Corso di Laurea in Informatica

UNIVERSITÀ DEGLI STUDI
DI MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche, Informatiche e Matematiche

Corso
Calcolo Parallelo
Esercitazioni

Titolare del corso: prof. Luca Zanni (luca.zanni@unimore.it)

AA 2018/2019

Chiamate collettive, Timing & Self-scheduling

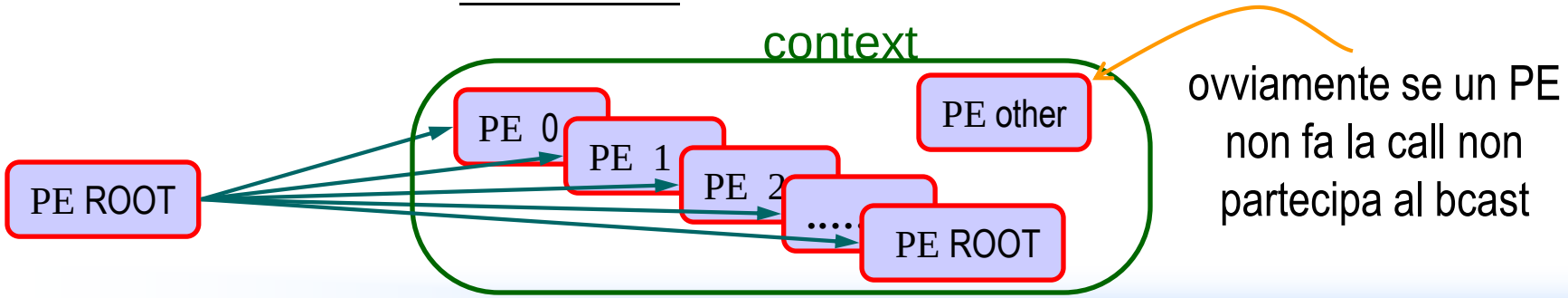
Chiamate di broadcast e riduzione

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.

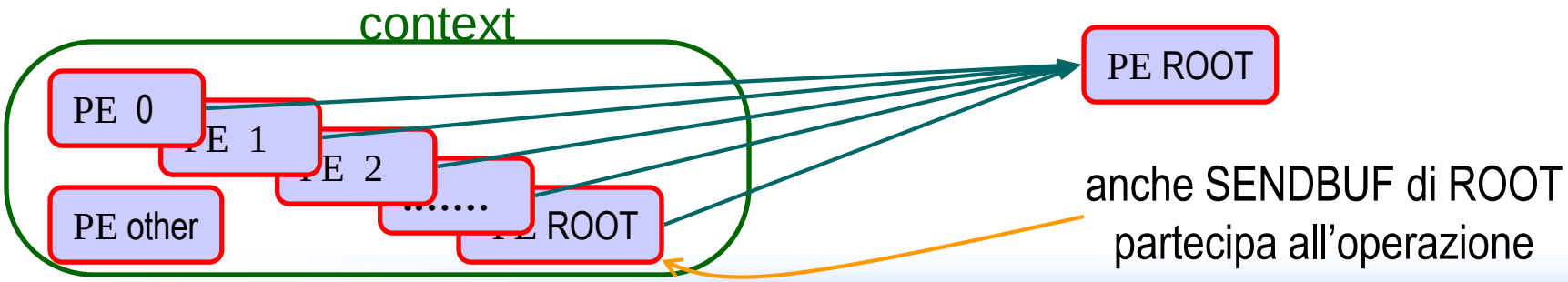


Chiamate di broadcast e riduzione

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:

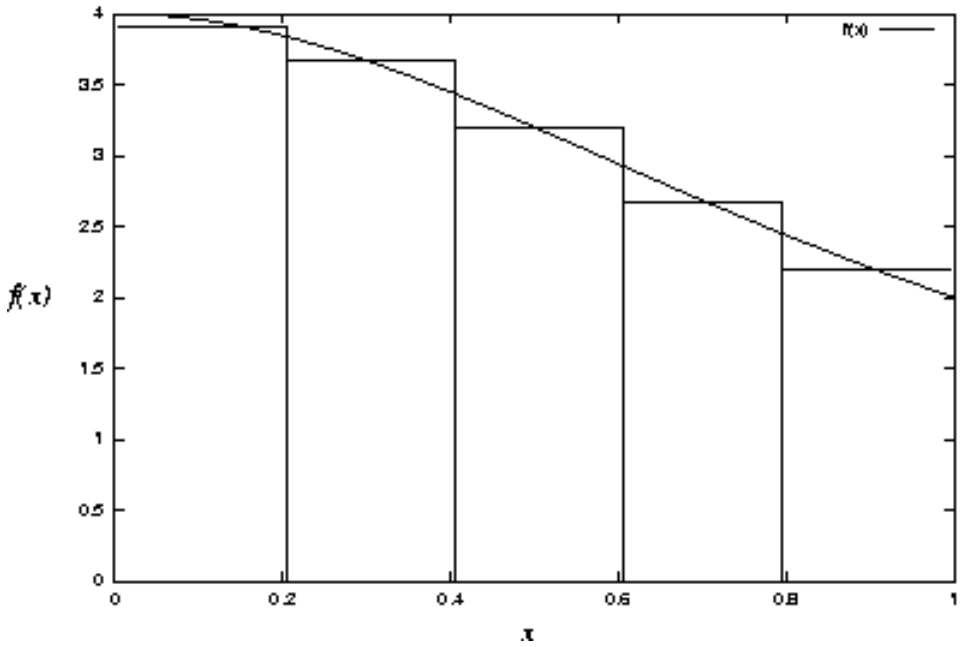
```
Proc 0: 1 5 9 13
Proc 1: 2 6 10 14
Proc 2: 3 7 11 15
Proc 3: 4 8 12 16
```

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

Esercizio mpi_pi.c

Esercizio: calcolare il valore di π usando l'identità (Par. 3.1 Gropp)
E le chiamate collettive MPI_Bcast ed MPI_Reduce

$$\pi = 4\int_0^1 \frac{1}{1+x^2}\,dx$$



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

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

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.

Esercizio

Modificare il programma per il calcolo di π in modo che riporti il tempo trascorso e il nome del nodo su cui un particolare intervallo, scelto dall'utente, è stato calcolato.

MPI_Wtime ( )

MPI_Get_processor_name(char *name, int *resultlen)

Compito per casa:

Studiare BENE le funzioni **MPI_Scatter** e **MPI_Gather**

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**.