Práctica 3. Implementación de algoritmos distribuidos con MPI.

1. Múltiples Productores-Consumidores con buffer FIFO por paso de mensajes.

```
// -----------------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos.
// Práctica 3. Implementación de algoritmos distribuidos con MPI
//
// Archivo: prodcons2-mu.cpp
// Implementación del problema del productor-consumidor con
// un proceso intermedio que gestiona un buffer finito y recibe peticiones
// en orden arbitrario
//
// Se realiza el problema de los múltiples productores consumidores con paso de
// mensajes MPI. Se hace con FIFO (primero en entrar es el primero en salir).
// -----------------------------------------------------------------------------

#include <iostream>
#include <thread> // this_thread::sleep_for
#include <random> // dispositivos, generadores y distribuciones aleatorias
#include <chrono> // duraciones (duration), unidades de tiempo
#include <mpi.h>

using namespace std;
using namespace std::this_thread ;
using namespace std::chrono ;

const int
tam_vector           = 10,
// Añadido a la plantilla:
np = 4,              // num. de prods.
nc = 5,              // num. de cons.
m = 20,              // num. de items a producir/consumir
id_min_prod = 0,     // id del primer prod.
id_max_prod = np - 1,   // id del ultimo prod.
id_buffer = np,      // id del buffer.
id_min_cons = np + 1,   // id del primer cons.
id_max_cons = np + nc,  // id del ultimo cons.
num_procesos_esperado = np + nc + 1, // prod + cons + buffer
k = m/np,            // num. de items a producir por cada prod
c = m/nc,            // num. de items a consumir por cada cons
```

```
etiq_prod = 0,          // etiqueta para los mensajes de los prods.
etiq_cons = 1;          // etiqueta para los mensajes de los cons.

//**********************************************************************
// plantilla de función para generar un entero aleatorio uniformemente
// distribuido entre dos valores enteros, ambos incluidos
// (ambos tienen que ser dos constantes, conocidas en tiempo de compilación)
//----------------------------------------------------------------------

template< int min, int max > int aleatorio()
{
static default_random_engine generador( (random_device())() );
static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
return distribucion_uniforme( generador );
}
// ---------------------------------------------------------------------
// producir produce los numeros en funcion del numero de orden del productor
// y lleva espera aleatorio
int producir(int orden_prod)
{
static int
contador[np] = {0} ;
int
inicio = orden_prod * k,
valor_producido = 0;
sleep_for( milliseconds( aleatorio<10,100>()) );
contador[orden_prod]++ ;
valor_producido = inicio + contador[orden_prod];
cout << "Productor " << orden_prod << " ha producido valor " << valor_producido
<< endl << flush;
return valor_producido;
}
// ---------------------------------------------------------------------

void funcion_productor(int orden_prod)
{
for ( unsigned int i= 0 ; i < k ; i++ )
// Poniendo esta condicion, la funcion producir nunca llegara a producir > ik + k -1
{
// producir valor
int valor_prod = producir(orden_prod);
// enviar valor
```

```
cout << "Productor " << orden_prod << " va a enviar valor " << valor_prod
<< endl << flush;
MPI_Ssend( &valor_prod, 1, MPI_INT, id_buffer, etiq_prod, MPI_COMM_WORLD );
}
}
// ----------------------------------------------------------------------

void consumir( int valor_cons, int orden_cons )
{
// espera bloqueada
sleep_for( milliseconds( aleatorio<110,200>()) );
cout << "Consumidor " << orden_cons << " ha consumido valor " << valor_cons << endl <<
flush ;
}
// ----------------------------------------------------------------------

void funcion_consumidor(int orden_cons)
{
int         peticion,
valor_rec = 1 ;
MPI_Status  estado ;

for( unsigned int i=0 ; i < c; i++ )
{
MPI_Ssend( &peticion,  1, MPI_INT, id_buffer, etiq_cons, MPI_COMM_WORLD);
MPI_Recv ( &valor_rec, 1, MPI_INT, id_buffer, etiq_cons, MPI_COMM_WORLD,&estado );
cout << "Consumidor " << orden_cons << " ha recibido valor " << valor_rec << endl << flush ;
consumir( valor_rec, orden_cons );
}
}
// ----------------------------------------------------------------------

void funcion_buffer()
{
int        buffer[tam_vector],      // buffer con celdas ocupadas y vacías
valor,                  // valor recibido o enviado
primera_libre       = 0, // índice de primera celda libre
primera_ocupada     = 0, // índice de primera celda ocupada
num_celdas_ocupadas = 0, // número de celdas ocupadas
etiq_aceptable ;         // etiqueta del emisor aceptable
MPI_Status estado ;                // metadatos del mensaje recibido
```

```
for( unsigned int i=0 ; i < m*2 ; i++ )
{
// 1. determinar si puede enviar solo prod., solo cons, o todos

if ( num_celdas_ocupadas == 0 )              // si buffer vacío
etiq_aceptable = etiq_prod ;              // $~~~$ solo prod.
else if ( num_celdas_ocupadas == tam_vector ) // si buffer lleno
etiq_aceptable = etiq_cons ;              // $~~~$ solo cons.
else                                      // si no vacío ni lleno
etiq_aceptable = MPI_ANY_TAG ;            // $~~~$ cualquiera

// 2. recibir un mensaje del emisor o emisores aceptables

MPI_Recv( &valor, 1, MPI_INT, MPI_ANY_SOURCE, etiq_aceptable, MPI_COMM_WORLD,
&estado );

// 3. procesar el mensaje recibido

switch( estado.MPI_TAG ) // leer emisor del mensaje en metadatos
{
case etiq_prod: // si ha sido el productor: insertar en buffer
buffer[primera_libre] = valor ;
primera_libre = (primera_libre+1) % tam_vector ;
num_celdas_ocupadas++ ;
cout << "Buffer ha recibido valor " << valor << endl ;
break;

case etiq_cons: // si ha sido el consumidor: extraer y enviarle
valor = buffer[primera_ocupada] ;
primera_ocupada = (primera_ocupada+1) % tam_vector ;
num_celdas_ocupadas-- ;
cout << "Buffer va a enviar valor " << valor << endl ;
MPI_Ssend( &valor, 1, MPI_INT, estado.MPI_SOURCE, etiq_cons, MPI_COMM_WORLD);
break;
}
}
}

// ----------------------------------------------------------------------

int main( int argc, char *argv[] )
{
```

```
int id_propio, num_procesos_actual;

// inicializar MPI, leer identif. de proceso y número de procesos
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual );

if ( num_procesos_esperado == num_procesos_actual )
{
// ejecutar la operación apropiada a 'id_propio'
if ( id_propio >= id_min_prod && id_propio <= id_max_prod )
funcion_productor(id_propio);
else if ( id_propio == id_buffer )
funcion_buffer();
else
funcion_consumidor(id_propio - id_min_cons);
}
else
{
if ( id_propio == 0 ) // solo el primero escribe error, indep. del rol
{ cout << "el número de procesos esperados es:    " << num_procesos_esperado << endl
<< "el número de procesos en ejecución es: " << num_procesos_actual << endl
<< "(programa abortado)" << endl ;
}
}

// al terminar el proceso, finalizar MPI
MPI_Finalize( );
return 0;
}
```

Práctica 3. Implementación de algoritmos distribuidos con MPI.

2. Cena de filósofos (con interbloqueo)

```
// -----------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos.
// Práctica 3. Implementación de algoritmos distribuidos con MPI
//
// Archivo: filosofos-interb.cpp
// Implementación del problema de los filósofos (sin camarero).
// Plantilla para completar.
//
// 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.
// -----------------------------------------------------------------------

#include <mpi.h>
#include <thread> // this_thread::sleep_for
#include <random> // dispositivos, generadores y distribuciones aleatorias
#include <chrono> // duraciones (duration), unidades de tiempo
#include <iostream>

using namespace std;
using namespace std::this_thread ;
using namespace std::chrono ;

const int
num_filosofos = 5 ,              // número de filósofos
num_filo_ten  = 2*num_filosofos, // número de filósofos y tenedores
num_procesos  = num_filo_ten ,   // número de procesos total (por ahora solo hay filo y ten)
// Añadido a la plantilla:
etiq_fi_coge = 0,                // etiqueta de que el filosofo coge el tenedor
etiq_fi_suelta = 1;              // etiqueta de que el filosofo suelta el tenedor


//**********************************************************************
// plantilla de función para generar un entero aleatorio uniformemente
// distribuido entre dos valores enteros, ambos incluidos
// (ambos tienen que ser dos constantes, conocidas en tiempo de compilación)
//----------------------------------------------------------------------
```

```
template< int min, int max > int aleatorio()
{
static default_random_engine generador( (random_device())() );
static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
return distribucion_uniforme( generador );
}

// ---------------------------------------------------------------------

void funcion_filosofos( int id )
{
int id_ten_izq = (id+1)              % num_filo_ten, //id. tenedor izq.
id_ten_der = (id+num_filo_ten-1) % num_filo_ten, //id. tenedor der.
valor;

while ( true )
{
cout <<"Filósofo " <<id << " solicita ten. izq." <<id_ten_izq <<endl;
// ... solicitar tenedor izquierdo (completar)
MPI_Ssend( &valor, 1, MPI_INT, id_ten_izq, etiq_fi_coge, MPI_COMM_WORLD);

cout <<"Filósofo " <<id <<" solicita ten. der." <<id_ten_der <<endl;
// ... solicitar tenedor derecho (completar)
MPI_Ssend( &valor, 1, MPI_INT, id_ten_der, etiq_fi_coge, MPI_COMM_WORLD);

cout <<"Filósofo " <<id <<" comienza a comer" <<endl ;
sleep_for( milliseconds( aleatorio<10,100>() ) );

cout <<"Filósofo " <<id <<" suelta ten. izq. " <<id_ten_izq <<endl;
// ... soltar el tenedor izquierdo (completar)
MPI_Ssend( &valor, 1, MPI_INT, id_ten_izq, etiq_fi_suelta, MPI_COMM_WORLD);

cout<< "Filósofo " <<id <<" suelta ten. der. " <<id_ten_der <<endl;
// ... soltar el tenedor derecho (completar)
MPI_Ssend( &valor, 1, MPI_INT, id_ten_der, etiq_fi_suelta, MPI_COMM_WORLD);

cout << "Filosofo " << id << " comienza a pensar" << endl;
sleep_for( milliseconds( aleatorio<10,100>() ) );
}
}
// ---------------------------------------------------------------------
```

```
void funcion_tenedores( int id )
{
int valor, id_filosofo ;  // valor recibido, identificador del filósofo
MPI_Status estado ;       // metadatos de las dos recepciones

while ( true )
{
// ..... recibir petición de cualquier filósofo (completar)
MPI_Recv( &valor, 1, MPI_INT, MPI_ANY_SOURCE, etiq_fi_coge, MPI_COMM_WORLD,
&estado );

// ..... guardar en 'id_filosofo' el id. del emisor (completar)
id_filosofo = estado.MPI_SOURCE;
cout <<"Ten. " <<id <<" ha sido cogido por filo. " <<id_filosofo <<endl;

// ..... recibir liberación de filósofo 'id_filosofo' (completar)
MPI_Recv( &valor, 1, MPI_INT, id_filosofo, etiq_fi_suelta, MPI_COMM_WORLD, &estado);
cout <<"Ten. "<< id<< " ha sido liberado por filo. " <<id_filosofo <<endl ;
}
}
// ----------------------------------------------------------------------

int main( int argc, char** argv )
{
int id_propio, num_procesos_actual ;

MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual );


if ( num_procesos == num_procesos_actual )
{
// ejecutar la función correspondiente a 'id_propio'
if ( id_propio % 2 == 0 )          // si es par
funcion_filosofos( id_propio ); //   es un filósofo
else                               // si es impar
funcion_tenedores( id_propio ); //   es un tenedor
}
else
{
if ( id_propio == 0 ) // solo el primero escribe error, indep. del rol
```

```
{ cout << "el número de procesos esperados es:    " << num_procesos << endl
<< "el número de procesos en ejecución es: " << num_procesos_actual << endl
<< "(programa abortado)" << endl ;
}
}

MPI_Finalize( );
return 0;
}

// --------------------------------------------------------------------
```

Práctica 3. Implementación de algoritmos distribuidos con MPI.

3. Cena de los filósofos con camarero

```
// -----------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos.
// Práctica 3. Implementación de algoritmos distribuidos con MPI
//
// Archivo: filosofos-camarero.cpp
// Implementación del problema de los filósofos (sin camarero).
// Plantilla para completar.
//
// Implementar una solución distribuida al problema de los filósofos de basado
// en un proceso camarero con espera selectiva. La espera selectiva se consigue
// con el uso de etiquetas.
// -----------------------------------------------------------------------

#include <mpi.h>
#include <thread> // this_thread::sleep_for
#include <random> // dispositivos, generadores y distribuciones aleatorias
#include <chrono> // duraciones (duration), unidades de tiempo
#include <iostream>

using namespace std;
using namespace std::this_thread ;
using namespace std::chrono ;

const int
num_filosofos = 5 ,              // número de filósofos
num_filo_ten  = 2*num_filosofos, // número de filósofos y tenedores
num_procesos  = num_filo_ten + 1,// número de procesos total (filo, ten y cam)
// Añadido a la plantilla:
etiq_fi_coge = 0,                // etiqueta de que el filosofo coge el tenedor
etiq_fi_suelta = 1,              // etiqueta de que el filosofo suelta el tenedor
// Añadido a la solución con el camarero
etiq_sentarse = 2,               // etiqueta de que el filosofo se sienta en la mesa
etiq_levantarse = 3,             // etiqueta de que el filosofo se levanta de la mesa
id_camarero = 10;                // identificador del camarero.

//**********************************************************************
// plantilla de función para generar un entero aleatorio uniformemente
```

```
// distribuido entre dos valores enteros, ambos incluidos
// (ambos tienen que ser dos constantes, conocidas en tiempo de compilación)
//----------------------------------------------------------------------

template< int min, int max > int aleatorio()
{
static default_random_engine generador( (random_device())() );
static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
return distribucion_uniforme( generador );
}

// ---------------------------------------------------------------------

void funcion_filosofos( int id )
{
int id_ten_izq = (id+1)              % num_filo_ten, //id. tenedor izq.
id_ten_der = (id+num_filo_ten-1) % num_filo_ten, //id. tenedor der.
valor;

while ( true ){
// 1

cout <<"Filósofo " <<id << " solicita sentarse."<<endl;
// Solicitar si puede sentarse en la mesa.
MPI_Ssend( &valor, 1, MPI_INT, id_camarero, etiq_sentarse, MPI_COMM_WORLD);

// 2

cout <<"Filósofo " <<id << " solicita ten. izq." <<id_ten_izq <<endl;
// ... solicitar tenedor izquierdo (completar)
MPI_Ssend( &valor, 1, MPI_INT, id_ten_izq, etiq_fi_coge, MPI_COMM_WORLD);

cout <<"Filósofo " <<id <<" solicita ten. der." <<id_ten_der <<endl;
// ... solicitar tenedor derecho (completar)
MPI_Ssend( &valor, 1, MPI_INT, id_ten_der, etiq_fi_coge, MPI_COMM_WORLD);

// 3

cout <<"Filósofo " <<id <<" comienza a comer" <<endl ;
sleep_for( milliseconds( aleatorio<10,100>() ) );

// 4
```

```
cout <<"Filósofo " <<id <<" suelta ten. izq. " <<id_ten_izq <<endl;
// ... soltar el tenedor izquierdo (completar)
MPI_Ssend( &valor, 1, MPI_INT, id_ten_izq, etiq_fi_suelta, MPI_COMM_WORLD);

cout<< "Filósofo " <<id <<" suelta ten. der. " <<id_ten_der <<endl;
// ... soltar el tenedor derecho (completar)
MPI_Ssend( &valor, 1, MPI_INT, id_ten_der, etiq_fi_suelta, MPI_COMM_WORLD);

// 5

cout <<"Filósofo " <<id << " solicita levantarse."<<endl;
// Solicitar si puede levantarse de la mesa.
MPI_Ssend( &valor, 1, MPI_INT, id_camarero, etiq_levantarse, MPI_COMM_WORLD);

// 6

cout << "Filosofo " << id << " comienza a pensar" << endl;
sleep_for( milliseconds( aleatorio<10,100>() ) );
}
}
// ----------------------------------------------------------------------

void funcion_tenedores( int id )
{
int valor, id_filosofo ;  // valor recibido, identificador del filósofo
MPI_Status estado ;       // metadatos de las dos recepciones

while ( true )
{
// ..... recibir petición de cualquier filósofo (completar)
MPI_Recv( &valor, 1, MPI_INT, MPI_ANY_SOURCE, etiq_fi_coge, MPI_COMM_WORLD,
&estado );

// ..... guardar en 'id_filosofo' el id. del emisor (completar)
id_filosofo = estado.MPI_SOURCE;
cout <<"Ten. " <<id <<" ha sido cogido por filo. " <<id_filosofo <<endl;

// ..... recibir liberación de filósofo 'id_filosofo' (completar)
MPI_Recv( &valor, 1, MPI_INT, id_filosofo, etiq_fi_suelta, MPI_COMM_WORLD, &estado);
cout <<"Ten. "<< id<< " ha sido liberado por filo. " <<id_filosofo <<endl ;
}
```

```
}
// ----------------------------------------------------------------------

void funcion_camarero()
{
int etiq_puede = 999;   // etiqueta de lo que puede hacer el filosofo
int s = 0,  // numero de filosofos sentados.
valor,  // valor recibido
id_filo;//id del filosofo que llama al camarero
MPI_Status estado ;       // metadatos de las dos recepciones

while ( true )
{
if(s == num_filosofos-1)   // si s == 4, sólo puede levantarse
etiq_puede = etiq_levantarse;
else if (s == 0)                 // si s == 0, solo puede sentarse
etiq_puede = etiq_sentarse;
else                             // si no, puede las dos
etiq_puede = MPI_ANY_TAG;

// ..... recibir petición de cualquier filósofo
MPI_Recv( &valor, 1, MPI_INT, MPI_ANY_SOURCE, etiq_puede, MPI_COMM_WORLD,
&estado );

id_filo = estado.MPI_SOURCE;

if(estado.MPI_TAG == etiq_sentarse){
s++;
cout <<"Filósofo " <<id_filo << " se ha sentado."<<endl;
} else if (estado.MPI_TAG == etiq_levantarse){
s--;
cout <<"Filósofo " <<id_filo << " se ha levantado."<<endl;
}
}
}
// ----------------------------------------------------------------------ç

int main( int argc, char** argv )
{
int id_propio, num_procesos_actual ;

MPI_Init( &argc, &argv );
```

```
MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual );


if ( num_procesos == num_procesos_actual )
{
// ejecutar la función correspondiente a 'id_propio'
if( id_propio == 10)                  // si es 10
funcion_camarero();                 // es el camarero
else if ( id_propio % 2 == 0 )          // si es par
funcion_filosofos( id_propio );     //   es un filósofo
else                                  // si es impar
funcion_tenedores( id_propio );     //   es un tenedor
}
else
{
if ( id_propio == 0 ) // solo el primero escribe error, indep. del rol
{ cout << "el número de procesos esperados es:    " << num_procesos << endl
<< "el número de procesos en ejecución es: " << num_procesos_actual << endl
<< "(programa abortado)" << endl ;
}
}

MPI_Finalize( );
return 0;
}

// ----------------------------------------------------------------------
```