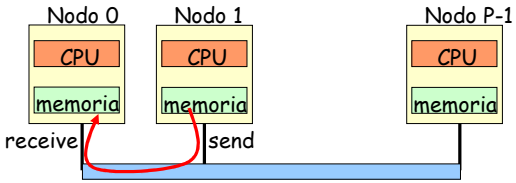
1

modello di programmazione mesage passing

Se un nodo (Nodo 0) ha bisogno di un dato in possesso di altro nodo (Nodo 1), e' necessario un esplicito trasferimento di dati

I sistemi MIMD a memoria distribuita richiedono l'estensione dei linguaggi di programmazione con opportune istruzioni per implementare le operazioni di
- Send(dato, destinazione)
- Receive(dato, sorgente)



modello a scambio di messaggi

2

Librerie per il Message Passing (1)

- Prima meta' anni '90: molte librerie per il Message Passing
  - Ambito accademico
    - PVM, Parallel Virtual Machine (ORNL/UTK)
    - Chameleon, (ANL).
    - Zipcode, (LLL).
  - Ambito Industriale
    - CMMD, (Thinking Machines)
    - MPL, (IBM SP-2).
    - NX, (Intel Paragon).
  - Ambito commerciale
    - Express,.

Necessita' di uno standard per lo sviluppo di software portabile

MPI, Message Passing Interface, (1995)

3

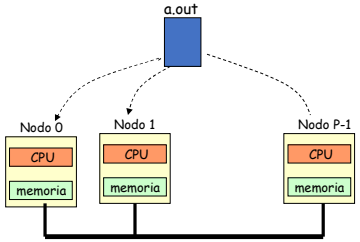
Librerie per il Message Passing (1)

- Tutte le comunicazioni e le sincronizzazioni richiedono chiamate a funzioni
  - Assenza di variabili condivise
  - Esecuzione sequenziale su un processore tranne che per le chiamate alla libreria
  - Funzioni per
    - comunicazione
      - point-to-point: Send e Receive
      - Collettive: Broadcast, Scatter/gather , sum, product, max
    - sincronizzazione
      - Barriera collettiva
      - Assenza di semafori a causa dell'assenza di dati condivisi
    - Definizione di variabili di ambiente
      - Quanti processori?
      - Chi sono?

4

Sviluppo applicazioni con MPI

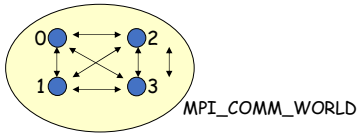
- MPI e' una libreria
  - Tutte le operazioni eseguite da chiamate a routine inserite nel codice sorgente
- Il sistema manda in esecuzione P copie dello stesso a.out (modello SPMD)
  - Ogni processo ha un suo spazio di indirizzamento
  - Non necessariamente corrispondenza 1-1 CPU/processo



5

Alcuni concetti base

- Un comunicatore e' un identificativo che caratterizza un gruppo di processi MPI che possono comunicare tra loro
- Ogni comunicatore ha:
  - Una dimensione
  - Un nome (di default e' MPI_COMM_WORLD)
- I processi sono
  - Univocamente identificato da un intero (rango)
- Solo processi appartanenti allo stesso contesto possono comunicare tra loro
- Un processo puo' fare parte di piu' comunicatori
- Ogni istruzione e' eseguita indipendentemente in ogni processo



6

Alcune osservazioni

- Tutti i programmi MPI iniziano con

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

e finiscono con

```
int   MPI_Finalize(void)
```

- Tali funzioni rispettivamente creano e distruggono l'ambiente di comunicazione. In particolare creano e ditruggono
  - il comunicatore
  - Il numero di processi ed il rango
  - le strutture dati per la comunicaizone

7

l'ambiente

- Due importanti questioni da risolvere in un programma parallelo:
  - Quanti processi partecipano al calcolo?
  - Chi sono io?
- Funzioni di MPI:
  - **int MPI_Comm_Size** (MPI_comm comm, int *size)
    - restituisce il numero di processi (size)
  - **int MPI_Comm_Rank** (MPI_comm comm, int *rank)
    - restituisce un numero (rank) tra 0 e size-1 che identifica il processo chiamante

8

send/receive

Principali funzioni di comunicazione punto/punto

Per spedire o ricevere dei dati e' necessario specificare:

- Intestazione del dato
  - Sorgente / destinazione
  - Identificativo (tag), utile per non sovrapporre messaggi
  - Comunicatore
- Descrizione del dato
  - Nome del dato
  - Dimensione del dato
  - Tipo del dato

In totale 6 argomenti

send/receive

```
MPI_Send(void *data, int count, MPI_datatype datatype,
         int dest, int tag, MPI_comm comm)
```

```
MPI_Recv(void *data, int count, MPI_datatype datatype,
         int source, int tag, MPI_comm comm,
         MPI_status *status)
```

- data e' il puntatore al dato
- count e' la dimensione (ad es per un array count >1)
- datatype e' il tipo di dati

- dest / source e' la destinazione/ sorgente del messaggio
- tag e' l'identificativo del messaggio
- comm e' il comunicatore
- status e' un flag di errore

I messaggi sono bufferizzati
(overhead per la copia nel e dal buffer locale)

6 funzioni base

Lo standard MPI definisce oltre 100 funzioni

Ma con le 6 funzioni

- MPI_Init
- MPI_Finalize
- MPI_Comm_rank
- MPI_Comm_size
- MPI_Send
- MPI_Recv

E' possibile scrivere (magari con difficolta') gran parte delle applicazioni

spedizioni bloccanti / non bloccanti

le funzioni MPI_SEND( ) e MPI_RECV( ) sono bloccanti

CIOE'

non ritornano se il messaggio non ha raggiunto il destinatario

le funzioni MPI_ISEND( ) e MPI_IRECV( )

sono invece non bloccanti

possibilita' di sovrapposizione tra comunicazione e calcolo

con miglioramento delle prestazioni

MPI_Isend

```
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 subito, senza assicurarsi della ricezione

buf non puo' essere sovrascritto finche' la richiesta è pendente

Occorre controllare lo stato della richiesta di spedizione

Può corrispondere a una ricezione bloccante

13

MPI_Irecv

```
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

Ritorna quando il richiesta di ricezione è stata registrata

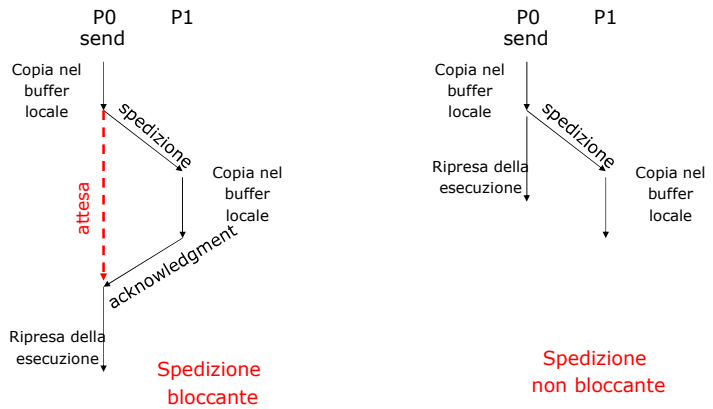
buf non puo' essere usato finche' la richiesta è pendente

Occorre controllare lo stato della richiesta di ricezione

Può corrispondere a una spedizione bloccante

14

Comunicazioni bloccanti vs non bloccanti



15

problema

come si fa a sapere che c'e' un messaggio pronto da leggere?

```
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:
  - nel proc sorgente: buf può essere aggiornato
  - nel proc ricevente: buf contiene i dati ricevuti

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

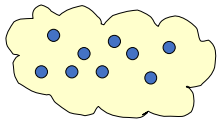
- Ritorna quando l'operazione è conclusa

16

Topologie virtuali

Nel comunicatore MPI_COMM_WORLD i processi non sono organizzati in particolari strutture

Ogni processo puo' interagire con qualunque altro processo



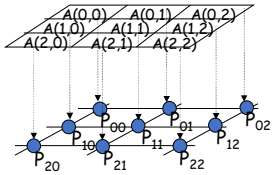
Spesso e' invece utile organizzare i processi in topologie virtuali (anelli, griglie), in maniera da rendere piu' naturale la distribuzione dei dati nei processori:

Esempio: distribuzione dei blocchi

di una matrice su una griglia di nodi

Necessita' di funzioni per la

gestione delle topologie virtuali



Griglia di processori

MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
int *periods, int reorder, MPI_Comm *comm_cart)

comm_old e' il comunicatore

ndims e' la dimensione della griglia

dims(ndims) e' un array. dims(i) e' il numero di proc nella direzione i

period(ndims) specifica se la griglia e' periodica nella direzione i

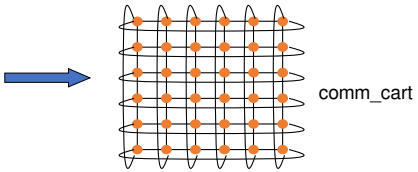
reorder specifica un eventuale riordinamento

comm_cart e' il nuovo comunicatore che definisce la griglia

esempio: ndims=2

dims(0) = dims(1) = 6

period(0) = period(1) = 1



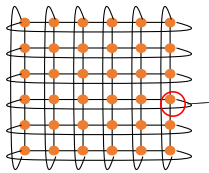
informazioni della griglia

MPI_Cartdim_get(MPI_Comm comm, int *ndims)

a partire dal comunicatore comm (creato con MPI_Cart_create) restituisce la dimensione della griglia ndims

MPI_Cart_get(MPI_Comm comm, int ndims, int *dims,
int *periods, int *coord)

a partire dal comunicatore comm (creato con MPI_Cart_create) restituisce il numero di proc in ogni direzione, l'eventuale periodicita', e le coordinate del proc chiamante



comm

ndims=2

dims(0) = dims(1) = 6
period(0) = period(1) = 1
coord(0)=4 coord(1)=5

comunicatori per sottogriglie

MPI_Cart_sub(MPI_Comm comm, int *directions, MPI_Comm newcomm)

a partire dal comunicatore comm (creato con MPI_Cart_create) restituisce un nuovo comunicatore definite da directions

directions[i] specifica se la direzione i-ma appartiene al nuovo comunicatore

ogni comunicatore consiste dei processi ottenuti facendo variare la coordinate i-ma

all'interno del comunicatore i processi sono numerati come l'indice che varia

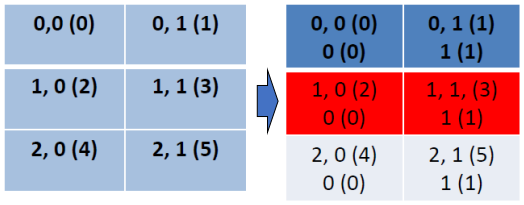
e' possibile creare facilmente comunicatori per le comunicazioni lungo le righe o le colonne

comunicatori per sottogriglie

```
MPI_Cart_sub(MPI_Comm comm,  int *directions, MPI_Comm newcomm)
```

esempio:

```
/* Crea comunicatori lungo le righe*/
directions[0] = 0;
directions[1] = 1; // questa direzione appartiene al comunicatore
MPI_Cart_sub(comm2D, directions, &commrow);
```



griglia 3x2

3 comunicatori di 2 nodi

Operazioni collettive

Sono coinvolti tutti i processi di un comunicatore

Le funzioni sono tutte bloccanti

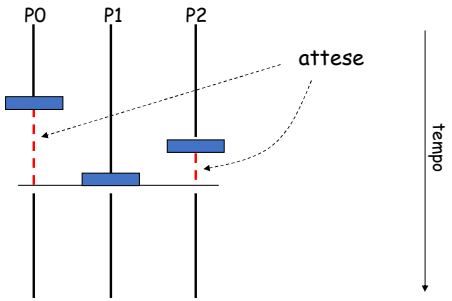
Classi di operazioni

- Sincronizzazione
- distribuzione dati
- Operazioni di riduzione

sincronizzazione

```
Int  MPI_Barrier (MPI_Comm  comm)
```

Blocca il chiamante finché tutti i processi effettuano la chiamata

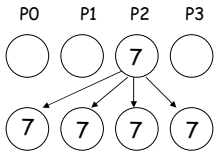


Distribuzione dati

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

Spedisce il contenuto di buffer da root a tutti gli altri processi

Es. Root=2

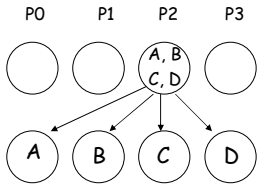


Distribuzione dati

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

distribuisce il contenuto di sendbuf da root a tutti gli altri processi

Es. Root=2



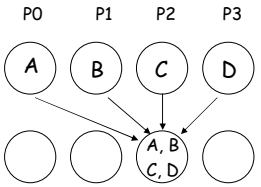
25

Distribuzione dati

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

raccoglie il contenuto di sendbuf dei vari processi in root

Es. Root=2



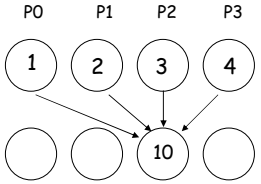
26

Operazioni di riduzione

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

- Raccoglie il risultato di una operazione in root
- Operazioni di max, sum, min, prod
- Operazioni associative definite dall'utente

Es. Sum , root=2



MPI_Allreduce esegue riduzione+replicazione

27