Pogramación Paralela y Distribuida

Licenciatura en Ciencias de la Computación

UNCuyo – Facultad de Ingeniería

Ministerio de Educación, Ciencia y Tecnología
Universidad Tecnológica Nacional
Facultad Regional Mendoza

LICPaD
Laboratorio de Investigación en Cómputo Paralelo/Distribuido

UNCUYO UNIVERSIDAD NACIONAL DE CUYO | FACULTAD DE INGENIERÍA

Early Adopters Awarded Fall 2011 (NSF/IEEE)

Programación Paralela y Distribuida

Herramientas para la Programación Paralela y Distribuida

Unidad 5

Ministerio de Educación, Ciencia y Tecnología
Universidad Tecnológica Nacional
Facultad Regional Mendoza

LICPaD
Laboratorio de Investigación en Cómputo Paralelo/Distribuido

UNCUYO UNIVERSIDAD NACIONAL DE CUYO | FACULTAD DE INGENIERÍA

Contenido

- Deadlocks
- Comunicadores
- Definición de tipos de datos
- Comunicación (extras)
- Tiempo
- Ejercicios/ejemplos

Contenido

- Deadlocks
- Comunicadores
- Definición de tipos de datos
- Comunicación (extras)
- Tiempo
- Ejercicios/ejemplos

Deadlocks

- La semántica de *MPI_Recv* y *MPI_Send* presenta algunas **restricciones** sobre cómo mezclarlas
- La implementación de *MPI_Send* en particular puede variar: puede usar **buffering** o ser estrictamente **bloqueante**
- En el primer caso (buffering) MPI **copia** el mensaje en un **buffer** y **retorna** (continúa la ejecución)
- En el segundo caso (bloqueante) **sólo retorna** cuando el **mensaje** ya ha sido **recibido**
- No nos tendría que preocupar la implementación…

Deadlocks

- … pero… cuidado. Veamos un ejemplo:

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
If (myrank ==  0) {
   MPI_Send(a,10, MPI_INT, 1, 1, MPI_COMM_WORLD);
   MPI_Send(b,10, MPI_INT, 1, 2, MPI_COMM_WORLD);
 }
else if (myrank == 1) {
   MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
   MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
```

El proceso 0 envía dos mensajes con diferentes tags al proceso 1

El proceso 1 los recibe pero en orden inverso

Si el **send** usa **buffering**, el código es **correcto**

Si el **send** es **bloqueante**, ambos quedarán en deadlock

Deadlocks

- Los programas deberían funcionar bien sin importar la implementación
- A esto se le llama programa seguro (*safe*)
- El ejemplo anterior es un programa *unsafe*
- Lo correcto es escribir programas olvidando las posibilidades de implementación
- Sólo debe tenerse en cuenta si se está usando una operación **bloqueante** o no

Deadlocks

- Una solución interesante sin cambiar el orden

```
int a[10], b[10], myrank;
MPI_Status status;
MPI_Request requests[2];
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
If (myrank ==  0) {
   MPI_Send(a,10, MPI_INT, 1, 1, MPI_COMM_WORLD);
   MPI_Send(b,10, MPI_INT, 1, 2, MPI_COMM_WORLD);
 }
else if (myrank == 1) {
   MPI_IRecv(b, 10, MPI_INT, 0, 2, &requests[0], MPI_COMM_WORLD);
   MPI_IRecv(a, 10, MPI_INT, 0, 1, &requests[0], MPI_COMM_WORLD);
}
...
```

Si cambiamos el *send* o el *receive* por una operación no bloqueante, el código se tornará seguro y se ejecutará correctamente en cualquier implementación de MPI

Deadlocks

- Recordemos otro ejemplo


```
int my_id, nproc, source;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
MPI_Send(&my_id,1,MPI_INT,(my_id+1)%nproc,1,MPI_COMM_WORLD);
MPI_Recv(&source,1,MPI_INT,MPI_ANY_SOURCE,1,MPI_COMM_WORLD);
```

- En este código cada proceso *i* envía un mensaje a *i*+1 (comunicación circular)
- Cuando el *send* es implementado con *buffering*, el código funciona bien
- Cuando la llamada es bloqueante, todos los procesos entran en una espera infinita, ya que nadie puede efectuar el *receive* (todos han enviado y no pueden avanzar hasta que el otro proceso haya recibido, pero esto no puede ocurrir)

Deadlocks

- Una versión correcta del código anterior

```
int my_id, nproc, source;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
if (myrank%2 == 1) {
  MPI_Send(&my_id,1,MPI_INT,(my_id+1)%nproc,1,MPI_COMM_WORLD);
  MPI_Recv(&source,1,MPI_INT,MPI_ANY_SOURCE,1,MPI_COMM_WORLD);
}
else {
  MPI_Recv(&source,1,MPI_INT,MPI_ANY_SOURCE,1,MPI_COMM_WORLD);
  MPI_Send(&my_id,1,MPI_INT,(my_id+1)%nproc,1,MPI_COMM_WORLD);
}
```

Los procesos impares realizan primero el *send* y luego el *receive*

Los procesos pares reciben el mensaje. Por eso primero hacen el *receive* y luego el *send*

Contenido

- Deadlocks
- Comunicadores
- Definición de tipos de datos
- Comunicación (extras)
- Tiempo
- Ejercicios/ejemplos

Comunicadores

- Todas las comunicaciones efectuadas en MPI están relacionadas a un **comunicador**
- Un comunicador asocia un **grupo** de **procesos**
- Un **grupo** es un conjunto ordenado de procesos donde cada uno es identificado por un número entero denominado rango (***rank***)
- El comunicador es un elemento opaco con un número de atributos y reglas que gobiernan su uso, creación y destrucción
- El comunicador por defecto, que incluye a todos los procesos, es **MPI_COMM_WORLD**

Comunicadores

- Operaciones básicas
  - Duplicación
    - **MPI_Comm_dup** permite crear un duplicado

      `MPI_Comm_dup(comm, newcomm)`
  - Partición
    - **MPI_Comm_split** crea un comunicador con un subconjunto de un grupo de procesos dado

      `MPI_Comm_split(comm,color,key,newcomm)`
    - **MPI_Comm_free** destruye un comunicador

      `MPI_Comm_free(comm)`

Comunicador original | Control para los subgrupos | Control para los rangos procesos | Comunicador nuevo

Comunicadores

- Ejemplo de partición en dos grupos

```
int my_id, new_id;
int main(int argc, char **argv){
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
    MPI_Comm_split(MPI_COMM_WORLD,my_id%2,my_id,&newcomm);
    MPI_Comm_rank(newcomm,&new_id);
    ...
    MPI_Comm_free(newcomm);
    MPI_Finalize();
}
```

Contenido

- Deadlocks
- Comunicadores
- Definición de tipos de datos
- Comunicación (extras)
- Tiempo
- Ejercicios/ejemplos

Definición de tipos de datos

- MPI permite al usuario crear sus propios tipos de datos, llamados datos **derivados**
- A través de estos tipos de dato soporta la comunicación de estructuras complejas tales como arreglos o estructuras que contienen combinaciones de tipos de datos simples
- Un tipo derivado es un objeto opaco que especifica dos cosas:
  - una secuencia de datos primitivos
  - una secuencia de desplazamientos

Definición de tipos de datos

- Ejemplo de uso

```
MPI_Datatype mi_tipodat
MPI_Datatype type[2]={MPI_INT, MPI_DOUBLE};
int blocklengths[2]={2,9};
MPI_Aint displacements[2];
MPI_Aint intex, doublex;
...
MPI_Type_extent (MPI_INT, &intex);
MPI_Type_extent (MPI_DOUBLE, &doublex);
displacements[0]=(MPI_Aint)0;
displacements[1]=2*intex;
MPI_Type_struct(2,blocklengths, displacements,
types, &mi_tipodato);
MPI_Type_commit(&mi_tipodato);
```

Es un tipo especial para el arreglo de desplazamientos

Devuelve la extensión de un tipo (no el tamaño debido a la posible alineación de datos)

Se debe cometer el tipo para que MPI lo reconozca

Contenido

- Deadlocks
- Comunicadores
- Definición de tipos de datos
- Comunicación (extras)
- Tiempo
- Ejercicios/ejemplos

Comunicación (extras)

- Comunicación con buffer
  - Para evitar el riesgo de un bloqueo no deseado se puede solicitar explícitamente que la comunicación se complete copiando el mensaje en un buffer
  - Dicho buffer deberá **asignarlo** el propio **proceso**
  - Así, se elimina el riesgo de bloqueo mientras se espera a que el interlocutor esté preparado (considerando comunicación bloqueante)
  - **MPI_Bsend** tiene el mismo formato que MPI_Send
  - Para que se pueda usar el envío con buffer es necesario que el programador asigne un buffer de salida
  - Para ello hay que reservar previamente una zona de memoria (de forma estática o con *malloc()*)

Comunicación (extras)

  - Lo siguiente es indicarle al sistema que la emplee como buffer. Esta operación la hace **MPI_Buffer_attach**()
  - Se recupera el buffer usando **MPI_Buffer_detach**()

```
int MPI_Buffer_attach(void* buffer, int size);
int MPI_Buffer_detach(void* buffer, int* size);
```

- Recepción por encuesta
  - A veces podemos estar a la espera de mensajes de diferentes clases (con tipos de datos diferente)
  - Si la clase nos viene dada por la etiqueta, nos gustaría saber el valor de la etiqueta antes de leer el mensaje
  - También puede ocurrir que nos llegue un mensaje de longitud desconocida, y resulte necesario averiguar primero el tamaño para así asignar dinámicamente el espacio de memoria

Comunicación (extras)

- Las funciones MPI_Probe() y MPI_Iprobe() nos permiten saber si tenemos un mensaje recibido y listo para leer, pero sin leerlo
- Así podemos averiguar la identidad del emisor del mensaje, la etiqueta del mismo y su longitud
- Una vez hecho esto, podemos proceder a la lectura real del mensaje con la correspondiente llamada a MPI_Recv() o MPI_Irecv()

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm,
    int *flag, MPI_Status *status);
int MPI_Probe(int source, int tag, MPI_Comm comm,
    MPI_Status *status);
```


Contenido

- Deadlocks
- Comunicadores
- Definición de tipos de datos
- Comunicación (extras)
- Tiempo
- Ejercicios/ejemplos

Tiempo

- Medir el tiempo que dura un programa es importante para establecer su eficiencia
- MPI ofrece las funciones **MPI_Wtime** y **MPI_Wtick**
- Ambas proveen alta resolución y poco costo
- **Wtime** devuelve un punto flotante que representa el número de segundos transcurridos a partir de cierto tiempo pasado
- **Wtick** permite saber cuántos segundo hay entre tics sucesivos de reloj. Por ej. si el reloj se incrementa cada 1 milisegundo, entonces wtick retorna 10^{-3}

Contenido

- Deadlocks
- Comunicadores
- Definición de tipos de datos
- Comunicación (extras)
- Tiempo
- Ejercicios/ejemplos

Ejercicio 1. ¿Qué hace el sgte. programa?

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

int main (int argc, char **argv)
{
   int m, n, p, pp;
   double t,tt=0;
   MPI_Status s;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &m);
   MPI_Comm_size(MPI_COMM_WORLD, &n);

   if (m !=0) {
       t=MPI_Wtime();
       MPI_Send(&t,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
   }
   else {
            for (p=1; p<n; p++) {
                    MPI_Recv(&t,1,MPI_DOUBLE,p,0,MPI_COMM_WORLD,&s);
                    if (t>tt)  {
                            tt=t;
                            pp=p;
                    }
            }
           printf("%d\n",pp);
   }
   MPI_Finalize();
}
```


Ejercicio 2

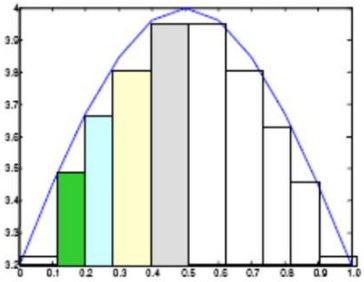
```
#include <mpi.h>
#include <stdio.h>

int main (int argc, char **argv)
{
 int mygid, myoid, npo, color, dato, p;
 MPI_Status status;
 MPI_Comm newcom;

 MPI_Init(&argc, &argv);
 MPI_Comm_rank (MPI_COMM_WORLD, &mygid);

 if (mygid%2 == 0) color = 0;
 else color = 1;

 MPI_Comm_split (MPI_COMM_WORLD, color, mygid, &newcom);
 MPI_Comm_rank (newcom, &myoid);
 MPI_Comm_size(newcom, &npo);

 if (myoid!=0)    MPI_Send(&mygid,1,MPI_INT,0,0,newcom);
 else {
      for (p=1;p<npo;p++)  {
          MPI_Recv(&dato,1,MPI_INT,p,0,newcom,&status);
          printf("%d / %d ------  %d\n",myoid,color,dato);
      }
 }
 MPI_Comm_free (&newcom);
 MPI_Finalize();
}
```

```
user@master:~/$ mpiexec -n 13 ./quehace
0 / 0 ------  2
0 / 1 ------  3
0 / 1 ------  5
0 / 1 ------  7
0 / 1 ------  9
0 / 1 ------  11
0 / 0 ------  4
0 / 0 ------  6
0 / 0 ------  8
0 / 0 ------  10
0 / 0 ------  12
```

Ejercicio 3

- Queremos escribir un programa que calcule el valor de PI en forma paralela.
- Sabemos que:

$$PI = \int_0^1 \frac{4}{1+x^2}$$




```
int main(int argc, char **argv)
{
  int n_interv, myid, numprocs, i;
  double mypi, pi, h, sum=0.0, x;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  if (myid ==0) {
     printf("Ingrese numero de intervalos: \n");
     scanf("%d",&n_interv);
  }

  MPI_Bcast(&n_interv,1,MPI_INT,0,MPI_COMM_WORLD);
  h = 1.0/(double) n_interv;

  for (i=myid+1; i<=n_interv; i+=numprocs) {
     x=h*((double) i - 0.5);
     sum+= 4.0/(1.0 + x*x);
  }
  mypi = h * sum;
  MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
  if (myid == 0) printf("PI es aproximadamente %.16f \n",pi);
  MPI_Finalize();
}
```

Referencias

- **Geist Al et al.** – **"***PVM: Parallel Virtual Machine – A User's guide an tutorial for networked parallel computing"* The MIT Press, 1995 Massachusetts Institute of Technology
- **Grama A. et al**. – "*Introduction to Parallel Computing Second Edition*" Addison-Wesley, 2003
- **Gropp W**. et al. – "*MPICH2 User's Guide Version 1.0.7 - Mathematics and Computer Science Division*" Argonne National Laboratory. 2008.
- **Morrison R. S. – "***Cluster Computing*", 2002
- **Pacheco P.S.** – "Parallel Programming with MPI" Morgan Kaufmann Publishers, Inc. 1997
- **Snir Marc et al**. – "*MPI: The Complete Reference*" The MIT Press, 1996 Cambridge Massachusetts, London England