Corso di Laurea
Magistrale in Informatica

UNIVERSITÀ DEGLI STUDI
DI MODENA E REGGIO EMILIA

Dipartimento di Scienze
Fisiche, Informatiche e Matematiche

Titolare del corso: prof. Marko Bertogna (marko.bertogna@unimore.it)

AA 2016/2017

MPI: concetti di base

Attività di laboratorio

Lab “Zironi”, primo piano dipartimento di Matematica.

iMac Linux (Ubuntu 16.04)
username: calcparX
password: _____

Collegarsi con ssh (usando username e passwd forniti) a aulad??.hpc.unimo.it
(con ??= 02 .. 13) .
Spostarsi nella dir /HOME/calcparX/CALCPARMAG1617.
Creare qui una dir nome.cognome.
Per scrivere un sorgente si può utilizzare vi da remoto (o altri editor disponibili, come gedit, nedit, emacs), oppure scrivere in locale e trasferire il file con scp.

Per compilare un sorgente seriale: icc –o eseguibile sorgente.c
Per lanciarlo: ./eseguibile
Useremo il compilatore intel

Attività di laboratorio

Per compilare un sorgente parallelo linkando automaticamente le librerie mpi:

```
mpiicc –o eseguibile sorgente.c
```

Per lanciare un eseguibile parallelo su 4 processori:

```
mpirun –np 4 eseguibile
```

Installare libreria MPI sul vostro computer

Quasi tutte le distribuzioni linux hanno una implementazione pacchettizata del paragima MPI. Su distribuzioni Debian-based installare la libreria è davvero facile:

apt-get install mpich

In seguito per questa implementazione (non Intel) basterà compilare con il comando:

mpicc –o eseguibile sorgente.c (!!! Non mpiicc !!!)

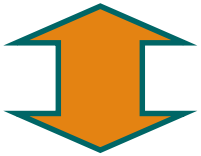
E per lanciare un eseguibile parallelo su 4 processori (come in lab):

mpirun –np 4 eseguibile (alcune volte mpiexec…)

Nonostante la diversità dei compilatori il codice che impareremo a sviluppare è completamente portabile.
Potrete quindi sviluppare comodamente i vostri algoritmi a casa e testarli sulle macchine solo per ottenere **performance migliori**.

Lo standard MPI

Dalla seconda metà degli anni ’90 le macchine parallele hanno cominciato a diffondersi al di fuori dei loro ambienti tradizionali: in centri di ricerca e università più piccole, in aziende private, in enti pubblici.

CAUSA  EFFETTO

I costi sono sensibilmente diminuiti.

Inoltre l’aumento della velocità di interconnessione delle reti LAN ha fatto sì che diventasse ragionevole anche pensare a PC connessi in rete locale come a macchine per il calcolo parallelo.

Infine si sta cominciando a pensare di utilizzare macchine collegate in reti WAN come server di calcolo distribuito (grid).

L’hardware c’è, o almeno siamo sulla buona strada.

Lo standard MPI

La ricerca di algoritmi paralleli è attualmente un settore molto fertile.
Il parallelismo di un codice può nascere:

- dalla fisica (processi indipendenti),
- dalla matematica (set indipendenti di operazioni matematiche),
- dalla fantasia del programmatore.

Se hardware e algoritmi ci sono, perché i programmi paralleli sono poco diffusi ?

L'ostacolo principale alla diffusione di codici paralleli è (stata) la loro difficoltà di sviluppo. Il collo di bottiglia è il software: è spesso faticoso e laborioso (quindi costoso) sviluppare un codice parallelo e se poi questo non è portabile potrebbe non valerne la pena.

Lo standard MPI è nato da questa esigenza di portabilità e semplicità.

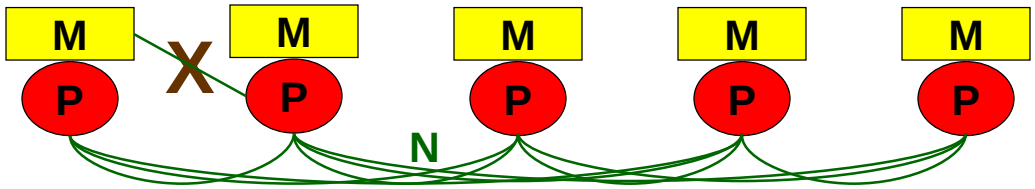
Si tratta di librerie oggi molto diffuse che si presentano al programmatore di un linguaggio ad alto livello (FORTRAN, FORTRAN90, C, C++) con un set di chiamate standard, indipendenti dalla specifica implementazione e dall'hardware su cui girano.

Lo standard MPI

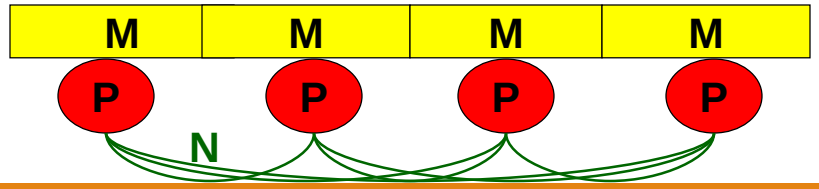
Il modello computazionale che realizzano è quello MESSAGE PASSING.
Altre possibilità sono quelli DATA PARALLEL (HPF, OpenMP)
e SHARED MEMORY (OpenMP, ShMem).

Attenzione a non confondere il modello computazionale (o di programmazione) con con quello architetturale che descrive la macchina (SIMD, MIMD UMA, ecc…).

Nel modello MESSAGE PASSING si suppone che ogni processo abbia una memoria locale (anche se fisicamente può non essere così) e che non possa accedere direttamente alla memoria degli altri processi.



Questo è vero anche se fisicamente la macchina è SMP



Lo standard MPI

I vantaggi del modello MESSAGE PASSING sono:

Universalità	E' naturale usarlo in ambienti eterogenei, in cui i PE sono diversi,
Facilità di debugging	Gli errori più comuni in un programma parallelo consistono nella sovrascrittura involontaria della memoria. Il modello MP costringe a esplicitare ogni processo di comunicazione da ambo le parti.
Performance	Agevola lo sfruttamento della struttura multi-livello della memoria evitando (o riducendo) problemi quali la coerenza di cache.

L'MPI Forum dal 1992 ha lavorato per standardizzare le librerie MPI.
La pagina di riferimento è http://www-unix.mcs.anl.gov/mpi/ (vedere sito del corso)

L'MPI è uno standard. Noi useremo una specifica implementazione: le librerie MPICH.
http://www-unix.mcs.anl.gov/mpi/mpich

MPI: i concetti di base

- Il codice sorgente che scriviamo è uno solo.
- L'eseguibile è comune a tutti i processi e sarà caricato da uno speciale loader su ogni PE.
- Ogni processo conosce (tramite una chiamata di libreria) il numero del PE su cui sta girando.
- Il programmatore prevede esplicitamente (nel codice ad es. con "IF") il branching in modo che ogni PE esegua la propria parte di codice.
- Ogni volta che un processo deve scambiare dati con un altro occorre introdurre istruzioni di *send* e *receive* (o di broadcast, riduzione, ecc…) esplicite in entrambi.
- E' possibile anche includere punti di sincronizzazione in modo che i processi si "aspettino" ad una data istruzione e ripartano insieme.

Programmeremo quindi usando un modello **SPMD** (Single-Program Multiple Data) su una architettura **MIMD**

MPI: i concetti di base

Prima di usare una chiamata (subroutine o funzione) ad una libreria MPI occorre inizializzare le librerie con

MPI_Init

Quando non servono più (o alla fine del programma) occorre dichiarare il termine del loro uso con

MPI_Finalize

Ecco un programma banale che non usa le MPI ma le inizializza e termina..

```
#include <stdio.h>
#include "mpi.h"
int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello world \n" );
    MPI_Finalize();
    return 0;
}
```

MPI: i concetti di base

```
#include <stdio.h>
#include "mpi.h"
int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello world \n" );
    MPI_Finalize();
    return 0;
}
```

- occorre includere *mpi.h*
- le chiamate sono funzioni
- l'error-code è il valore della funzione
- gli arg di MPI_Init sono gli indirizzi degli arg del main
- vari argomenti sono tipi specifici (MPI_Comm, MPI_Datatype,...)

I nomi delle chiamate e gli argomenti sono gli stessi (a parte poche eccezioni).

MPI: i concetti di base

Il modulo (.mod) o l'header (.h) MPI contengono la dichiarazione e definizione di varie costanti indispensabili per l'uso delle MPI. La lista completa è parte dello standard

http://www.mpi-forum.org/docs/mpi-11-html/node169.html#Node169

Ad esempio le costanti che indicano gli error-code sono:

MPI_SUCCESS	MPI_ERR_REQUEST	MPI_ERR_UNKNOWN
MPI_ERR_BUFFER	MPI_ERR_ROOT	MPI_ERR_TRUNCATE
MPI_ERR_COUNT	MPI_ERR_GROUP	MPI_ERR_OTHER
MPI_ERR_TYPE	MPI_ERR_OP	MPI_ERR_INTERN
MPI_ERR_TAG	MPI_ERR_TOPOLOGY	MPI_PENDING
MPI_ERR_COMM	MPI_ERR_DIMS	MPI_ERR_IN_STATUS
MPI_ERR_RANK	MPI_ERR_ARG	MPI_ERR_LASTCODE

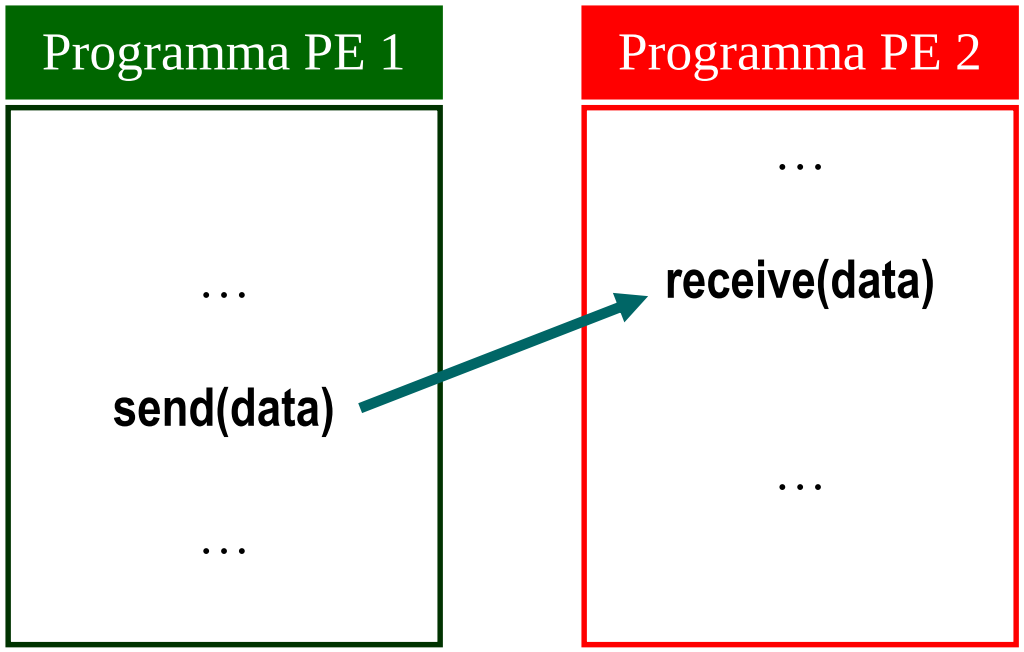
Se in valore della funzione è

$\neq$ MPI_SUCESS

si è verificato un errore nella routine di libreria (e il programma di default abortisce).

MPI: i concetti di base

Per scambiare dati occorre quindi la partecipazioni di entrambi i processi.



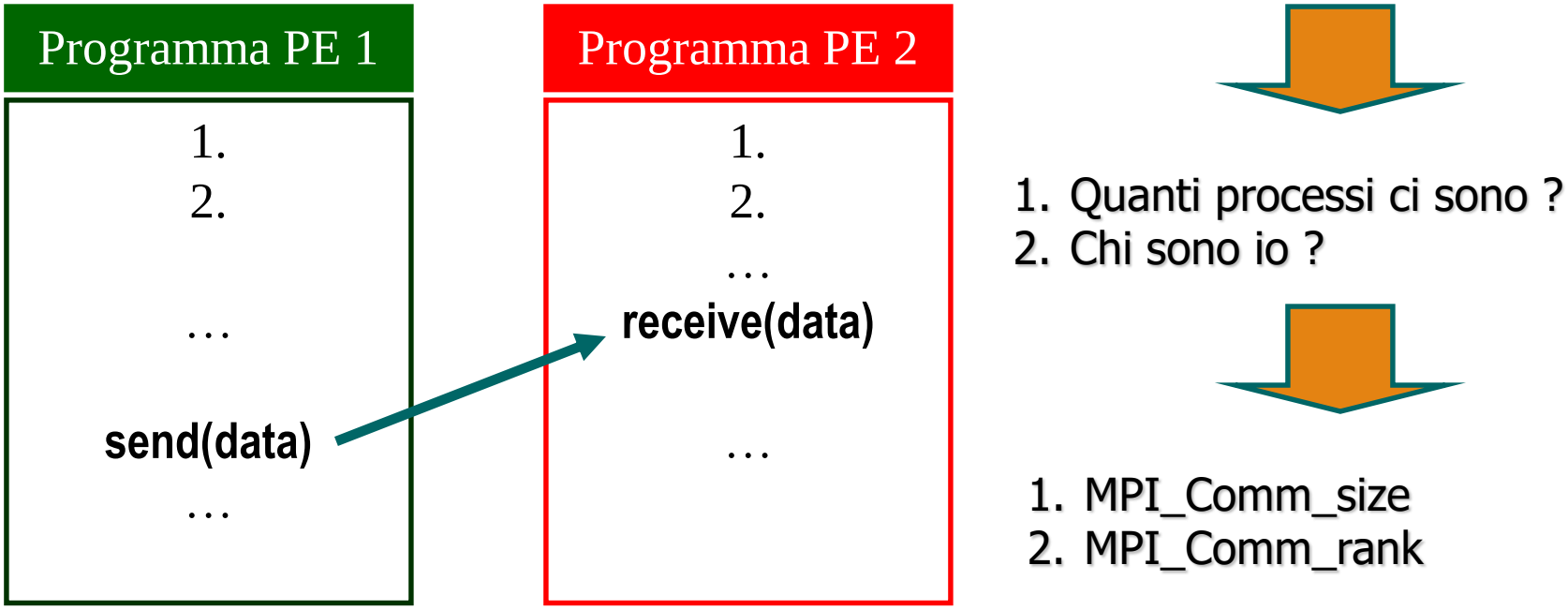
La possibilità di scambiare dati senza la partecipazione esplicita di entrambi (PUT e GET) è parte delle specifiche MPI-2 che noi non trattiamo.

In generale però occorreranno più informazioni alle chiamate *put* e *get*, almeno:

- il PE di destinazione per il *send* e quello di origine per il *get*;
- una etichetta (tag) per identificare il messaggio;
- una descrizione dei dati in transito.

MPI: i concetti di base

Ma prima di cominciare a scambiarsi i dati i programmi devono conoscere l'environment



1. int MPI_Comm_size (MPI_Comm comm, int *numpe)
2. int MPI_Comm_rank (MPI_Comm comm, int *mynumpe)

MPI: i concetti di base

Possiamo dare un senso all'uso delle MPI nei nostri due programmi helloworld.

```
#include <stdio.h>
#include "mpi.h"
int main( int argc, char *argv[] )
{
  int rank, size;
  MPI_Init( &argc, &argv );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  printf( "Hello world from process %d of %d\n", rank, size );
  MPI_Finalize();
  return 0;
}
```

Se come argomento che indica il "communicator" (qui il primo argomento) indichiamo
MPI_COMM_WORLD
vogliamo fare un'operazione che tiene conto di TUTTI i processi. Vedremo altri casi.

MPI: i concetti di base

Adesso che sappiamo inizializzare/terminare le MPI e identificare i processi, facciamoli dialogare.

Abbiamo detto che in teoria serve almeno:

- il PE di destinazione per il *send* e quello di origine per il *get*;
- una etichetta (tag) per identificare il messaggio;
- una descrizione dei dati in transito.

MPI_Send(buffaddr, count, datatype, destinationpe, tag, comm)

MPI_Recv(buffaddr, maxcount, datatype, sourcepe, tag, comm, status)

Queste due istruzioni sono **BLOCKING**. Ovvero l'esecuzione continua solo quando l'operazione indicata (di send o receive, non entrambe) è andata a buon fine.

MPI: i concetti di base

MPI_Send(buffaddr, count, datatype, destinationpe, tag, comm)

MPI_Recv(buffaddr, maxcount, datatype, sourcepe, tag, comm, status)

argomenti:

- buffaddr puntatore all'indirizzo iniziale del buffer da spedire/ricevere. Di solito sarà il nome della variabile o array da mandare o ricevere. (Attenzione: in FORTRAN gli argomenti sono passati sempre per indirizzo, non per contenuto).
- count numero di elementi di tipo datatype che costituiscono il buffer. Se vogliamo ad esempio mandare una singola variabile, sarà =1; se vogliamo mandare un array sarà = dimensione array.
- datatype indica il tipo di dato che stiamo trasmettendo/ricevendo. Può essere un tipo predefinito (MPI_INT, MPI_DOUBLE_PRECISION), una struttura di tipi o un tipo definito da noi.
- destinationpe indica il numero del processo a cui il buffer viene inviato.
- tag permette di mettere un etichetta all'invio in modo da identificarlo quando lo si riceve.
- comm indica il communicator (context+group) nel cui ambito viene effettuata l'operazione.

MPI: i concetti di base

MPI_Send(buffaddr, count, datatype, destinationpe, tag, comm)

MPI_Recv(buffaddr, maxcount, datatype, sourcepe, tag, comm, status)

argomenti:

- buffaddr puntatore all'indirizzo iniziale del buffer da spedire/ricevere. Di solito sarà il nome della variabile o array da mandare o ricevere.
- maxcount numero di elementi di tipo datatype che costituiscono il buffer di ricezione, ovvero massimo numero di elementi che posso ricevere.
- datatype indica il tipo di dato che stiamo trasmettendo/ricevendo.
- sourcepe indica il numero del processo da cui ricevere il contenuto del buffer.
- tag permette di mettere un etichetta alla ricezione in modo da "accoppiarlo" con l'invio corrispondente.
- comm indica il communicator (context+group) nel cui ambito viene effettuata l'operazione.
- status contiene info sull'effettiva lunghezza del messaggio ricevuto, sull'identità effettiva del mittente e sul tag effettivo (utili in receive generici). In FORTRAN è un array di MPI_STATUS_SIZE interi.

MPI: i concetti di base

- MPI_INIT
- MPI_FINALIZE
- MPI_COMM_SIZE
- MPI_COMM_RANK
- MPI_SEND
- MPI_RECV

Con le sei routine introdotte è già possibile scrivere qualunque programma parallelo. Sono però indispensabili altri tipi di chiamate per incrementare efficienza, leggibilità, flessibilità di un codice parallelo.

Consideriamo adesso alcuni semplici esempi!

Sorgenti: **hello_world_mpi.c, Hello_world_mpi2.c, ping_pong_mpi.c**

Mpi_hello_world.c

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>int main (int argc, char* argv[]){
int rank, size;

//Inizializzo la libreria MPI
MPI_Init(&argc, &argv);

//Richiedo il numero totale di processor elements
MPI_Comm_size(MPI_COMM_WORLD, &size);

//Richiedo il mio ID tra questi          MPI_Comm_rank(MPI_COMM_WORLD,
&rank);

printf("CIAO! Sono il processo %d di %d\n", rank, size);

//Chiudo la libreria  MPI_Finalize();
return 0;
}
```

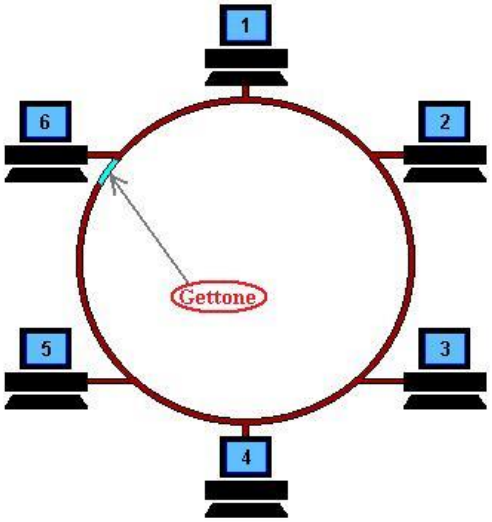
Mpi_hello_world2.c

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

int main (int argc, char* argv[]){
  int rank, size, A; MPI_Status stat;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0){
    printf("Sono il processo %d, inserisci un numero: ",rank); scanf("%d",&A);
    MPI_Send(&A, 1, MPI_INT, 1, 15, MPI_COMM_WORLD);
  }
  if (rank==1){
    MPI_Recv(&A, 1, MPI_INT, 0, 15, MPI_COMM_WORLD, &stat);
    printf("Sono il processo %d, ho ricevuto il valore %d\n", rank, A);
  }

  MPI_Finalize(); return 0;
}
```

Esercizio: Mpi_ring.c

Primo esercizio con più di due processi!

Creare un programma eseguibile con un numero arbitrario di processi (n>1) in cui un valore intero è inizializzato dal processo zero e viene passato sequenzialmente ad ogni processo con ID crescente. L'esecuzione termina quando il processo 0 riceve il valore dall'ultimo processo.



MPI_Status

La funzione MPI_Recv prende come parametro anche l'indirizzo di una struttura MPI_Status (che può essere ignorata con MPI_STATUS_IGNORE).

Se passiamo un MPI_Status alla funzione esso verrà popolato con informazioni aggiuntive riguardo la ricezione del messaggio:

1.L'ID del mittente
2.Il tag del messaggio
3.La lunghezza del messaggio

Funzione per recuperare lunghezza e tipo di dati del messaggio:
MPI_Get_count(MPI_Status* status, MPI_Datatype datatype, int* count)

Esercizio: Mpi_check_status.c

Creare un programma eseguibile con due processi.
Il processo 0 invia un array con un numero random di elementi
Il processo 1 riceve il messaggio e scopre la lunghezza del messaggio mediante la funzione MPI_Get_count.

MPI_Get_count(MPI_Status* status, MPI_Datatype datatype, int* count)

Esercizio completo: Mpi_monte_carlo_pi.c

Il metodo monte-carlo per il calcolo approssimato di pi greco:

*If a circle of radius R is inscribed inside a square with side length 2R, then the area of the circle will be **pi*R^2** and the area of the square will be **(2R)^2**.*
*So the ratio of the area of the circle to the area of the square will be **pi/4**.*

*This means that, if you pick **N** points at random inside the square, approximately **N*pi/4** of those points should fall inside the circle.*

This program picks points at random inside the square. It then checks to see if the point is inside the circle (it knows it's inside the circle if x^2 + y^2 < R^2, where x and y are the coordinates of the point and R is the radius of the circle).

*The program keeps track of how many points it's picked so far (**N**) and how many of those points fell inside the circle (**M**). Pi is then approximated as follows:*

```
pi = 4*M/N
```

Esercizio completo: Mpi_monte_carlo_pi.c

- *Scrivere un programma sequenziale per il calcolo approssimato del pi greco mediante il metodo monte carlo.*

- *Parallelizzarlo mediante MPI*

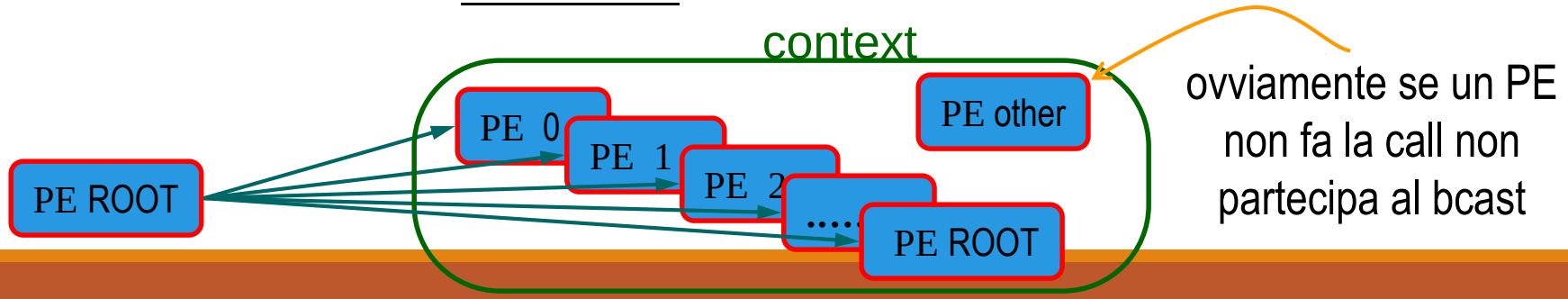
Chiamate di broadcast e riduzione

Spesso occorre che il messaggio sia mandato contemporaneamente a/da più PE.
In questo caso si usa un tipo di chiamata MPI nota come comunicazione collettiva.
Come primi esempi vediamo MPI_Bcast e MPI_Reduce. (parag. 3.1-3.4 Gropp)

MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM)

- BUFFER, COUNT, DATATYPE descrizione del messaggio (come per SEND).
- ROOT numero del PE che manda il messaggio a tutti i processi del gruppo, incluso se stesso.
- COMM indica il communicator (context+group) nel cui ambito viene effettuata l'operazione.

- Alla fine il contenuto del BUFFER di ROOT è copiato in quello di tutti gli altri.
- Deve essere chiamato da tutti i PE del gruppo con gli stessi argomenti.
- E' una comunicazione uno a molti.

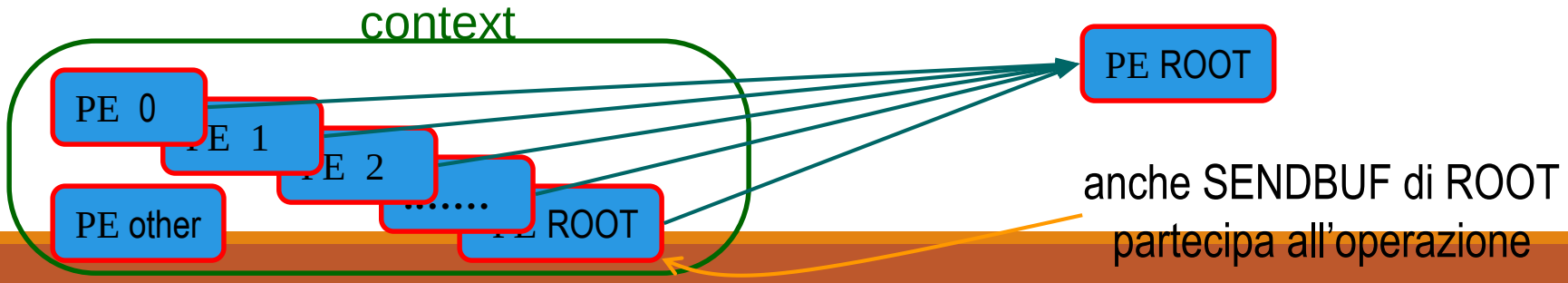


Chiamate di broadcast e riduzione

MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM)

- SENDBUF, RECVBUF, COUNT, DATATYPE descrizione del messaggio e dei buffer di partenza e arrivo.
- OP descrizione dell'operazione da eseguire con tutti i SENDBUF come operatori e il cui risultato va in RECVBUF di ROOT. E' un parametro definito nel modulo MPI (prossima pagina) o una operazione definita dall'utente (vedremo in seguito)
- ROOT numero del PE che raccoglie il risultato dell'operazione.
- COMM indica il communicator nel cui ambito viene effettuata l'operazione.

- Alla fine il contenuto del RECVBUF di ROOT ha il risultato di OP.
- Deve essere chiamato da tutti i PE del gruppo con gli stessi argomenti.
- E' una comunicazione molti a uno che esegue anche un'operazione.



Chiamate di broadcast e riduzione

Le operazioni collettive

MPI_MAX
return the maximum
MPI_MIN
return the minumum
MPI_SUM
return the sum
MPI_PROD
return the product
MPI_LAND
return the logical and
MPI_BAND
return the bitwise and
MPI_LOR
return the logical or
MPI_BOR
return the bitwise of
MPI_LXOR
return the logical exclusive or
MPI_BXOR
return the bitwise exclusive or
MPI_MINLOC
return the minimum and the location (actually, the value of the second element of the structure where the minimum of the first is found)
MPI_MAXLOC
return the maximum and the location

eseguite dalle seguenti chiamate

MPI_REDUCE
MPI_ALLREDUCE
MPI_REDUCE_SCATTER
MPI_SCAN

N.B. non tutte le operazioni sono possibili con tutti i datatype. Ad esempio non si può eseguire un MPI_MAX o MPI_MIN con dati MPI_COMPLEX.

Chiamate di broadcast e riduzione

Guardiamo anche le altre tre chiamate di riduzione.

MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM)

- Alla fine TUTI i PE hanno nel RECVBUF il risultato di OP.
- Gli argomenti sono gli stessi di MPI_REDUCE ma manca ROOT.
- Deve essere chiamato da tutti i PE del gruppo con gli stessi argomenti.
- E' una comunicazione molti a molti che esegue anche un'operazione.

MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM)

- Funziona come MPI_ALLREDUCE ma concorrono a formare il risultato del PE **r** solo i SENDBUF dei PE **da 1 a r**.
- Alla fine TUTI i PE hanno nel RECVBUF il risultato di OP ma su operandi diversi.
- Deve essere chiamato da tutti i PE del gruppo con gli stessi argomenti.
- E' una comunicazione molti a molti che esegue anche un'operazione.

Chiamate di broadcast e riduzione

MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM)

- RECVCOUNTS è un array di tanti elementi quanti sono i PE coinvolti. Ogni elemento è un intero che indica la lunghezza dei RECVBUF di ciascun PE.

- Prima fa l'operazione di Reduce OP sugli Σ_i RECVCOUNTS[i] elementi del vettore SENDBUF distribuito sui vari PE.
- Poi il vettore dei risultati è diviso in tanti segmenti quanti sono i PE e distribuito secondo RECVCOUNTS[i] nei vari RECVBUF.
- Deve essere chiamato da tutti i PE del gruppo con gli stessi argomenti.
- E' una comunicazione molti a molti che esegue anche un'operazione.
- E' equivalente ad un MPI_REDUCE seguito da un MPI_SCATTERV. E' però di solito più efficiente.

Esempio MPI_Reduce_Scatter

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

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

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

   int sendbuf[size];
   int recvbuf;

  for (int i=0; i<size; i++)
      sendbuf[i] = 1 + rank + size*i;

   printf("Proc %d: ", rank);
   for (int i=0; i<size; i++) printf("%d ", sendbuf[i]);
   printf("\n");
   int recvcounts[size];
   for (int i=0; i<size; i++)
      recvcounts[i] = 1;

   MPI_Reduce_scatter(sendbuf, &recvbuf,
         recvcounts, MPI_INT, MPI_MAX,
         MPI_COMM_WORLD);

   printf("Proc %d: %d\n", rank, recvbuf);
   MPI_Finalize();
   return 0;
}
```

Output:

```
Proc 0: 1 5 9 13
Proc 1: 2 6 10 14
Proc 2: 3 7 11 15
Proc 3: 4 8 12 16
```

```
Proc 0: 4
Proc 1: 8
Proc 2: 12
Proc 3: 16
```

Timing dei programmi MPI

Si parallelizza un programma per aumentarne le prestazioni

è essenziale misurarne la velocità

lo standard MPI prevede una semplice chiamata per cronometrare l'esecuzione di un prg

MPI_Wtime ( )

D= MPI_Wtime( )

ritorna un valore DOUBLE PRECISION

Ritorna il tempo (in secondi) passato da un istante arbitrario:
occorrerà sottrarre due istanti per avere l'intervallo.
Lo standard MPI garantisce che l'istante di riferimento non cambi durante il programma.

Per conoscere la risoluzione del clock si usa la funzione DOUBLE PRECISION

MPI_Wtick( )

D= MPI_WTick( )

Dove sta girando il mio programma?

La chiamata CALL MPI_Comm_rank (comm, mynumpe) fornisce un numero identificativo del processo, ma è arbitrariamente assegnato dalle librerie MPI.

Per sapere su quale processore un programma sta girando esiste

MPI_Get_processor_name(char *name, int *resultlen)

La chiamata ritorna il nome del processore (in unix corrisponde all'output di hostame) nella stringa di caratteri name la cui lunghezza deve essere almeno

MPI_MAX_PROCESSOR_NAME

Restituisce anche la lunghezza (in caratteri) del nome del processore nella variabile intera *resultlen*

Attenzione alle macchine **SMP**:

- ✓ il sistema operativo può spostare un programma tra i vari processori;
- ✓ non è detto che la chiamata restituisca il nome del processore particolare, alcune implementazioni restituiscono solo il nome del nodo SMP.

Algoritmi Self-Scheduling (Master-Slave)

Un problema importante da affrontare è il bilanciamento del lavoro tra i processori. Si può suddividere il carico a priori (come per il π) ma in questo modo non teniamo conto di eventuali differenze di prestazioni tra i processori o diversità di tempo richiesto dai processi: se un processore finisce il proprio compito subito, rimarrà inutilizzato.

Semplice soluzione:

- Dividiamo concettualmente il lavoro in più compiti semplici (meglio se numerosi).
- Diamo ad un processo (master) il compito di coordinare il lavoro (ma può anche lavorare).
- Il master assegna un compito a ciascun processo (slave).
- Non appena un processo termina il proprio compito e comunica il risultato, gli viene assegnato il successivo compito della lista, chiunque esso sia.

In questo modo tutti i processori sono occupati fino alla fine, ovvero fino a quando non terminano i compiti. Questo è il self-scheduling. E' particolarmente conveniente quando i processi slave non devono comunicare tra loro.

Vediamo un esempio concreto: Moltiplicazione matrice-vettore.
(ci permetterà di usare le comunicazioni punto-a-punto MPI_SEND e MPI_RECV)

Algoritmi Self-Scheduling: prodotto matrice-vettore

Parte iniziale

```
#define MIN(X, Y) (((X) < (Y)) ? (X) : (Y))

int main(int argc, char *argv[])
{
  const int MAX_ROWS = 1000, MAX_COLS = 1000;
  int myid, master, numprocs, rows, cols;
  int i, j, numsent, sender, anstype, row;
  double *Aloc, *Bloc, *Cloc, *Btemp;  double ans;
  MPI_Status status;

  double *a =(double *) malloc(MAX_ROWS*MAX_COLS*sizeof(double));
  double *b=(double *) malloc(MAX_COLS*sizeof(double));
  double *c=(double *) malloc(MAX_ROWS*sizeof(double));
  double *buffer=(double *) malloc(MAX_COLS*sizeof(double));

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  master = 0; rows = 100; cols = 100;
```

Problema:

calcolare

c = A b

dove A e b sono rispettivamente una matrice quadrata e un vettore, entrambi residenti nella memoria di un processo (master).

descriviamo il programma…

(par. 3.6 Gropp)

Algoritmi Self-Scheduling: prodotto matrice-vettore

Parte master

```
if ( myid == master ){
//master initializes and then dispatches, initialize a and b (arbitrary)
   for(j=0; j<cols; j++){
       b[j] = 1;
       for(i=0; i<row; i++)
           a[i*cols + j];
  }

  //send b to each slave process
  numsent = 0;
  MPI_Bcast(b, cols, MPI_DOUBLE_PRECISION, master,
                MPI_COMM_WORLD);

//send a row to each slave process; tag with row number
for(i = 0; i < MIN(numprocs-1,rows); i++){
  for (j = 0; j<cols; j++){
       buffer[j] = a[i*cols +j];
  }
  MPI_Send(buffer, cols, MPI_DOUBLE_PRECISION, i+1, i,
  MPI_COMM_WORLD);
  numsent = numsent+1;
}
```

```
for(i = 0; i<rows; i++){
    MPI_Recv(&ans, 1, MPI_DOUBLE_PRECISION,
MPI_ANY_SOURCE,
          MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    sender = status.MPI_SOURCE;
    anstype = status.MPI_TAG; //row is tag value
    c[anstype] = ans;
    if (numsent < rows){ //send another row
      for(j = 0; j<cols; j++)
        buffer[j] = a[numsent*cols +j];
     MPI_Send(buffer, cols, MPI_DOUBLE_PRECISION, sender,
numsent,
                MPI_COMM_WORLD);
      numsent = numsent+1;
    }
    else{ //Tell sender that there is no more work
        MPI_Send(MPI_BOTTOM, 0, MPI_DOUBLE_PRECISION,
                  sender, rows, MPI_COMM_WORLD);
    }
}
```

Algoritmi Self-Scheduling: prodotto matrice-vettore

Parte slave

```
} else {
  // slaves receive b, compute dot products until done message recvd
  MPI_Bcast(b, cols, MPI_DOUBLE_PRECISION, master,
                        MPI_COMM_WORLD);

   if (myid <= rows){ //skip if more processes than work
        while(1){
            MPI_Recv(buffer, cols, MPI_DOUBLE_PRECISION,
            master, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            if (status.MPI_TAG == rows){
MPI_Finalize();
                    return 0;
                      }
                      row = status.MPI_TAG;
            ans = 0.0;
            for(i = 0; i<cols; i++)
            ans = ans+buffer[i]*b[i];
            MPI_Send(&ans, 1, MPI_DOUBLE_PRECISION, master,
                        row, MPI_COMM_WORLD);
                }
    }
}
MPI_Finalize(); return 0;
}
```

Ricorda:

Per ricevere un messaggio con qualsiasi tag abbiamo usato la costante **MPI_ANY_TAG**.

Allo stesso modo per ricevere da qualunque processo abbiamo usato **MPI_ANY_SOURCE**.