Lezione 4

Dalla prima lezione:

- Obiettivi del calcolo parallelo:

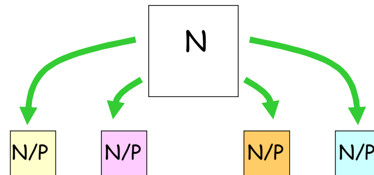
  a) Risolvere uno stesso problema in minor tempo
  b) Risolvere un problema piu' grande nello stesso tempo

- Esempio: previsioni meteorologiche
  - Previsione a 3 giorni in un'ora
  - Con un calcolatore di potenza doppia si puo' fare

  a) Previsione a 3 giorni in mezz'ora
  b) Previsione a 6 giorni in un'ora

a) Uno stesso problema in minor tempo

- Idea di base:
- Decomposizione di un problema di dimensione N in P sottoproblemi indipendenti di uguale dimensione N/P da risolvere concorrentemente



Come misurare l'efficacia della suddivisione

- Misuriamo di quanto si riduce il tempo di esecuzione
- Sia T(N,1) il tempo per la risoluzione del problema di dimensione N con P=1 esecutore
- Idealmente, fissato P>1, ci si aspetta che il tempo di risoluzione di ognuno dei P sottoproblemi sia

$$T(N,P)=T(N,1)/P$$

- Alla base di tale aspettativa ci sono alcune semplificazioni:
  - Suddivisione esatta del problema in P parti
  - Assenza di costi dovuti alla suddivisione
  - Assenza di influenza di P sul tempo di esecuzione

Speed UP

- Idealmente si ha

$$\frac{T(N,1)}{T(N,P)} = P$$

- Il rapporto

$$S_P = \frac{T(N,1)}{T(N,P)}$$ **Speed Up**

misura quanto siamo vicini al caso ideale.

Ovviamente si ha:

nel caso ideale $S_P^* = P$

in pratica $S_P < P$

Efficienza

- Il valore ottenuto dello SpeedUp potrebbe avere poco senso se non lo si confronta al valore ideale

- Il rapporto

$$E_P = \frac{S_P}{S_P^*} = \frac{S_P}{P}$$ **Efficienza**

Misura, in percentuale, quanto siamo vicini allo SpeedUp ideale

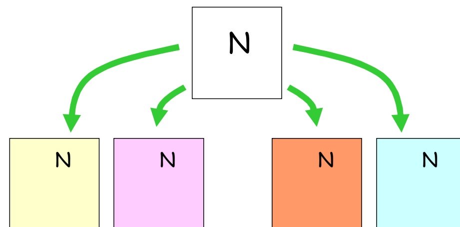
Ovviamente si ha

Caso ideale $E_P^* = 1$

in pratica $E_P < 1$

b) Uno problema piu' grande nello stesso tempo

- Idea di base:
- Risoluzione concorrente di P problemi di dimensione N



N.B. complessivamente si risolve un problema di dimensione PN

Come misurare l'efficacia della suddivisione

- Misuriamo se, aumentando la dimensione dle problema, il tempo rimane costante

- Sia T(N,1) il tempo per la risoluzione di un problema di dimensione N con P=1 esecutore

- Idealmente, fissato P>1, ci si aspetta che il tempo di risoluzione di ciascuno dei P sottoproblemi sia

  T(PN,P)=T(N,1)

- Alla base di tale aspettativa ci sono alcune semplificazioni:
  - Suddivisione esatta del problema in P parti
  - Assenza di costi dovuti alla suddivisione
  - Assenza di influenza di P sul tempo di esecuzione

Efficienza scalata

- Idealmente si ha

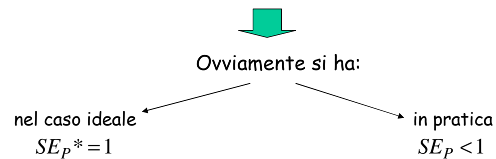
$$\frac{T(N,1)}{T(P \cdot N,P)} = 1$$

- Il rapporto

$$SE_P = \frac{T(N,1)}{T(P \cdot N,P)}$$ **Efficienza Scalata**

Misura, in percentuale, quanto siamo vicini al caso ideale.

Ovviamente si ha:



nel caso ideale $SE_P{*} = 1$

in pratica $SE_P < 1$

SpeedUp Scalato

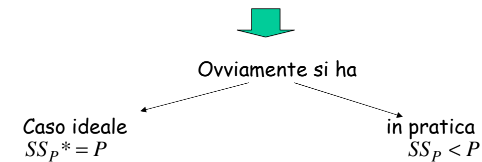
- Anche il valore ottenuto dell'efficienza scalata potrebbe avere poco senso se non lo si confronta con P

- Il rapporto

$$SS_P = P\, SE_P = \frac{P\, T(N,1)}{T(P \cdot N,P)}$$ **Speed Up Scalato**

Misura come usiamo i P processori

Ovviamente si ha



Caso ideale $SS_P{*} = P$

in pratica $SS_P < P$

Caso ideale vs caso reale

- Quali sono i fattori che impediscono di raggiungere il caso ideale?
  - comunicazione / sincronizzazione
  - I/O, accesso al file system
  - generazione di thread e/o processi
  - Inizializzazione di variabili
  - Overhead per la suddivisione del problema

- In generale T(N,1) si puo' decomporre

$$T(N,1) = T_{ser}(N,1) + T_{par}(N,1)$$

con
$T_{par}(N,1)$ (sezione perfettamente parallelizzabile)
$T_{seq}(N,1) > 0$ (sezione sequenziale)

Qual'e' l'impatto?

- Da $T(N,1) = T_{ser}(N,1) + T_{par}(N,1)$ si ha

$$T(N,P) = T_{ser}(N,1) + \frac{T_{par}(N,1)}{P}$$

- Definizione: $\alpha = T_{ser}(N,1) / T(N,1)$
e' la frazione seriale, cioe' la percentuale di codice che non e' parallelizzabile

- SpeedUp

$$S_P = \frac{T(N,1)}{T(N,P)} = \frac{T(N,1)}{T_{seq}(N,1) + T_{par}(N,1)/P} = \frac{1}{\alpha + (1-\alpha)/P}$$

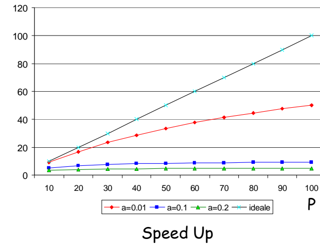
Legge di Amdhal

Impatto della parte seriale su S_P

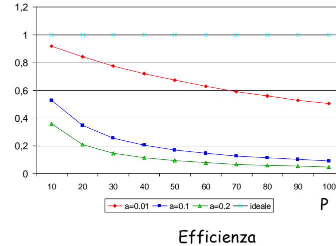
- Cosa accade quando P aumenta?

$$\lim_{P\to\infty} S_P = \lim_{P\to\infty} \frac{1}{\alpha+(1-\alpha)/P} = \frac{1}{\alpha}$$

$$\lim_{P\to\infty} E_P = \lim_{P\to\infty} \frac{S_P}{P} = 0$$



Speed Up



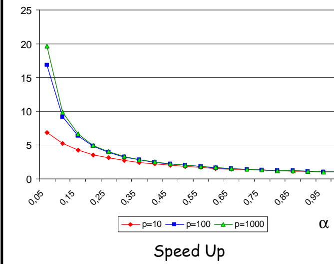
Efficienza

Impatto della parte seriale su S_P

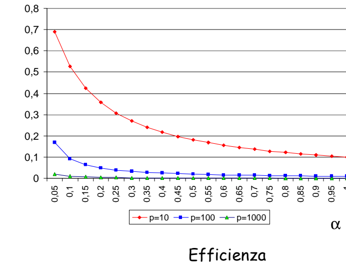
- Cosa accade quando α aumenta?

$$\lim_{\alpha\to 1} S_P = \lim_{\alpha\to 1} \frac{1}{\alpha+(1-\alpha)/P} = 1$$

$$\lim_{\alpha\to 1} E_P = \lim_{\alpha\to 1} \frac{S_P}{P} = \frac{1}{P}$$



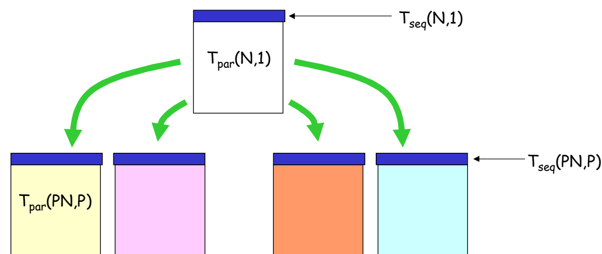
Speed Up



Efficienza

Impatto della parte seriale su S_P

- Che accade quando P e N aumentano tale che
  1. $T_{par}(N,1) = T_{par}(PN,P) =$ cost
     - (dimensione dei sottoproblemi costante al cresere di P)
  2. $T_{seq}(N,1) = T_{seq}(PN,1) =$ cost
     - (la parte sequenziale e' indipendente da P)



Impatto della parte seriale su S_P

- Si ha
  - Dalla 1. ➔ $T_{par}(NP,1) = PT_{par}(N,1)$
  - Dalla 2. ➔ $\lim_{N\to\infty} \alpha = \lim_{N\to\infty} \frac{T_{seq}(N,1)}{T(N,1)} = \lim_{N\to\infty} \frac{cost}{T(N,1)} = 0$
- E quindi

$$S_P = \frac{T(PN,1)}{T(PN,P)} = \frac{T_{seq}(N,1) + P\cdot T_{par}(N,1)}{T(N,1)} = \alpha + (1-\alpha)P = P + (1-P)\alpha$$

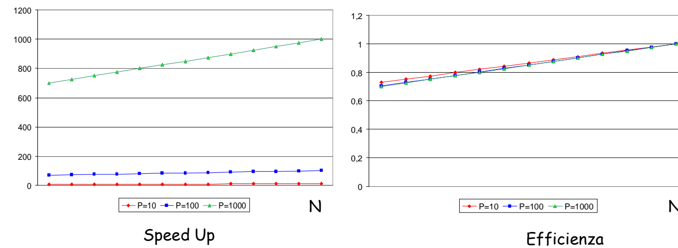
- Da cui

$$\lim_{N\to\infty} S_P = P \qquad \lim_{N\to\infty} E_P = \frac{S_P}{P} = 1$$

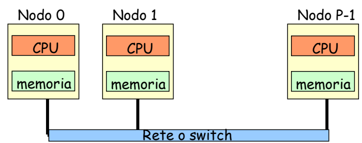
Legge di Gustafson
(reinterpretazione della legge di Amdhal)

Legge di Gustafson

- La legge di Gustafson afferma che se la dimensione del problema aumenta sufficientemente, qualunque efficienza puo' essere ottenuta con un numero qualunque di processori

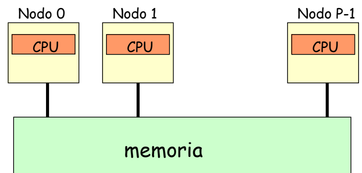


Due modelli base di sistemi paralleli



Sistema a memoria distribuita (o multicomputer)

e' composto da un insieme di nodi di calcolo ognuno con una sua memoria e una sua CPU che comunicano attraverso una rete

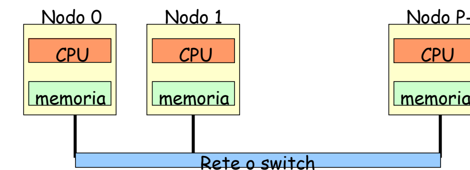


Sistema a memoria condivisa (o multiprocessore)

e' composto da un insieme di nodi di calcolo che accedono ad un'unica memoria

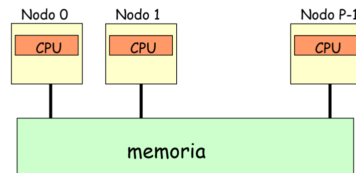
Sistemi a memoria distribuita: caratteristiche

- Non esiste uno spazio di indirizzamento globale
- Necessita' di comunicare dati tra i nodi
- Vantaggi:
  - Economicita', scalabilita'
- Svantaggi:
  - piu' difficili da programmare,
  - costo delle comunicazioni



Sistemi a memoria condivisa: caratteristiche

- spazio di indirizzamento globale
- Necessita' di sincronizzare gli accessi
- Vantaggi:
  - facile condivisione di dati, supporto da parte del S.O.
- Svantaggi:
  - poco scalabile, collo di bottiglia, coerenza delle cache



Per ora …

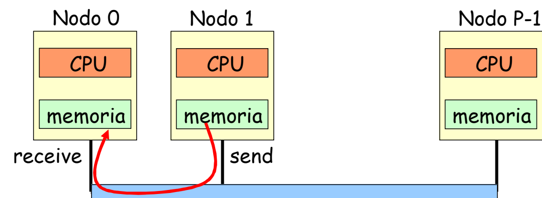
- Occupiamoci dei sistemi a memoria distribuita

Tra qualche lezione…

- passiamo a quelli a memoria condivisa

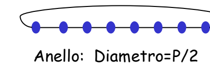
Sistemi a memoria distribuita

- Se un nodo (Nodo 0) ha bisogno di un dato in possesso di altro nodo (Nodo 1) , e' necessario un esplicito trasferimento di dati
- Necessarie istruzioni (o funzioni) di tipo send e receive

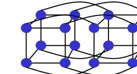


Alcune caratteristiche: topologia

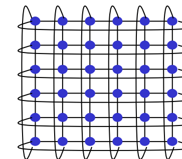
- Topologia: come sono connessi i nodi
- Diametro: la massima distanza tra due nodi in un sistema



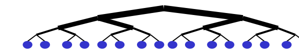
Anello: Diametro=P/2



ipercubo: Diametro=log(P)



Griglia 2D: Diametro=sqrt(2)



ipercubo: Diametro=log(P)

Le moderne reti nascondono il costo dei salti, cosi' che la topologia non e' piu' un fattore critico per le prestazioni degli algoritmi

Alcune caratteristiche: Latenza and Bandwidth

- Il tempo per spedire un messaggio lungo N bytes e' circa

Tempo = latenza + N*costo_byte
= latenza + N/bandwidth

- La topologia e' irrilevante.

- Spesso chiamato "modello α–β" (β = 1/bandwidth)

Tempo = $\alpha + n^*\beta$

- dove $\alpha \gg \beta \gg$ Tempo_flop.
  - Un messaggio "lungo" e' piu' economico di molti messaggi "corti".

$$\alpha + N*\beta \ll N*(\alpha + 1*\beta)$$

  - Spesso sono necessari migliaia di operazioni f.p. per compensare una comunicazione

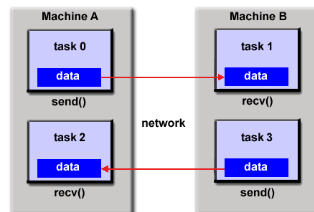
Alcune caratteristiche: latenza e bandwidth

machine	α	β
T3E/Shm	1.2	0.003
T3E/MPI	6.7	0.003
IBM/LAPI	9.4	0.003
IBM/MPI	7.6	0.004
Quadrics/Get	3.267	0.00498
Quadrics/Shm	1.3	0.005
Quadrics/MPI	7.3	0.005
Myrinet/GM	7.7	0.005
Myrinet/MPI	7.2	0.006
Dolphin/MPI	7.767	0.00529
Giganet/VIPL	3.0	0.010
GigE/VIPL	4.6	0.008
GigE/MPI	5.854	0.00872

- Spesso sono ricavati empiricamente
- α e β sono espressi in microsecondi
- Il bandwidth effettivo e' circa l'80% di quello fisico a causa dell'overhead dovuto al pacchetto dati trasmesso
- I parametri dipendono fortemente anche dal software e dai protocolli di trasmissione

Modello di programmazione a scambio di messaggi

- 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)



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

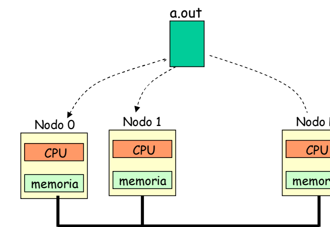
MPI, Message Passing Interface, (1995)

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
      - Colletive: 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?

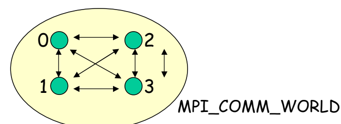
Sviluppo applicazioni con MPI

- 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



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



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

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

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

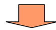
- Lo standard MPI definisce oltre 100 funzioni
- Ma con le 6 funzioni
  - MPI_Init
  - MPI_Finalize
  - MPI_Comm_rank
  - MPI_Comm_size
  - MPI_Send
  - MPI_Recv

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)

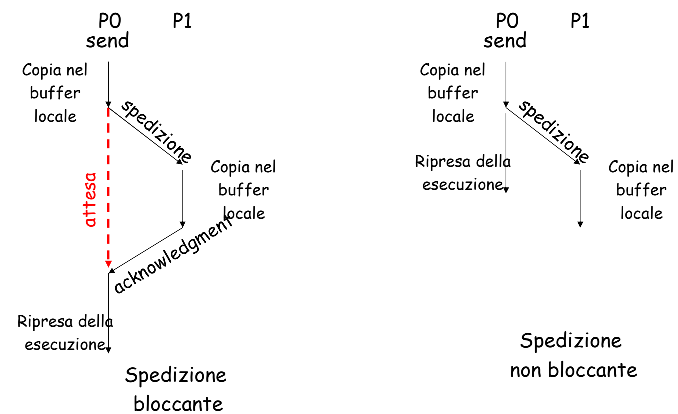
- Inizializza il processo di spedizione
- Ritorna subito, senza assicurarsi della ricezione
- buf non puo' essere sovrascritto finche' la richiesta è pendente
- Occorre controllare lo stato della richiesta di spedizione
- Può corrispondere a una ricezione bloccante

MPI_RECV

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

Comunicazioni bloccanti vs non bloccanti



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 )
```

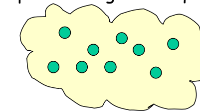
- Ritorna subito dopo aver controllato lo stato
- flag = true se l'operazione è stata completata e:
  - nel proc sorgente: buf può essere aggiornato
  - nel proc ricevente: buf contiene i dati ricevuti

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

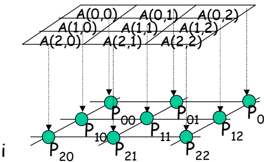
- Ritorna quando l'operazione è conclusa

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



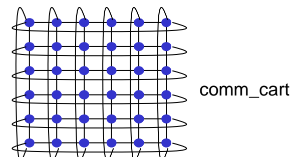
Necessita' di funzioni per la gestione delle topologie virtuali

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



comm_cart

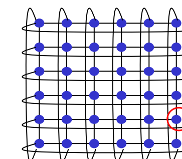
informazioni della griglia

```
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
  ```

Operazioni collettive

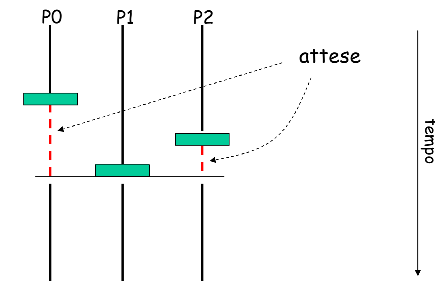
- Sono coinvolti tutti i processi di un comunicatore
- Le funzioni sono tutte bloccanti

- 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

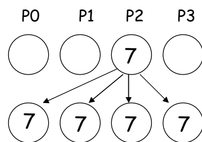


Distribuzione dati

```
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

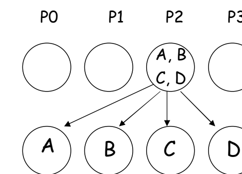


Distribuzione dati

```
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

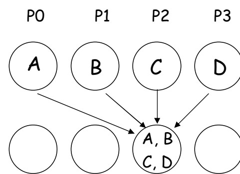


Distribuzione dati

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

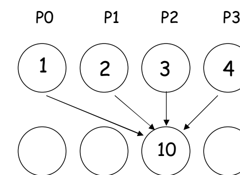


Operazioni di riduzione

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