Reduce, ReduceScatter, Scan prendono in *input* sui dati sia operazioni predefinite che operazioni definite dall'utente.

Altre routines collettive in MPI

MPI_SCATTERV(sendbuf,sendcounts,displs,sendtype,recvbuf,recvcount,recvtype,root,comm,ierror)

sendbuf(input) nome del dato inviato

sendcounts(input) vettore di interi di lunghezza della taglia del gruppo che contiene il numero di elementi da inviare ad ogni processore

displs (input) vettore di interi di lunghezza della taglia del gruppo contiene i displacement nell'array sendbuf da cui partono i dati da inviare ad ogni processo

sendtype (input) il tipo del dato inviato

recvbuf (out) nome del dato ricevuto

recvcount (input) numero di elementi del dato ricevuto

recvtype(input) tipo del dato ricevuto

root(input) proessore che esegue l'operazione

comm (input) comunicatore usato per la spedizione

ierror(input)

Esempio MPI_SCATTERV

```
program scatterv
implicit none
include 'mpif.h'
integer isend(6),irecv(3),ircnt,ierr
integer iscnt(0:2),idisp(0:2),nprocs,myrank
        data isend/1,2,2,3,3,3/
        data iscnt/1,2,3/ idisp/0,1,3/

CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)

ircnt=myrank+1

CALL MPI_SCATTERV(isend,iscnt,idisp,MPI_INTEGER,irecv,ircnt,MPI_INTEGER,
0,MPI_COMM_WORLD,ierr)
```

```
print*,rank,'irecv',irecv
CALL MPI_FINALIZE(ierr)
end
```

3 proc

1 irecv 2 2 0

2 irecv 3 3 3

0 irecv 1 0 0

Operazioni Collettive non predefinite in MPI

MPI_Op_create(user_function, commute, op)

user_function (input) funzione di riduzione definita dall'utente

commute (input) true se è commutativa altrimenti definita false

op (output) operazione di riduzione

MPI_Op_free(op) usata per cancellare l'operazione di riduzione op definita

Un esempio

```
 program exampleOp
 include 'mpif.h'
 integer ierr, rank, i, n
 parameter (n = 1000)
 real a(n), b(n)
 integer op
 external smod5

   call MPI_INIT(ierr)
   call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)

 do i = 1, n
    a(i) = i + rank
 end do
 print *, 'process ', rank, ' a(1) =', a(1)

   call MPI_OP_CREATE(smod5, .TRUE., op, ierr)
   call MPI_REDUCE(a, b, n, MPI_REAL, op, 0,
&                  MPI_COMM_WORLD, ierr)
   call MPI_OP_FREE(op, ierr)

 if(rank .eq. 0) write(*,*) ' b(1) =', b(1)

   call MPI_FINALIZE(ierr)
```

```
      end
```

```
      subroutine smod5(in, inout, l, type)
      integer l, type,i
      real in(l), inout(l)

      do i = 1, l
        inout(i) = in(i)+inout(i)
      end do
      return
      end
```

```
process 0 a(1) = 1.000000000

process 1 a(1) = 2.000000000

b(1) = 3.000000000
```

Esempio prodotto Matrice vettore

Condizioni(per semplicità):

- Matrice suddivisa in blocchi di righe.
- Matrice quadrata.
- Numero delle righe divisibile per il numero dei processori.

Prodotto matrice vettore:punto a punto

```
program pmv
implicit none
include 'mpif.h'
integer, dimension(MPI_STATUS_SIZE) :: STATUS
integer :: IE,RANGO,NP,N,BS,I
real, dimension(:,:), allocatable         :: A
real, dimension(:), allocatable          :: X,Y

call MPI_INIT(IE)
call MPI_COMM_RANK(MPI_COMM_WORLD,RANGO,IE)
call MPI_COMM_SIZE(MPI_COMM_WORLD,NP,IE)

 if(RANGO.eq.0) then
   write(*,*) 'dai l'ordine della matrice A'
   read(*,*) N
   if(NP*(N/NP).ne.N) goto 10
   allocate(A(N,N),X(N),Y(N))
   call random_number(A)
   call random_number(X)
   A=int(100*A)
   X=int(100*X)
   write(*,*) (DOT_PRODUCT(A(I,1:N),X),I=1,N)
   BS=N/NP
```

```
   do I=1,NP-1

   CALL MPI_SEND(N,1,MPI_INTEGER,I,10,MPI_COMM_WORLD,IE)

   CALL MPI_SEND(A(BS*I+1:BS*(I+1),1:N),BS*N,MPI_REAL,I,I*50,MPI_COMM_WORLD,IE)

   CALL MPI_SEND(X,N,MPI_REAL,I,I*60,MPI_COMM_WORLD,IE)

   END DO

   do I=1,BS
   Y(I)=DOT_PRODUCT(A(I,1:N),X)
   end do

   do I=1,NP-1

   call MPI_RECV(Y(BS*I+1:BS*(I+1)),BS,MPI_REAL,I,20,MPI_COMM_WORLD,STATUS,IE)

   end do

   write(*,*) Y
else

   call MPI_RECV(N,1,MPI_INTEGER,0,10,MPI_COMM_WORLD,STATUS,IE)
```

```
            BS=N/NP
            allocate(A(BS,1:N),X(N),Y(BS))

            call MPI_RECV(A,BS*N,MPI_REAL,0,RANGO*50,MPI_COMM_WORLD,STATUS,IE)

            call MPI_RECV(X,N,MPI_REAL,0,RANGO*60,MPI_COMM_WORLD,STATUS,IE)

            do I=1,BS
            Y(I)=DOT_PRODUCT(A(I,1:N),X)
            end do

            call MPI_SEND(Y,BS,MPI_REAL,0,20,MPI_COMM_WORLD,IE)

          endif

10          call MPI_FINALIZE(IE)

          end
```

```
dai l'ordine della matrice A
4
3044.000000 4423.000000 6499.000000 5265.000000
3044.000000 4423.000000 6499.000000 5265.000000
```

Prodotto matrice vettore:collettive

```
program pmv
implicit none
include 'mpif.h'
integer, dimension(MPI_STATUS_SIZE) :: STATUS
integer :: IE,RANGO,NP,N,BS,I,J
real, dimension(:,:), allocatable         :: A,B
real, dimension(:), allocatable          :: X,Y,Z
real :: c

call MPI_INIT(IE)
call MPI_COMM_RANK(MPI_COMM_WORLD,RANGO,IE)
call MPI_COMM_SIZE(MPI_COMM_WORLD,NP,IE)

if(RANGO.eq.0) then
  write(*,*) 'dai l''ordine della matrice A'
  read(*,*) N
if(NP*(N/NP).ne.N) goto 10
  BS=N/NP
  allocate(A(N,N),X(N),Y(N),B(N,BS),Z(BS))
  call random_number(A)
  call random_number(X)
  A=int(100*A)
  X=int(100*X)
```

```
  write(*,*) (DOT_PRODUCT(A(I,1:N),X),I=1,N)
! Viene trasposta la matrice A
   do I=1,N
   do J=1,I-1
   C=A(I,J)
   A(I,J)=A(J,I)
   A(J,I)=C
   end do
   end do
       endif

call MPI_BCAST(BS,1,MPI_INTEGER,0,MPI_COMM_WORLD,IE)

if(RANGO.ne.0) then
N=BS*NP
allocate(B(N,BS),X(N),Z(BS))
endif

call MPI_BCAST(X,N,MPI_REAL,0,MPI_COMM_WORLD,IE)
call MPI_SCATTER(A,BS*N,MPI_REAL,B,BS*N,MPI_REAL,0,MPI_COMM_WORLD,IE)

do i=1,BS
Z(I)=DOT_PRODUCT(B(1:N,I),X)
end do
```

```
call MPI_GATHER(Z,BS,MPI_REAL,Y,BS,MPI_REAL,0,MPI_COMM_WORLD,IE)

if(RANGO.eq.0) then
Write(*,*) Y(1:N)
endif

10          call MPI_FINALIZE(IE)

end
```

Comunicazione punto a punto: dettaglio

Quando un singolo messaggio viene spedito dal processo 0 al processo 1 questi passaggi avvengono in modo temporalmente ordinato.

- Il dato è copiato in una variabile dal processo 0.
- Il processo 0 chiama una delle funzioni di send dell'MPI.
- Il sistema operativo copia il dato dalla variabile ad una zona di memoria apposita che chiameremo variabile di sistema.
- Il sistema operativo spedisce il dato presente nella variabile di sistema al processo 1.

Dal punto vista del ricevente:

- il processo 1 chiama una delle funzioni di ricezione MPI.
- Il sistema operativo riceve il dato dal processore 0 e lo copia in una variabile di sistema.
- Il sistema operativo copia il dato dalla variabile di sistema alla variabile del processo 1.
- Il processo 1 usa il dato presente nella variabile.

Comunicazioni punto a punto blocking

Esistono quattro modi diversi per il send:

- MPI_SSEND Send sincrono: si completa quando la ricezione è cominciata
- MPI_BSEND Send buffered: si completa sempre (a meno di errori) senza considerare se la ricezione é stata completata
- MPI_SEND Send standard: sincrono o buffered
- MPI_RSEND Send ready: si completa sempre (a meno di errori) senza considerare se la ricezione è stata completata
- MPI_RECV Recv: si completa quando la il messaggio è arrivato

Questi modi esistono anche nella forma no blocking.

Comunicazioni punto a punto no blocking

- MPI_ISSEND Send sincrono
- MPI_IBSEND Send buffered
- MPI_ISEND Send standard
- MPI_IRSEND Send ready
- MPI_IRECV Recv

Gli argomenti sono gli stessi con al posto della variabile status la variabile request, importante per testare se la comunicazione è stata completata.

Ad esempio:

CALL MPI_RECV(array, count, datatype,source,tag, comm,status,ierr)

CALL MPI_IRECV(array, count, datatype,source,tag, comm,request,ierr)

Comunicazioni *Blocking* e *Non–Blocking*

In precedenza abbiamo mostrato comunicazioni MPI di tipo *Blocking*:

- MPI_Send non ritorna all'esecuzione finchè il *buffer* dei dati non è stato svuotato (disponibile per essere riutilizzato).
- MPI_Recv non ritorna all'esecuzione finchè il *buffer* dei dati è pieno (disponibile per essere utilizzato).

Tutto molto semplice ma può essere *non–sicuro*!!

Comunicazioni *Blocking* e *Non–Blocking*

```
program dl
include 'mpif.h'

parameter (n=100)

integer nprocs, mype, ierr, itag, itag2, imesg, imesg2
integer istatus, i
dimension imesg(n), imesg2(n)
dimension istatus(MPI_STATUS_SIZE)

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, mype, ierr)

itag = 1
itag2= 2
do i=1,n
imesg(i) = mype
imesg2(i)= mype+1
enddo
```

```
 if(mype.eq.1) then

 call MPI_SEND(imesg,n,MPI_INTEGER,0,itag,
+              MPI_COMM_WORLD,ierr)

 call MPI_RECV(imesg2,n,MPI_INTEGER,0,itag2,
+              MPI_COMM_WORLD,istatus,ierr)

 write(6,*) 'messaggio = ', imesg2(1)
 elseif(mype.eq.0) then

 call MPI_SEND(imesg2,n,MPI_INTEGER,1,itag2,
+              MPI_COMM_WORLD,ierr)

 call MPI_RECV(imesg,n,MPI_INTEGER,1,itag,
+              MPI_COMM_WORLD,istatus,ierr)

 write(6,*) 'messaggio = ', imesg(1)
 endif

 call MPI_FINALIZE(ierr)

 end
```

Comunicazioni *Blocking* e *Non–Blocking*

Lo eseguo su due processori di un nodo PWR5 del Caspur:

- con n$\leq$8100
  messaggio = 1
  messaggio = 1

- con n>8100
  ??????????

Deadlock

MPI può usare indifferentemente la modalità *Buffered* o *Sincrona*:
Il completamento o no dell'operazione dipenderà dalla natura del messaggio e dall'architettura che si sta considerando.

Lo *standard* MPI comunque afferma che:
send funziona sempre per messaggi di taglia abbastanza piccola ma può fallire quando la taglia è troppo grande (utilizza un *send* sincrono).

Soluzioni al *deadlock*

Si puó invertire l'ordine delle chamate:

Processore 1

```
call MPI_SEND(imesg2,...)

call MPI_RECV(imesg,...)
```

Processore 0

```
call MPI_RECV(imesg2,...)

call MPI_SEND(imesg,...)
```

Oppure usare per entrambi i processori la chiamata a

MPI_SENDRECV

Soluzioni al deadlock: *MPI_SENDRECV*

call MPI_SENDRECV(sendbuf,sendcount,sendtype,dest,sendtag,recvbuf,recvcount

recvtype,source,recvtag,comm,status)

sendbuf,recvbuf: buffer di invio e di ricezione

sendcount,recvcount: numero di elementi inviati e ricevuti (possono essere diversi)

sendtype,recvtype: tipo di dati inviati e ricevuti (possono essere diversi)

sendtag,recvtag: tag di invio e di ricezione

dest: rank del processore a cui invio

source: rank del processore da cui ricevo

comm: comunicatore

status: vettore che mi indica lo stato dell'operazione

Soluzioni al deadlock

```
if(mype.eq.1) then
   call MPI_ISEND(imesg,...,ireq,...)
   call MPI_RECV(imesg2,...)
   call MPI_WAIT(ireq,...)
elsif(mype.eq.0) then
   call MPI_ISEND(imesg2,...,ireq,...)
   call MPI_RECV(imesg)
   call MPI_WAIT(ireq)
endif
```

Soluzioni al deadlock

MPI_WAIT(request,status,ierror)

- blocca l'operazione identificata da request finché non è completata
- status(MPI_STATUS_SIZE) contiene informazioni sull'operazione completata

Può bastare una sola chiamata ad MPI_ISEND?

Soluzioni al deadlock

```
program dl1
include 'mpif.h'
parameter (n=40000)
integer nprocs, mype, ierr, itag, itag2, imesg, imesg2
integer istatus, i
dimension imesg(n), imesg2(n)
dimension dstatus(MPI_STATUS_SIZE)


call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, mype, ierr)

itag = 1
itag2= 2
richiesta=1
do i=1,n
imesg(i) = mype
imesg2(i)= mype+1
enddo
```

```
 if(mype.eq.1) then
 call MPI_ISEND(imesg,n,MPI_INTEGER,0,itag,
+               MPI_COMM_WORLD,richiesta,ierr)
 call MPI_RECV(imesg2,n,MPI_INTEGER,0,itag2,
+              MPI_COMM_WORLD,istatus,ierr)
 call MPI_WAIT(richiesta,status,ierr)
 write(6,*) 'messaggio = ', imesg2(1)
 elseif(mype.eq.0) then
 call MPI_SEND(imesg2,n,MPI_INTEGER,1,itag2,
+              MPI_COMM_WORLD,ierr)
 call MPI_RECV(imesg,n,MPI_INTEGER,1,itag,
+              MPI_COMM_WORLD,istatus,ierr)
 write(6,*) 'messaggio = ', imesg(1)
 endif
 call MPI_FINALIZE(ierr)
 end
```

Soluzioni al deadlock

Si può usare MPI_BSEND

utilizza un *buffer* aggiuntivo al *buffer* di sistema.

Il problema è definire opportunamente la dimensione del *buffer* con

MPI_Buffer_attach (buffer,size)

size deve essere espressa in data bytes.

Equazione di Laplace: seriale

```
program laplace
implicit none
real t(0:1001),tnew(1000),tol
real dif,difmax
integer i,n

print*, 'valori di n e di tol',n,tol
read*,n,tol

do i=0,n
 t(i)=0.0
end do
t(n+1)=100.0

10   continue

do i=1,n
  tnew(i)=(t(i-1)+t(i+1))/2.
end do

difmax=0.0
do i=1,n
  dif= abs(tnew(i)- t(i))
```

```
if(dif.gt.difmax) difmax=dif
t(i)=tnew(i)
end do
if(difmax.gt.tol) goto 10

do i=1,n
  write(*,*) t(i)
end do

end
```

Equazione di Laplace: chiamate collettive

```
program laplace
implicit none
include 'mpif.h'
real t(0:1001),tnew(1000),tol
real dif,difmax,gdmax
integer i,n,my_id,low,high,master
integer left,right,ierr,nproc
integer status(MPI_STATUS_SIZE)


call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,my_id,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)

master=0
if(my_id.eq.master) then
print*, 'valori di n e di tol',n,tol
read*,n,tol
endif

call MPI_BCAST(n,1,MPI_INTEGER,MASTER,MPI_COMM_WORLD,ierr)
call MPI_BCAST(tol,1,MPI_REAL,MASTER,MPI_COMM_WORLD,ierr)
```

```
low=my_id*n/nproc+1
high=low+n/nproc-1

if(my_id.eq.0) then
left=MPI_PROC_NULL
else
left=my_id-1
endif

if(my_id.eq.nproc-1) then
 right=MPI_PROC_NULL
else
 right=my_id+1
endif
do i=low,high
 t(i)=0.0
end do
if(my_id.eq.nproc-1) t(n+1)=100.0

10   continue

call MPI_SENDRECV(t(low),1,MPI_REAL,left,1,t(high+1),1,MPI_REAL,right,1,MPI_COMM_WORLD,status,ierr)
call MPI_SENDRECV(t(high),1,MPI_REAL,right,2,t(low-1),1,MPI_REAL,left,2,MPI_COMM _WORLD,status,ierr)
```

```
do i=low,high
  tnew(i)=(t(i-1)+t(i+1))/2.
end do
difmax=0.0
do i=low,high
  dif= abs(tnew(i)- t(i))
if(dif.gt.difmax) difmax=dif
  t(i)=tnew(i)
end do

call MPI_ALLREDUCE(difmax,gdmax,1,MPI_REAL,MPI_MAX,MPI_COMM_WORLD,ierr)

if(gdmax.gt.tol) goto 10

call MPI_GATHER(t(low),n/nproc,MPI_REAL,t(1),n/nproc,MPI_REAL,master,MPI_COMM_WORLD,ierr)

if(my_id.eq.master) then
do i=1,n
  write(*,*) t(i)
end do
endif

call MPI_FINALIZE(ierr)

end
```

Equazione di Laplace: chiamate asincrone

```
program laplace
implicit none
include 'mpif.h'
real t(0:1001),tnew(1000),tol
real dif,difmax,gdmax
integer i,n,my_id,low,high,master
integer left,right,ierr,nproc
integer status(MPI_STATUS_SIZE),request(4)

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,my_id,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)

master=0
if(my_id.eq.master) then
print*, 'valori di n e di tol',n,tol
read*,n,tol
endif

call MPI_BCAST(n,1,MPI_INTEGER,MASTER,MPI_COMM_WORLD,ierr)
call MPI_BCAST(tol,1,MPI_REAL,MASTER,MPI_COMM_WORLD,ierr)

low=my_id*n/nproc+1
high=low+n/nproc-1
```

```
if(my_id.eq.0) then
left=MPI_PROC_NULL
else
left=my_id-1
endif
if(my_id.eq.nproc-1) then
 right=MPI_PROC_NULL
else
 right=my_id+1
endif
do i=low,right
 t(i)=0.0
end do
if(my_id.eq.nproc-1) t(n+1)=100.0
10   continue

call MPI_IRECV(t(high+1),1,MPI_REAL,right,1,MPI_COMM_WORLD,request(1),ierr)
call MPI_IRECV(t(low-1),1,MPI_REAL,left,2,MPI_COMM_WORLD,request(2),ierr)
call MPI_ISEND(t(low),1,MPI_REAL,left,1,MPI_COMM_WORLD,request(3),ierr)
call MPI_ISEND(t(high),1,MPI_REAL,right,2,MPI_COMM_WORLD,request(4),ierr)

do i=low+1,high-1
  tnew(i)=(t(i-1)+t(i+1))/2.
```

```
end do
   call MPI_WAITALL(4,request,status,ierr)
   tnew(low)=(t(low-1)+t(low+1))/2.
   tnew(high)=(t(high-1)+t(high+1))/2.

difmax=0.0
do i=low,high
  dif= abs(tnew(i)- t(i))
if(dif.gt.difmax) difmax=dif
  t(i)=tnew(i)
end do
call MPI_ALLREDUCE(difmax,gdmax,1,MPI_REAL,MPI_MAX,MPI_COMM_WORLD,ierr)
if(gdmax.gt.tol) goto 10
call MPI_GATHER(t(low),n/nproc,MPI_REAL,t(1),n/nproc,MPI_REAL,master,MPI_COMM_WORLD,ierr)
if(my_id.eq.master) then
do i=1,n
  write(*,*) t(i)
end do
endif
call MPI_FINALIZE(ierr)
end
```

Ulteriori routine MPI per...

- poter inviare tipi diversi di variabili nello stesso messaggio;
- poter aggregare i processi in un comunicatore in modo che 'fittino' meglio il percorso di comunicazione;
- calcolare il tempo speso nell'eseguire un codice parallelo;
- ...

Riferimenti Web:

- **MPI** http://www-unix.mcs.anl.gov/mpi/

Sulla macchina di lavoro la chiamata
man nome_della_routine_MPI
fornisce informazioni sulla routine corrispondente.

Esempio: man MPI_WAIT

```
MPI_WAIT, MPI_Wait

Purpose

Waits for a nonblocking operation to complete.

C synopsis

#include <mpi.h>
int MPI_Wait(MPI_Request *request,MPI_Status *status);

C++ synopsis

#include mpi.h
void MPI::Request::Wait();

#include mpi.h
void MPI::Request::Wait(MPI::Status& status);

FORTRAN synopsis

include 'mpif.h' or use mpi
MPI_WAIT(INTEGER REQUEST,INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
```

Description

MPI_WAIT returns after the operation identified by request completes. If the object associated with request was created by a nonblocking operation, the object is deallocated and request is set to MPI_REQUEST_NULL. MPI_WAIT is a non-local operation.

You can call MPI_WAIT with a null or inactive request argument. The operation returns immediately. The status argument returns tag = MPI_ANY_TAG, source = MPI_ANY_SOURCE. The status argument is also internally configured so that calls to MPI_GET_COUNT and MPI_GET_ELEMENTS return count = 0. This is called an empty status.
Information on the completed operation is found in status. You can query the status object for a send or receive operation with a call to MPI_TEST_CANCELLED. For receive operations, you can also retrieve information from status with MPI_GET_COUNT and MPI_GET_ELEMENTS. If wildcards were used by the receive for either the source or tag, the actual source and tag can be retrieved by:

In C:

```
source = status.MPI_SOURCE
tag = status.MPI_TAG
```

In FORTRAN:

```
source = status(MPI_SOURCE)
tag = status(MPI_TAG)
```

The error field of MPI_Status is never modified. The success or failure is indicated by the return code only.

Passing MPI_STATUS_IGNORE for the status argument causes PE MPI to skip filling in the status fields. By passing this value for status, you can avoid having to allocate a status object in programs that do not need to examine the status fields.

When one of the MPI wait or test calls returns status for a nonblocking operation request and the corresponding blocking operation does not provide a status argument, the status from this wait or test call does not contain meaningful source, tag, or message size information.

When you use this subroutine in a threads application, make sure that the wait for a given request is done on only one thread. The wait does not have to be done on the thread that created the request. See IBM Parallel Environment for AIX: MPI Programming Guide for more information on programming with MPI in a threads environment.

Parameters

request
     is the request to wait for (handle) (INOUT)
status
     is the status object (Status) (INOUT). Note that in FORTRAN a single
     status object is an array of integers.
IERROR
     is the FORTRAN return code. It is always the last argument.

Errors

A GRequest free function returned an error

A GRequest query function returned an error

Invalid status ignore value

Invalid form of status ignore

Invalid request handle

Truncation occurred

MPI not initialized

MPI already finalized

Develop mode error if:

Illegal buffer update (ISEND)

Inconsistent datatype (MPE_I collectives)

Inconsistent message length (MPE_I collectives)

Inconsistent op (MPE_I collectives)

Match of blocking and non-blocking collectives (MPE_I collectives)

Related information

- MPI_TEST
- MPI_WAITALL
- MPI_WAITANY
- MPI_WAITSOME