Sistemas Concurrentes y Distribuidos.
Práctica 3. Implementación de algoritmos distribuidos con MPI.

Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Curso 15-16

Índice

Sistemas Concurrentes y Distribuidos.
Práctica 3. Implementación de algoritmos distribuidos con MPI.

Sección 1
Objetivos

Objetivos

- El objetivo general es iniciarse en la programación de algoritmos distribuidos.
- Conocer varios problemas sencillos de sincronización y su solución distribuida mediante el uso de la interfaz de paso de mensajes MPI:
  - Diseñar una solución distribuida al problema del *productor-consumidor* con buffer acotado para varios productores y varios consumidores, usando MPI.
  - Diseñar diversas soluciones al problema de la *cena de los filósofos* usando MPI.

Sección 2
Productor-Consumidor con buffer acotado en MPI

Subsección 2.1
Aproximación inicial en MPI

Aproximación inicial en MPI

- Supongamos que disponemos de una versión distribuida del problema del productor-consumidor que usa tres procesos y la interfaz de paso de mensajes MPI. Para ello, tendremos un proceso productor (proceso 0 del comunicador universal) que producirá datos, un proceso Buffer (proceso 1) que gestionará el intercambio de datos y un proceso consumidor que procesará los datos (proceso 2). El esquema de comunicación entre estos procesos se muestra a continuación:

- El proceso **Productor** se encarga de ir generando enteros comenzando por el 0 y enviárselos al proceso **Buffer**. El proceso **Consumidor** envía peticiones al proceso Buffer, recibe los enteros de **Buffer**, los imprime por pantalla y calcula su raíz cuadrada.
- El proceso **Buffer** debería atender las peticiones de ambos procesos (**Productor** y **Consumidor**).

Aproximación inicial. Código usando MPI

- Una aproximación inicial al problema se muestra en el siguiente código MPI para tres procesos, que modela la interacción de los mismos de una forma incorrecta al forzar una excesiva sincronización entre productor y consumidor.

```
#include "mpi.h"
...
#define Productor 0
#define Buffer  1
#define Consumidor 2
#define ITERS 20
...
int main( int argc, char *argv[] )
{ int rank, size ;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank ) ;
  MPI_Comm_size( MPI_COMM_WORLD, &size ) ;
  if ( rank == Productor ) productor() ;
  else if ( rank == Buffer ) buffer() ;
  else consumidor() ;
  MPI_Finalize( ) ;
  return 0;
}
```

Proceso Productor, Consumidor y Buffer

```
void productor()
{ for( unsigned int i = 0 ; i < ITERS; i++)
  { cout << "Productor produce valor " << i << endl ;
    MPI_Ssend( &i, 1, MPI_INT, Buffer, 0, MPI_COMM_WORLD );
  }
}
void consumidor()
{ int value, peticion=1; float raiz; MPI_Status status;
  for( unsigned int i = 0 ; i < ITERS ; i++ )
  { MPI_Ssend( &peticion, 1, MPI_INT, Buffer, 0, MPI_COMM_WORLD );
    MPI_Recv ( &value,    1, MPI_INT, Buffer, 0, MPI_COMM_WORLD, &status);
    cout << "Consumidor recibe valor "<<value<<" de Buffer "<<endl ;
    raiz = sqrt(value);
  }
}
void buffer()
{ int value,peticion;  MPI_Status status;
  for( unsigned int i = 0 ; i < ITERS ; i++ )
  { MPI_Recv(&value,    1, MPI_INT, Productor, 0, MPI_COMM_WORLD,&status);
    MPI_Recv(&peticion, 1, MPI_INT, Consumidor, 0, MPI_COMM_WORLD,&status);
    MPI_Ssend( &value,  1, MPI_INT, Consumidor, 0, MPI_COMM_WORLD);
    cout<< "Buffer envia valor " << value << " a Consumidor " << endl ;
  }
}
```


Subsección 2.2
Solución con selección no determinista

Solución con selección no determinista

- Se debe permitir que el productor pueda enviar **TAM** datos sin tener que interrumpirse, y que el consumidor no se retrase cuando haya datos almacenados en el proceso buffer.
- Una forma de corregir dicho código consiste en usar una sentencia de **selección no determinista de órdenes con guarda** en el proceso **Buffer** que permita cierta asincronía entre productor y consumidor en función del tamaño del buffer temporal (**TAM**).
- En MPI, no hay ninguna sentencia de selección no determinista de órdenes con guarda, pero es fácil emularla con las funciones de sondeo **MPI_Probe** y/o **MPI_Iprobe**.

Proceso Buffer con selección no determinista

```
void buffer()
{ int value[TAM], peticion, pos=0,rama;
  MPI_Status status;
  for( unsigned int i = 0 ; i < ITERS*2 ; i++ )
  { if (pos==0) rama=0;          // el consumidor no puede consumir
    else if (pos==TAM) rama=1;  // el productor no puede producir
    else                        // se puede consumir o producir
    { MPI_Probe( MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      if ( status.MPI_SOURCE == Productor ) rama = 0 ; else rama = 1 ;
    }
    switch(rama)
    { case 0 :
       MPI_Recv(&value[pos],1,MPI_INT, Productor,0,MPI_COMM_WORLD,&status);
       cout<< "Buffer recibe "<< value[pos] << " de Prod." << endl ;
       pos++; break;
      case 1 :
       MPI_Recv( &peticion,1,MPI_INT,Consumidor,0,MPI_COMM_WORLD,&status);
       MPI_Ssend( &value[pos-1],1,MPI_INT,Consumidor,0,MPI_COMM_WORLD);
       cout<< "Buffer envia "<< value[pos-1] << " a Cons." << endl ;
       pos--; break;
    }
 }
}
```

Ejercicio propuesto

Extender el programa MPI anteriormente presentado que implementa el productor-consumidor con buffer acotado (los fuentes del programa se proporcionan junto con el guión de prácticas) para que el proceso buffer dé servicio a 5 productores y 4 consumidores. Para ello, se lanzarán 10 procesos y asumiremos que los procesos $0 \ldots 4$ son productores, el proceso **`Buffer`** es el proceso 5 y el resto de procesos en el comunicador universal $(6 \ldots 9)$ son consumidores.

Documentación para el portafolios

Los alumnos redactarán un documento donde se responda de forma razonada a cada uno de los siguientes puntos:

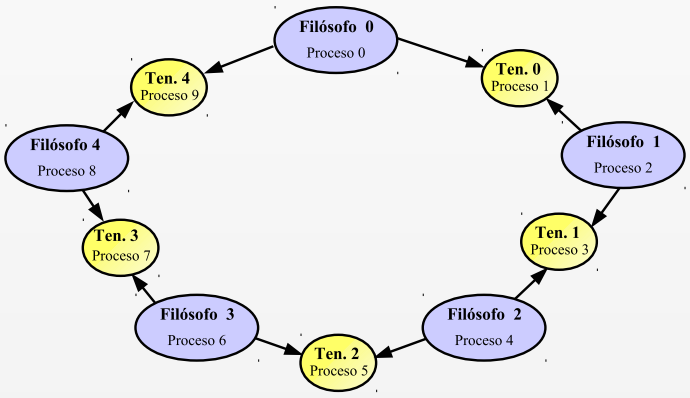
1. Describe qué cambios has realizado sobre el programa de partida y el propósito de dichos cambios.
2. Incluye el código fuente completo de la solución adoptada.
3. Incluye un listado parcial de la salida del programa.

Sección 3
Cena de los Filósofos

Subsección 3.1
Cena de los filósofos en MPI

Cena de los filósofos en MPI.

- Se pretende realizar una implementación del problema de la cena de los filósofos en MPI utilizando el siguiente esquema:
- Tenemos 5 procesos filósofos y 5 procesos tenedor (10 procesos en total). Supondremos que los procs. filósofos se identifican con número pares y los tenedores con números impares. El filósofo i ($i = 0, \ldots, 4$) será el proc. $2i$ y el tenedor i será el $2i+1$.



Cena de los filósofos. Programa principal

```
#include "mpi.h"
#include <iostream>
#include <time.h>
#include <stdlib.h>
  ...
void Filosofo(int id, int nprocesos); // Codigo proc. Filosofo
void Tenedor (int id, int nprocesos); // Codigo proc. Tenedor

int main( int argc, char** argv )
{ int rank,size;
  srand( time(0) );
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  if ( size != 10 )
  { if (rank == 0) cout << "El numero de procesos debe ser 10" << endl;
    MPI_Finalize( ); return 0;
  }
  if ( rank%2 == 0) Filosofo(rank,size); // los pares son filosofos
  else Tenedor(rank,size);               // los impares son tenedores
  MPI_Finalize();
  return 0;
}
```


Procesos filósofos

- En principio, cada filósofo realiza repetidamente la siguiente secuencia de acciones:
  - Pensar (sleep aleatorio).
  - Tomar los tenedores (primero el tenedor izquierdo y después el derecho).
  - Comer (sleep aleatorio).
  - Soltar tenedores (en el mismo orden).
- Las acciones pensar y comer pueden implementarse mediante un mensaje por pantalla seguido de un retardo durante un tiempo aleatorio. Las acciones de tomar tenedores y soltar tenedores pueden implementarse enviando mensajes de petición y de liberación a los procesos tenedor situados a ambos lados de cada filósofo.

Plantilla de la Función Filosofo

```
void Filosofo( int id, int nprocesos )
{
  int izq = (id+1)           % nprocesos ,
      der = (id+nprocesos-1) % nprocesos ;
  while ( true )
  { // solicita tenedor izquierdo
    cout << "Filosofo " << id << " solicita tenedor izq. " << izq << endl;
    // ...
    // solicita tenedor derecho
    cout << "Filosofo " << id << " coge tenedor der. " << der << endl;
    // ...
    cout << "Filosofo " << id << " COMIENDO" << endl ;
    sleep( (rand()%3)+1 );  // comiendo
    // suelta el tenedor izquierdo
    cout << "Filosofo "<<id<< " suelta tenedor izq. " << izq << endl;
    // ...
    // suelta el tenedor derecho
    cout << "Filosofo " << id << " suelta tenedor der. " << der << endl;
    // ...
    cout << "Filosofo " << id << " PENSANDO" << endl;
    sleep( (rand()%3)+1 ); // pensando
 }
}
```


Procesos tenedor

- Un tenedor solamente podrá ser asignado a uno de los dos filósofos que realicen la petición. Hasta que el tenedor no sea liberado no podrá ser asignado de nuevo. Cada proceso tenedor tendrá que ejecutar repetidamente la siguiente secuencia de acciones:
  - Esperar mensajes de petición de tenedor y recibir uno.
  - Esperar mensaje de liberación.

```
void Tenedor( int id, int nprocesos )
{
  int buf, Filo ;
  MPI_Status status;
  while ( true )
  { // Espera un peticion desde cualquier filosofo vecino ...
    // ...
    // Recibe la peticion del filosofo ...
    // ...
    cout << "Ten. " << id << " recibe petic. de " << Filo << endl;
    // Espera a que el filosofo suelte el tenedor...
    // ...
    cout << "Ten. " << id << " recibe liberac. de " << Filo << endl ;
  }
}
```

Ejercicio propuesto

- Implementar una solución distribuida al problema de los filósofos de acuerdo con el esquema descrito en las plantillas. Usar la operación síncrona de envío **`MPI_Ssend`** para realizar las peticiones y liberaciones de tenedores.
- El esquema propuesto (cada filósofo coge primero el tenedor de su izquierda y después el de la derecha) puede conducir a interbloqueo. Identificar la secuencia de peticiones de filósofos que conduce a interbloqueo en el programa y realizar pequeñas modificaciones en el programa (y en el comportamiento de las entidades que participan) que eliminan la posibilidad de interbloqueo (sin añadir nuevos procesos).

Documentación para el portafolios

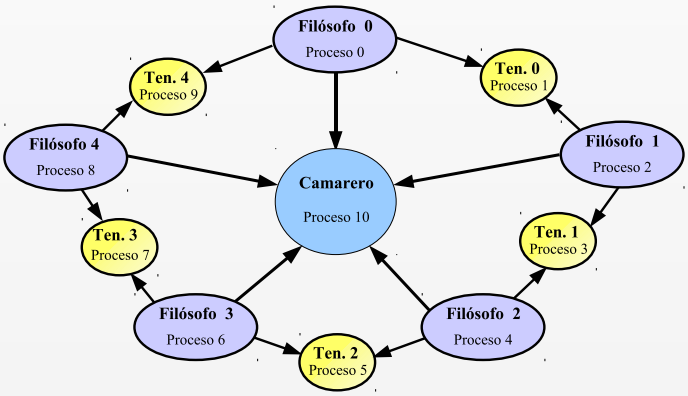
Los alumnos redactarán un documento donde se responda de forma razonada a cada uno de los siguientes puntos:

1. Describe los aspectos más destacados de tu solución al problema de los filósofos, la situación que conduce al interbloqueo y tu solución al problema del interbloqueo.
2. Incluye el código fuente completo de la solución adoptada para evitar la posibilidad de interbloqueo.
3. Incluye un listado parcial de la salida de este programa.

Subsección 3.2
Cena de los filósofos con camarero en MPI

Cena de los filósofos con camarero

- Una forma de evitar la posibilidad de interbloqueo consiste en impedir que todos los filósofos intenten ejecutar la acción de "tomar tenedor" al mismo tiempo. Para ello podemos usar un proceso *camarero central* que permita sentarse a la mesa como máximo a 4 filósofos. Podemos suponer que tenemos 11 procesos y que el camarero es el proc. 10.



Proceso filósofo con Camarero central

- Ahora, cada filósofo ejecutará repetidamente la siguiente secuencia de acciones:
  - Pensar
  - Sentarse
  - Tomar tenedores
  - Comer
  - Soltar tenedores
  - Levantarse
- Cada filósofo pedirá permiso para sentarse o levantarse enviando un mensaje al camarero, el cual llevará una cuenta del número de filósofos que hay sentados a la mesa en cada momento.

Ejercicio propuesto

- Implementar una solución distribuida al problema de los filósofos con camarero central que se ha descrito, usando MPI.
- **Documentación para el portafolios**
  Los alumnos redactarán un documento donde se responda de forma razonada a cada uno de los siguientes puntos:
  1. Describe tu solución al problema de los filósofos con camarero central.
  2. Incluye el código fuente completo de la solución adoptada.
  3. Incluye un listado parcial de la salida del programa.