ETSIIT
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Ingeniería Informática + ADE

Universidad de Granada (UGR)

Autor: Ismael Sallami Moreno

Asignatura: 2º Parcial SCD

Índice

1 Examen Carlos Ureña 2014-2015

1.1. Enunciado

Asignatura: Sistemas Concurrentes y Distribuidos.
Año Académico: 2014/2015.
Grado: Doble Grado en Ingeniería Informática y ADE + Matemáticas.
Descripción: Examen correspondiente a la práctica 3 de SCD.
Profesor: Carlos Ureña.
Fecha: 12 de enero de 2015.

1.1.1. Ejercicio 1. Corrección de errores (6 puntos)

Considerar el código del archivo disponible en la subsection de plantilla de este documento o aquí, que intenta plantear una solución al problema de la cena de los filósofos con camarero. Hay diversos errores en el código proporcionado que deberás solucionar:

1. Intenta compilar el código proporcionado. Comprueba que hay diversos errores básicos de sintaxis que impiden que el código genere un ejecutable. Encuentra dichos errores y arréglalos hasta que el código compile correctamente. Indica en el folio del examen el número de línea y cómo debería quedar ésta una vez solventado el problema.

2. Ejecuta el programa y observa la salida que genera. El algoritmo presenta un error en su diseño. Explica brevemente en el folio del examen por qué la salida proporcionada no es correcta.

3. Observa el código del programa ejecutado. Encuentra y corrige el error en el diseño del algoritmo que impide que el programa muestre una salida coherente. Señala en el folio del examen en qué línea está el problema y cómo debería quedar ésta para solucionar el error.

4. Se desea modificar el algoritmo dado del problema de los filósofos con camarero para dar servicio a 7 filósofos y 7 tenedores en lugar de a 5. También se desea cambiar el identificador de los procesos para que los filósofos sean los procesos con rank 0, 1, 2, ..., 6 y los tenedores sean los procesos 7, 8, ..., 13. Describe brevemente qué cambios debes realizar en el código proporcionado y en qué líneas.

1.1.2. Ejercicio 2. Problema de múltiples Productores-Consumidores con Paso de Mensajes (4 puntos)

Partiendo del algoritmo desarrollado durante la Práctica 3 para resolver el problema de múltiples Productores-Consumidores usando paso de mensajes MPI, se pide realizar los siguientes cambios:

- El valor que los procesos productores que envían al buffer debe ser un valor aleatorio entre 0 y 9 en lugar de uno secuencial como hasta ahora.

- Así mismo, el productor deberá enviar un mensaje al buffer pidiendo permiso para enviar el valor, y el buffer deberá responder al productor autorizando a enviar el mensaje antes de hacerlo (al igual que ya hace el consumidor).

Realiza los cambios necesarios para que, en el caso de que en el buffer se almacenen dos valores iguales consecutivos (independientemente del productor que los envíe), el buffer finalice.

- Antes de finalizar deberá avisar a todos los productores y consumidores para que estos finalicen enviando un valor especial en los mensajes de autorización (por ejemplo, un valor negativo). Sólo cuando todos los procesos hayan finalizado el buffer terminará.
- Implementa esta condición de forma que cuando cualquier proceso vaya a finalizar, imprima un mensaje en pantalla informando de ello.

1.1.3. Plantilla

```
#include <iostream>
#include <time.h>
#include <unistd.h>
#include <stdlib.h>
#include <mpi.h>

#define NUM_FILOSOFOS 5
#define NUM_PROCESOS (2*(NUM_FILOSOFOS)+1)
#define RANK_CAMARERO ((NUM_PROCESOS)-1)

#define TAG_SENTARSE 0
#define TAG_LEVANTARSE 1

using namespace std;
void Filosofo(int id);
void Tenedor(int id);
void Camarero();

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

void retraso_aleatorio()
{
    static bool primera = true;

    // inicializar generador de números aleatorios (la primera vez)
    if (primera)
    {
        srand(time(NULL));
        primera = false;
    }
    // retraso aleatorio, de entre una y dos décimas de segundo
    usleep(1000L * (100L + rand() % 100L));
}

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

```
36
37 int main(int argc, char **argv)
38 {
39     int rank, size;
40
41     MPI_Init(&argc, &argv);
42     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
43     MPI_Comm_size(MPI_COMM_WORLD, &size);
44
45     if (size != NUM_PROCESOS)
46     {
47         if (rank == 0)
48             cout << "El numero de procesos debe ser " << NUM_PROCESOS <<
                    endl << flush;
49         MPI_Finalize();
50         return 0;
51     }
52
53     if (rank == RANK_CAMARERO)
54         Camarero(); // el 10 es el camarero
55     else if ((rank % 2) == 0)
56         Filosofo(rank); // los pares (0,2,4,6,8) son Filosofos
57     else
58         Tenedor(rank); // los impares (1,3,5,7,9) son Tenedores
59
60     MPI_Finalize();
61     return 0;
62 }
63
64 // ----------------------------------------------------------------------
65
66 void Filosofo(int id)
67 {
68     const int
69         primero = (id + 1) % (NUM_PROCESOS - 1),
70         segundo = (id + (NUM_PROCESOS - 2)) % (NUM_PROCESOS - 1);
71
72     while (true)
73     {
74         // solicita sentarse
75         cout << "Filosofo " << id << " solicita sentarse." << endl << flush
                ;
76         MPI_Ssend(&id, 1, MPI_INT, RANK_CAMARERO, TAG_SENTARSE,
                MPI_COMM_WORLD);
77
78         // solicita y coge primer tenedor
79         cout << "Filosofo " << id << " solicita primer tenedor: " <<
                primero << endl << flush;
80         MPI_Ssend(&id, 1, MPI_INT, primero, primero, MPI_COMM_WORLD);
81         cout << "Filosofo " << id << " coge primer tenedor: " << primero <<
                 endl << flush;
82
83         // solicita coge segundo tenedor
84         cout << "Filosofo " << id << " coge segundo tenedor: " << segundo
                << endl << flush;
85         MPI_Ssend(&id, 1, MPI_INT, segundo, MPI_COMM_WORLD);
```

```
86          cout << "Filosofo " << id << " coge segundo tenedor: " << segundo
                << endl << flush;
87
88          // come
89          cout << "Filosofo " << id << " COMIENDO" << endl << flush;
90          retraso_aleatorio();
91
92          // suelta el segundo tenedor
93          cout << "Filosofo " << id << " suelta tenedor: " << segundo << endl
                 << flush;
94          MPI_Ssend(&id, 1, MPI_INT, primero, primero, MPI_COMM_WORLD);
95
96          // suelta el primer tenedor
97          cout << "Filosofo " << id << " suelta tenedor: " << primero << endl
                 << flush;
98          MPI_Ssend(&id, 1, MPI_INT, segundo, segundo, MPI_COMM_WORLD);
99
100         // solicita levantarse
101         cout << "Filosofo " << id << " solicita levantarse." << endl <<
                flush;
102         MPI_Ssend(&id, 1, MPI_INT, RANK_CAMARERO, TAG_LEVANTARSE,
                MPI_COMM_WORLD);
103
104         // piensa
105         cout << "Filosofo " << id << " PENSANDO" << endl << flush;
106         retraso_aleatorio();
107     }
108 }
109
110 // -----------------------------------------------------------------------
111
112 void Tenedor(int id)
113 {
114     int valor;
115     MPI_Status status;
116
117     while (true)
118     {
119         // espera un mensaje desde cualquier filosofo vecino
120         cout << "Tenedor " << id << " espera petición." << endl << flush;
121         MPI_Recv(&valor, 1, MPI_INT, MPI_ANY_SOURCE, id, MPI_COMM_WORLD, &
                status);
122         cout << "Tenedor " << id << " recibe petición de " << status.
                MPI_SOURCE << endl << flush;
123
124         // espera a que el filosofo suelte el tenedor
125         MPI_Recv(&valor, 1, MPI_INT, status.MPI_SOURCE, id, MPI_COMM_WORLD,
                 &status);
126         cout << "Tenedor " << id << " recibe liberación de " << status.
                MPI_SOURCE << endl << flush;
127     }
128 }
129
130 // -----------------------------------------------------------------------
131
132 void Camarero()
```

```
133 {
134     int valor, num_sentados = 0, tag_aceptada;
135     MPI_Status status;
136 
137     while (true)
138     {
139         if (num_sentados < NUM_FILOSOFOS - 1)
140             tag_aceptada = MPI_ANY_TAG;
141         else
142             tag_aceptada = TAG_LEVANTARSE;
143 
144         // espera un mensaje desde cualquier filosofo con etiqueta
                aceptable
145         cout << "Camarero espera petición de sentarse/levantarse (sentados
                == " << num_sentados << ")" << endl << flush;
146 
147         MPI_Recv(&valor, 1, MPI_INT, MPI_ANY_SOURCE, tag_aceptada,
                MPI_COMM_WORLD, &status);
148         cout << "Camarero recibe petición de filosofo " << status.
                MPI_SOURCE << endl << flush;
149 
150         if (status.MPI_TAG == TAG_LEVANTARSE)
151             num_sentados--;
152         else
153             num_sentados++;
154     }
155 }
```

1.2. Solución Carlos Ureña 2014-2015

Ejercicio 1

- Errores:
  - En la línea 84 falta un argumento en la función `MPI_Ssend`. No está puesto el tag del mensaje (debería ser `"segundo"`).
  - En la línea 124 que los dos primeros argumentos están al revés.
- Error en el diseño: podemos ver que tras la ejecución del programa, los filósofos no se levantan de la mesa. El error se encuentra en la línea 101, en el paso del mensaje al camarero para solicitar levantarse, el tag del mensaje es el de sentarse (`TAG_SENTARSE`), en lugar del de levantarse (`TAG_LEVANTARSE`).
- Último apartado:
  - para dar servicio a 7 filósofos y 7 tenedores, se deben cambiar las constantes `NUM_FILOSOFOS` en la línea 7.
  - para ello debemos de cambiar la línea 54 poniendo que el rank debe de ser menor al número de filósofos.
  - además, debemos de cambiar la asignación de los tenedores en la línea 68 y 69, debido a que la presente no es coherente ni consistente desembocando en un error en el que varios filósofos quieren coger el mismo tenedor.

```cpp
#include <iostream>
#include <time.h>
#include <unistd.h>
#include <stdlib.h>
#include <mpi.h>

#define NUM_FILOSOFOS 7
#define NUM_PROCESOS (2*(NUM_FILOSOFOS)+1)
#define RANK_CAMARERO ((NUM_PROCESOS)-1)

#define TAG_SENTARSE 0
#define TAG_LEVANTARSE 1

using namespace std ;
void Filosofo( int id );
void Tenedor ( int id );
void Camarero() ;

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

void retraso_aleatorio()
{
    static bool primera = true ;

    // inicializar generador de numeros aleatorios (la primera vez)
    if ( primera )
    {  srand(time(NULL)) ;
       primera = false ;
    }
    // retraso aleatorio, de entre una y dos décimas de segundo
    usleep( 1000L * (100L + rand() % 100L ) );
}

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

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 ( size != NUM_PROCESOS )
    {
        if( rank == 0 )
        cout << "El numero de procesos debe ser " << NUM_PROCESOS << endl
           << flush;
        MPI_Finalize( );
        return 0;
    }

    if ( rank == RANK_CAMARERO )
        Camarero() ; // el 14 es el camarero
    else if (rank<NUM_FILOSOFOS)
```

```
55          Filosofo( rank );
56      else
57          Tenedor( rank );
58
59      MPI_Finalize( );
60      return 0;
61  }
62
63  // ---------------------------------------------------------------------
64
65  void Filosofo( int id )
66  {
67      const int
68          primero = NUM_FILOSOFOS + id ,
69          segundo = NUM_FILOSOFOS + ((id+1)%NUM_FILOSOFOS) ;
70
71      while ( true )
72      {
73          // solicita sentarse
74          cout << "Filosofo " << id << " solicita sentarse." << endl << flush
                ;
75          MPI_Ssend( &id, 1, MPI_INT, RANK_CAMARERO, TAG_SENTARSE,
                MPI_COMM_WORLD );
76
77          // solicita y coge primer tenedor
78          cout << "Filosofo " << id << " solicita primer tenedor: " <<
                primero << endl << flush;
79          MPI_Ssend( &id, 1, MPI_INT, primero, primero, MPI_COMM_WORLD );
80          cout << "Filosofo " << id << " coge primer tenedor: " << primero <<
                 endl << flush;
81
82          // solicita coge segundo tenedor
83          cout << "Filosofo " << id << " coge segundo tenedor: " << segundo
                << endl << flush;
84          MPI_Ssend( &id, 1, MPI_INT,segundo, segundo ,MPI_COMM_WORLD);
85          cout << "Filosofo " << id << " coge segundo tenedor: " << segundo
                << endl << flush;
86
87          // come
88          cout << "Filosofo " << id << " COMIENDO" << endl << flush;
89          retraso_aleatorio() ;
90
91          // suelta el segundo tenedor
92          cout << "Filosofo " << id << " suelta tenedor: " << segundo << endl
                 << flush;
93          MPI_Ssend( &id, 1, MPI_INT, primero, primero, MPI_COMM_WORLD );
94
95          // suelta el primer tenedor
96          cout << "Filosofo " << id << " suelta tenedor: " << primero << endl
                 << flush;
97          MPI_Ssend( &id, 1, MPI_INT, segundo, segundo, MPI_COMM_WORLD );
98
99          // solicita levantarse
100         cout << "Filosofo " << id << " solicita sentarse." << endl << flush
                ;
101         MPI_Ssend ( &id, 1, MPI_INT, RANK_CAMARERO, TAG_LEVANTARSE,
```

```
            MPI_COMM_WORLD );

        // piensa
        cout << "Filosofo " << id << " PENSANDO" << endl << flush ;
        retraso_aleatorio() ;
    }
}

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

void Tenedor( int id )
{
    int valor ;
    MPI_Status status;

    while ( true )
    {
        // espera un mensaje desde cualquier filosofo vecino
        cout <<"Tenedor " << id << " espera petición." << endl << flush;
        MPI_Recv( &valor, 1, MPI_INT, MPI_ANY_SOURCE, id, MPI_COMM_WORLD, &
            status );
        cout << "Tenedor " << id << " recibe petición de " << status.
            MPI_SOURCE << endl << flush ;

        // espera a que el filosofo suelte el tenedor
        MPI_Recv( &valor, 1, MPI_INT, status.MPI_SOURCE, id, MPI_COMM_WORLD
            , &status );
        cout<<"Tenedor " << id << " recibe liberación de " << status.
            MPI_SOURCE << endl << flush;
    }
}

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

void Camarero()
{
    int valor, num_sentados = 0, tag_aceptada ;
    MPI_Status status;

    while ( true )
    {
        if ( num_sentados < NUM_FILOSOFOS-1 )
            tag_aceptada = MPI_ANY_TAG ;
        else
            tag_aceptada = TAG_LEVANTARSE ;

        // espera un mensaje desde cualquier filosofo con etiqueta
            aceptable
        cout << "Camarero espera petición de sentarse/levantarse (sentados
            == " << num_sentados << ")" << endl << flush;

        MPI_Recv( &valor, 1, MPI_INT, MPI_ANY_SOURCE, tag_aceptada,
            MPI_COMM_WORLD, &status);
        cout << "Camarero recibe petición de filosofo " << status.
            MPI_SOURCE << endl << flush ;

```

```
149         if ( status.MPI_TAG == TAG_LEVANTARSE )
150            num_sentados -- ;
151         else
152             num_sentados ++ ;
153     }
154 }
```

La solución en código se encuentra aqui.

Ejercicio 2

En la resolución de este ejercicio usamos esta (plantilla)(pincha en plantilla para acceder a ella).

- Hemos añadido en la línea 58 en la función de producir valor esta línea **int valor = aleatorio<0, 9>();** para que produzca el valor aleatorio.
- En la función del productor debemos añadir **MPI_Ssend(&peticion, 1, MPI_INT, n_p, etiq_productor, MPI_COMM_WORLD);** para que el productor pida permiso para enviar el valor y añadimos en el **switch** que cuando sea el caso del productor reciba la petición de aviso: **MPI_Recv(&peticion_productor, 1, MPI_INT, MPI_ANY_SOURCE, etiq_productor, MPI_COMM_WORLD, &estado);**

Para la solución del apartado 2.2 revisa este código donde los comentarios **apartado2.2** reflejan los cambios que se han realizado.

```
1  #include <chrono> // duraciones (duration), unidades de tiempo
2  #include <cstdlib>
3  #include <iostream>
4  #include <mpi.h>
5  #include <random> // dispositivos, generadores y distribuciones aleatorias
6  #include <thread> // this_thread::sleep_for
7
8  using namespace std;
9  using namespace std::this_thread;
10 using namespace std::chrono;
11
12 #define TERM -1 // Valor especial para finalizar
13
14 const int
15     etiq_prod = 1, // Etiqueta para mensajes de productores
16     etiq_cons = 2, // Etiqueta para mensajes de consumidores
17     num_prod = 4, // Número de productores
18     num_cons = 5, // Número de consumidores
19     id_ini_productores = 0, // ID inicial de los productores
20     id_ini_consumidores = num_prod + 1, // ID inicial de los consumidores
21     id_buffer = num_prod, // ID del buffer
22     num_procesos_esperado = num_prod + num_cons + 1, // Número total de
           procesos esperados
23     num_items = 20, // Número total de ítems a producir/consumir
24     tam_vector = 10; // Tamaño del buffer
25
26 /**
27  * @brief Función que produce un número aleatorio entre dos valores
```

```
 *
 * @tparam min Valor mínimo. Constante conocida en tiempo de compilación
 * @tparam max Valor máximo. Constante conocida 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);
}
// ---------------------------------------------------------------------
/**
 * @brief Función que simula la producción de un valor entre 0 y 9
 *
 * @note Duerme un tiempo aleatorio entre 10 y 100 milisegundos
 *
 * @return int Valor producido
 */
int producir() {
    sleep_for(milliseconds(aleatorio<10, 100>())); // Simula tiempo de
       producción
    return aleatorio<0, 9>(); // Devuelve un valor aleatorio entre 0 y 9
}

/**
 * @brief Función que simula la producción de valores por un productor
 *
 * @param orden Orden del productor (global)
 */
void funcion_productor(int orden) {
    for (unsigned int i = 0; i < num_items / num_prod; i++) {
        // producir valor
        int valor_prod = producir();
        int rec;
        MPI_Status estado;
        // enviar valor
        cout << "Productor va a enviar valor " << valor_prod << endl <<
           flush;
        MPI_Ssend(&valor_prod, 1, MPI_INT, id_buffer, etiq_prod,
           MPI_COMM_WORLD); // Enviar valor al buffer
        MPI_Recv(&rec, 1, MPI_INT, id_buffer, 0, MPI_COMM_WORLD, &estado);
           // Recibir confirmación del buffer
        if (rec == TERM) { // Si recibe el valor especial, termina
            break;
        }
        MPI_Ssend(&valor_prod, 1, MPI_INT, id_buffer, 0, MPI_COMM_WORLD);
           // Enviar valor al buffer
    }
    cout << "Termina productor " << orden << endl; // Mensaje de finalizaci
       ón del productor
}
// ---------------------------------------------------------------------

/**
 * @brief Función que simula la consumición de un valor
 *
 * @param valor_cons Valor a consumir
```

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

/**
 * @brief Función que simula la consumición de valores por un consumidor
 *
 * @param orden Orden del consumidor (global)
 */
void funcion_consumidor(int orden) {
    int peticion, valor_rec = 1;
    MPI_Status estado;

    for (unsigned int i = 0; i < num_items / num_cons; i++) {
        MPI_Ssend(&peticion, 1, MPI_INT, id_buffer, etiq_cons, MPI_COMM_WORLD); // Enviar petición al buffer
        MPI_Recv(&valor_rec, 1, MPI_INT, id_buffer, 0, MPI_COMM_WORLD, &estado); // Recibir valor del buffer
        if (valor_rec == TERM) { // Si recibe el valor especial, termina
            break;
        }
        cout << "Consumidor ha recibido valor " << valor_rec << endl << flush;
        consumir(valor_rec); // Consumir el valor recibido
    }
    cout << "Termina consumidor " << orden << endl; // Mensaje de finalización del consumidor
}

/**
 * @brief Función que simula un buffer de tamaño tam_vector
 */
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
    MPI_Status estado;           // metadatos del mensaje recibido
    bool terminar = false;       // bandera para finalizar

    for (unsigned int i = 0; i < num_items * 2 && !terminar; i++) {
        // 1. determinar si puede enviar solo prod., solo cons, o todos
        int etiq;

        if (num_celdas_ocupadas == 0)              // si buffer vacío
            // solo prods
            etiq = etiq_prod;
        else if (num_celdas_ocupadas == tam_vector) // si buffer lleno
            // solo cons
```

```
128             etiq = etiq_cons;
129         else                                    // si no vacío ni lleno
130             // cualquiera
131             etiq = MPI_ANY_TAG;
132 
133         // 2. recibir un mensaje del emisor o emisores aceptables
134 
135         MPI_Recv(&valor, 1, MPI_INT, MPI_ANY_SOURCE, etiq, MPI_COMM_WORLD,
136                  &estado); // Recibir mensaje del productor o consumidor
137 
138         // 3. procesar el mensaje recibido
139 
140         switch (estado.MPI_TAG) { // leer emisor del mensaje en metadatos
141 
142             case etiq_prod: // si ha sido el productor: insertar en buffer
143                 // Da permiso
144                 MPI_Ssend(&valor, 1, MPI_INT, estado.MPI_SOURCE, 0,
                        MPI_COMM_WORLD); // Enviar confirmación al productor
145                 MPI_Recv(&valor, 1, MPI_INT, estado.MPI_SOURCE, 0,
                        MPI_COMM_WORLD, &estado); // Recibir valor del productor
146 
147                 // Dos valores iguales consecutivos recibidos
148                 if (valor == buffer[(primera_libre - 1 + tam_vector) %
                        tam_vector]) {
149                     cout << "Empezamos a terminar el programa" << endl;
150                     terminar = true;
151                     int aux;
152                     MPI_Request request;
153                     // terminar procesos productores
154                     for (int i = 0; i < id_buffer; i++) {
155                         MPI_Irecv(&aux, 1, MPI_INT, i, MPI_ANY_TAG,
                                MPI_COMM_WORLD, &request); // Recibir cualquier
                                mensaje pendiente
156                         MPI_Irecv(&aux, 1, MPI_INT, i, MPI_ANY_TAG,
                                MPI_COMM_WORLD, &request); // Recibir cualquier
                                mensaje pendiente
157                         aux = TERM;
158                         MPI_Ssend(&aux, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
                                // Enviar valor especial para finalizar
159                     }
160 
161                     // terminar procesos consumidores
162                     for (int i = id_buffer + 1; i < num_procesos_esperado;
                            i++) {
163                         MPI_Irecv(&aux, 1, MPI_INT, i, MPI_ANY_TAG,
                                MPI_COMM_WORLD, &request); // Recibir cualquier
                                mensaje pendiente
164                         aux = TERM;
165                         MPI_Ssend(&aux, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
                                // Enviar valor especial para finalizar
166                     }
167                 }
168                 if (!terminar) {
169                     buffer[primera_libre] = valor; // Insertar valor en el
                            buffer
170                     primera_libre = (primera_libre + 1) % tam_vector; //
```

```
                        Actualizar índice de primera celda libre
171                     num_celdas_ocupadas++; // Incrementar número de celdas
                        ocupadas
172                     cout << "Buffer ha recibido valor " << valor << endl;
173                 }
174                 break;
175
176             case etiq_cons: // si ha sido el consumidor: extraer y enviarle
177                 if (!terminar) {
178                     valor = buffer[primera_ocupada]; // Extraer valor del
                        buffer
179                     primera_ocupada = (primera_ocupada + 1) % tam_vector;
                        // Actualizar índice de primera celda ocupada
180                     num_celdas_ocupadas--; // Decrementar número de celdas
                        ocupadas
181                     cout << "Buffer va a enviar valor " << valor << endl;
182                     MPI_Ssend(&valor, 1, MPI_INT, estado.MPI_SOURCE, 0,
                        MPI_COMM_WORLD); // Enviar valor al consumidor
183                 }
184                 break;
185         }
186     }
187     cout << "Termina buffer" << endl; // Mensaje de finalización del buffer
188 }
189
190 // ----------------------------------------------------------------------
191
192 int main(int argc, char *argv[]) {
193     int id_propio, num_procesos_actual;
194
195     if (num_items % num_prod != 0 || num_items % num_cons != 0) {
196         cout << "Numero de productores y consumidores no valido" << endl;
197         exit(1);
198     }
199
200     // inicializar MPI, leer identif. de proceso y número de procesos
201     MPI_Init(&argc, &argv);
202     MPI_Comm_rank(MPI_COMM_WORLD, &id_propio);
203     MPI_Comm_size(MPI_COMM_WORLD, &num_procesos_actual);
204
205     if (num_procesos_esperado == num_procesos_actual) {
206         // ejecutar la operación apropiada a 'id_propio'
207         if (id_ini_productores <= id_propio && id_propio <
                id_ini_productores + num_prod)
208             funcion_productor(id_propio - id_ini_productores); // Ejecutar
                    función productor
209         else if (id_propio == id_buffer)
210             funcion_buffer(); // Ejecutar función buffer
211         else
212             funcion_consumidor(id_propio - id_ini_consumidores); //
                    Ejecutar función consumidor
213     } else {
214         if (id_propio == 0) // solo el primero escribe error, indep. del
                rol
215         {
216             cout << "el número de procesos esperados es:    "
```

```
217                     << num_procesos_esperado << endl
218                     << "el número de procesos en ejecución es: "
219                     << num_procesos_actual << endl
220                     << "(programa abortado)" << endl;
221             }
222         }
223
224         // al terminar el proceso, finalizar MPI
225         MPI_Finalize();
226         return 0;
227 }
```

Pincha aqui para acceder al fichero.

2 Examen de Mario Bros con Yoshi como NPC

2.1. Enunciado

Se pide resolver el siguiente ejercicio utilizando directivas MPI vistas en clase. El juego consiste en comer tantas tartas como sea posible. El que más kg de tartas haya comido, gana. Contaremos con las siguientes restricciones:

- Habrá solamente dos jugadores, Mario y Peach.
- Habrá un NPC, Yoshi, que será el árbitro.
- El proceso Yoshi se encargará:
  - Por un lado, de proveer tartas a cada equipo. Repondrá un máximo de x veces.
    - Cada vez que repone, decide si provee de 1 o 2 tartas a la vez. Creará un vector con 1 o 2 elementos.
    - Cada tarta tendrá un peso asociado que será también aleatorio entre 1 y 10 kg. Un vector [1,10] indicará el peso mínimo y máximo de las tartas.
    - Mostrará cuántas tartas va a reponer y a qué equipo.
  - Por otro lado, recibirá la suma de los kg que cada equipo se ha comido en cada tanda. Para cada tanda servida:
    - Comparará quién ha comido más kg de tarta en dicha tanda.
    - Sumará 1 punto al equipo que más kg haya comido.
  - Finalmente, comparará los puntos de ambos equipos y proclamará vencedor al que más kg de tarta comió.
- Los procesos Mario y Peach realizarán las siguientes operaciones:
  - Recibirán las tartas que se van a comer.
  - Mostrarán las tartas que han recibido y sus pesos.
  - Sumarán el peso de cada tarta y aleatoriamente se comerán una porción.

    - La cantidad que se han comido se la notificarán a Yoshi.

- Mientras haya envíos que hacer, se han de poder realizar, evitando que haya ningún tipo de bloqueo. Es decir, Yoshi podrá reponer tartas sin tener que sufrir ningún tipo de bloqueo por otras operaciones (a excepción de retrasos aleatorios).
- Se han de incluir retrasos aleatorios.

2.2. Solución

```
1  //resolucion
2  #include <iostream>
3  #include <string>
4  #include <random>
5  #include <chrono>
6  #include <thread>
7  #include <mpi.h>
8  using namespace std ;
9
10 const int    ID_YOSHI = 0,
11             ID_MARIO = 1,
12             ID_PEACH = 2,
13             num_jugadores = 2,
14             num_procesos_esperados = num_jugadores + 1;
15
16 const double TERMINAR = -1;
17
18 const int    tag_jugador = 1,
19             tag_yoshi = 2;
20
21
22 /**
23  * @brief Función que produce un número aleatorio real entre dos valores.
24  *          El número aleatorio se genera en el rango [min, max].
25  *
26  * @param min Valor mínimo. Incluido.
27  * @param max Valor máximo. Incluido.
28  *
29  * @return double Número aleatorio generado.
30  */
31 double aleatorio_real(int min, int max) {
32    static const int precision = 1e3;
33     static default_random_engine generador( (random_device())() );
34     static uniform_int_distribution<int> distribucion_uniforme( 0,
          precision ) ;
35    double valor_aleatorio = distribucion_uniforme( generador )/((double)
         precision);
36     return min + valor_aleatorio * (max - min);
37 }
38
39 /**
40  * @brief Función que produce un número aleatorio entero entre dos valores.
41  *          El número aleatorio se genera en el rango [min, max].
42  *
```

```
 * @param min Valor mínimo. Incluido.
 * @param max Valor máximo. Incluido.
 * @return int Número aleatorio generado.
 */
int aleatorio_entero(int min, int max) {
   return round(aleatorio_real(min, max));
}

/**
 * @brief Función que bloquea el hilo actual durante un número de
     milisegundos.
 *
 * @param num_milisegundos Número de milisegundos a bloquear.
 */
void bloquear(int num_milisegundos) {
   chrono::milliseconds duracion(num_milisegundos);
   this_thread::sleep_for(duracion);
}

/**
 * @brief Dado un id, devuelve el nombre del jugador/NPC correspondiente.
 *
 * @param id ID del jugador/NPC.
 * @return string Nombre del jugador/NPC.
 */
string id_a_nombre(int id) {
   switch (id) {
      case ID_YOSHI:
         return "Yoshi";
      case ID_MARIO:
         return "Mario";
      case ID_PEACH:
         return "Peach";
      default:
         return "Desconocido";
   }
}


/**
 * @brief Función que ejecuta la hebra de Yoshi.
 *
 */
void funcion_yoshi() {
   const int   num_reposiciones = 10,
               num_tartas_max_por_reposicion = 2,
               peso_minimo_tarta = 1,
               peso_maximo_tarta = 10;

   int puntos[num_jugadores] = {0};

   MPI_Status estado;

   // Cada ronda
   for (int i = 0; i < num_reposiciones; i++) {

```

```
98        // Enviamos las tartas a los jugadores
99        for (int id_jugador = 1; id_jugador <= num_jugadores; id_jugador++) {
100          // Número de tartas a enviar al jugador j
101          int num_tartas = aleatorio_entero(1, num_tartas_max_por_reposicion
                 );
102
103          // Pesos de las tartas a enviar
104          double * pesos_tartas = new double[num_tartas];
105          for (int k = 0; k < num_tartas; k++)
106             pesos_tartas[k] = aleatorio_real(peso_minimo_tarta,
                    peso_maximo_tarta);
107
108          // Producción de tartas
109          bloquear(aleatorio_entero(10, 200));
110          cout << "Yoshi: Número de tartas producidas para " << id_a_nombre(
                 id_jugador) << ": " << num_tartas << endl;
111
112          // Enviamos las tartas
113          MPI_Send(pesos_tartas, num_tartas, MPI_DOUBLE, id_jugador,
                 tag_yoshi, MPI_COMM_WORLD);
114
115          delete[] pesos_tartas;
116       }
117
118       // Esperamos a recibir los pesos comidos por los jugadores
119       double peso_comido_jugador;
120       double peso_comido_max = 0;
121       int id_ganador_ronda;
122       for (int j = 1; j <= num_jugadores; j++) {
123          MPI_Recv(&peso_comido_jugador, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
                 tag_jugador, MPI_COMM_WORLD, &estado);
124          if (peso_comido_jugador > peso_comido_max) {
125             peso_comido_max = peso_comido_jugador;
126             id_ganador_ronda = estado.MPI_SOURCE;
127          }
128       }
129
130       // Sumamos un punto al jugador que más haya comido. Si hay empate, al
              que haya comido antes
131       puntos[id_ganador_ronda - 1]++;
132
133       // Informamos de los puntos actuales
134       cout << "Yoshi: Punto para " << id_a_nombre(id_ganador_ronda) << endl
              ;
135       cout << "Yoshi: Puntos en la ronda " << i+1 << ": " << endl;
136       for (int j = 0; j < num_jugadores; j++)
137          cout << "\t- " << id_a_nombre(j+1) << ": " << puntos[j] << "
                 puntos." << endl;
138       cout << "--------------------------------" << endl;
139    }
140
141    // Finalizamos el juego
142    for (int j = 1; j <= num_jugadores; j++)
143       MPI_Send(&TERMINAR, 1, MPI_DOUBLE, j, tag_yoshi, MPI_COMM_WORLD);
144
145
```

```
146     // Informamos del ganador. Si hay empate, se queda con el que se
            apuntase antes
147     int id_ganador = 0;
148     for (int j = 0; j < num_jugadores; j++)
149        if (puntos[j] > puntos[id_ganador])
150           id_ganador = j;
151
152
153     cout << "Yoshi: ¡El ganador es " << id_a_nombre(id_ganador + 1) << "!"
            << endl;
154  }
155
156  /**
157   * @brief Función que ejecuta la hebra de Mario o Peach.
158   *
159   * @param id_jugador ID del jugador.
160   */
161  void funcion_player(int id_jugador) {
162
163     MPI_Status estado;
164     int num_tartas;
165     double peso_total, peso_comido;
166
167     while (true) {
168
169        // Esperamos a recibir el número de tartas
170        MPI_Probe(ID_YOSHI, tag_yoshi, MPI_COMM_WORLD, &estado);
171        MPI_Get_count(&estado, MPI_DOUBLE, &num_tartas);
172
173        // Una vez sabemos cuántas tartas hay, creamos un vector para
               almacenar sus pesos
174        double * pesos_tartas = new double[num_tartas];
175
176        // Recibimos las tartas
177        MPI_Recv(pesos_tartas, num_tartas, MPI_DOUBLE, estado.MPI_SOURCE,
               estado.MPI_TAG, MPI_COMM_WORLD, &estado);
178
179        // Comprobamos si se ha recibido el mensaje de terminar
180        if (num_tartas == 1 && pesos_tartas[0] == TERMINAR) {
181           cout << id_a_nombre(id_jugador) << ": ¡Termino!" << endl;
182           delete[] pesos_tartas;
183           break;
184        }
185
186        // Informamos de las tartas recibidas, y calculamos el peso total
187        peso_total = 0;
188        cout << id_a_nombre(id_jugador) << ": Número de tartas recibidas: "
               << num_tartas << endl;
189        for (int i = 0; i < num_tartas; i++) {
190           cout << "\t- Tarta " << i << ": " << pesos_tartas[i] << " kg."<<
                  endl;
191           peso_total += pesos_tartas[i];
192        }
193
194        // Calculamos el peso que vamos a comer
195        peso_comido = aleatorio_real(0, peso_total);
```

```
196
197         // Simulamos acción de comer
198         cout << id_a_nombre(id_jugador) << ": Voy a comerme " << peso_comido
                << " kg de tarta." << endl << endl;
199         bloquear(aleatorio_entero(10, 200));
200
201         // Informamos de que hemos terminado de comer
202         MPI_Send(&peso_comido, 1, MPI_DOUBLE, ID_YOSHI, tag_jugador,
                MPI_COMM_WORLD);
203         delete[] pesos_tartas;
204      }
205   }
206
207   // ---------------------------------------------------------------------
208   int main (int argc, char* argv[]) {
209      int id_propio, num_procesos_actual;
210
211      MPI_Init(&argc, &argv);
212      MPI_Comm_rank(MPI_COMM_WORLD, &id_propio);
213      MPI_Comm_size(MPI_COMM_WORLD, &num_procesos_actual);
214
215      if (num_procesos_actual != num_procesos_esperados) {
216         if (id_propio == 0) {
217            cerr << "El número de procesos esperados es " <<
                   num_procesos_esperados << " y el número de procesos actual es "
                    << num_procesos_actual << endl;
218         }
219         MPI_Finalize();
220         return 1;
221      }
222
223      if (id_propio == ID_YOSHI) {  // Se trata de Yoshi
224         funcion_yoshi();
225      } else { // Se trata de un jugador
226         funcion_player(id_propio);
227      }
228
229      MPI_Finalize();
230      return 0;
231   }
```

3 Examen de Consumidores (valores pares e impares)

3.1. Enunciado

Se pide modificar la solución al problema de múltiples productores y consumidores realizada en clase utilizando directivas MPI, de forma que:

- El número de productores sea 3 (y que pueda variar fácilmente).
- El número de consumidores sea 2, de forma que:
  - Habrá un consumidor que solo pueda consumir los valores pares.

- El otro consumidor solo podrá consumir los valores impares.

3.2. Solución

3.2.1. Partimos de la plantilla

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

//
// Sistemas concurrentes y Distribuidos.
// Práctica 3. Implementación de algoritmos distribuidos con MPI
//
// Archivo: prodcons2.cpp
// Implementación del problema del productor-consumidor con
// un proceso intermedio que gestiona un buffer finito y recibe peticiones
// en orden arbitrario
// (versión con un único productor y un único consumidor)
//
// Historial:
// Actualizado a C++11 en Septiembre de 2017
//
   ----------------------------------------------------------------------


#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
   num_items             = 20,
   tam_vector            = 10,
   n_p = 4,
   n_c = 5,
   num_procesos_esperado = n_p + n_c + 1,
   k = num_items/n_p;

//variables para el test de produccion de valores
int valores_producidos[n_p] = {0};
int valores_consumidos [n_c] = {0};

const int etiq_productor = 0, // etiqueta de productor
    etiq_consumidor = 1; // etiqueta de consumidor


//**********************************************************************
// plantilla de función para generar un entero aleatorio uniformemente
// distribuido entre dos valores enteros, ambos incluidos
```

```cpp
// (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 secuencia (1,2,3,....)
// y lleva espera aleatorio
int producir(int numero_productor) {
   static int contador = 0;
   sleep_for(milliseconds(aleatorio<10, 100>()));
   contador++;
   cout << "Productor " << numero_productor << " ha producido valor " << contador << endl << flush;
   valores_producidos[numero_productor]++;
   return contador;
}
// ----------------------------------------------------------------------

void funcion_productor(int numero_productor) {
   for (int i = 0; i < k; i++) {
      int valor_prod = producir(numero_productor);
      MPI_Ssend(&valor_prod, 1, MPI_INT, n_p, etiq_productor, MPI_COMM_WORLD);
      cout << "Productor " << numero_productor << " envió valor " << valor_prod << endl << flush;
   }
}
// ----------------------------------------------------------------------

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

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

   for( unsigned int i=0 ; i < num_items/n_c; i++ )
   {
      MPI_Ssend( &peticion,  1, MPI_INT, n_p, etiq_consumidor, MPI_COMM_WORLD); //n_p hace referencia al id de buffer
      MPI_Recv ( &valor_rec, 1, MPI_INT, n_p, etiq_consumidor, MPI_COMM_WORLD,&estado );
```

```
94        cout << "Consumidor con rol " << numero_consumidor << " ha recibido
              valor " << valor_rec << endl << flush ;
95        consumir( valor_rec );
96        valores_consumidos[numero_consumidor]++;
97     }
98  }
99  // -----------------------------------------------------------------------
100
101 void funcion_buffer()
102 {
103    int        buffer[tam_vector],      // buffer con celdas ocupadas y vací
               as
104               valor,                   // valor recibido o enviado
105               primera_libre       = 0, // índice de primera celda libre
106               primera_ocupada     = 0, // índice de primera celda ocupada
107               num_celdas_ocupadas = 0; // número de celdas ocupadas
108    MPI_Status estado ;                 // metadatos del mensaje recibido
109
110    int etiq_aceptable; // etiqueta aceptable, almacena el valor de la
           etiqueta del mensaje recibido
111
112    for( unsigned int i=0 ; i < num_items*2 ; i++ )
113    {
114       etiq_aceptable = (num_celdas_ocupadas == 0) ? etiq_productor
115                        : (num_celdas_ocupadas == tam_vector) ?
                              etiq_consumidor
116                        : MPI_ANY_TAG;
117
118       // 2. recibir un mensaje del emisor o emisores aceptables
119
120       MPI_Recv( &valor, 1, MPI_INT, MPI_ANY_SOURCE, etiq_aceptable,
              MPI_COMM_WORLD, &estado );
121
122       // 3. procesar el mensaje recibido
123
124       switch( estado.MPI_TAG ) // leer emisor del mensaje en metadatos
125       {
126          case etiq_productor: // si ha sido el productor: insertar en
                 buffer
127             buffer[primera_libre] = valor ;
128             primera_libre = (primera_libre+1) % tam_vector ;
129             num_celdas_ocupadas++ ;
130             cout << "Buffer ha recibido valor " << valor << endl ;
131             break;
132
133          case etiq_consumidor: // si ha sido el consumidor: extraer y
                 enviarle
134             valor = buffer[primera_ocupada] ;
135             primera_ocupada = (primera_ocupada+1) % tam_vector ;
136             num_celdas_ocupadas-- ;
137             cout << "Buffer va a enviar valor " << valor << endl ;
138             MPI_Ssend( &valor, 1, MPI_INT, estado.MPI_SOURCE ,
                    etiq_consumidor, MPI_COMM_WORLD);
139             break;
140       }
141    }
```

```
142 }
143
144 //Función extra para verificar que cada productor ha producido la cantidad
    correcta de ítems
145 void test_produccion() {
146
147    for (int i = 0; i < n_p; i++)
148    {
149       if(valores_producidos[i] == k){
150          cout << endl << "
              ------------------------------------------------------------" <<
              endl;
151          cout << "| Productor " << i << " ha producido la cantidad correcta
               de ítems |" << endl;
152          cout << "
              ------------------------------------------------------------" <<
              endl;
153       }
154    }
155
156    for (int i = 0; i < n_c; i++)
157    {
158       if (valores_consumidos[i] == num_items/n_c)
159       {
160          cout << endl << "
              ------------------------------------------------------------" <<
              endl;
161          cout << "| Consumidor " << i << " ha consumido la cantidad
              correcta de ítems |" << endl;
162          cout << "
              ------------------------------------------------------------" <<
              endl;
163       }
164    }
165
166
167
168
169 }
170
171
172 // ---------------------------------------------------------------------
173 int main( int argc, char *argv[] )
174 {
175    int id_propio, num_procesos_actual;
176
177    // inicializar MPI, leer identif. de proceso y número de procesos
178    MPI_Init( &argc, &argv );
179    MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
180    MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual );
181
182    if(num_procesos_esperado == num_procesos_actual) {
183       if(id_propio < n_p) {
184          funcion_productor(id_propio);
185
186       }
```

```
187       else if (id_propio == n_p) {
188          funcion_buffer();
189
190       }
191       else {
192          funcion_consumidor(id_propio - n_p - 1); // debemos de restarlo
               para que coincida con el numero del proceso consumidor
193       }
194    } else {
195       if ( id_propio == 0 ) // solo el primero escribe error, indep. del
           rol
196       { cout << "el número de procesos esperados es:    " <<
              num_procesos_esperado << endl
197             << "el número de procesos en ejecución es: " <<
                 num_procesos_actual << endl
198             << "(programa abortado)" << endl ;
199       }
200    }
201
202    // al terminar el proceso, finalizar MPI
203    MPI_Finalize( );
204
205    test_produccion();
206    return 0;
207 }
```

3.2.2. Solución

```
1  //
       ------------------------------------------------------------------------
2  //
3  // Sistemas concurrentes y Distribuidos.
4  // Práctica 3. Implementación de algoritmos distribuidos con MPI
5  //
6  // Archivo: prodcons2.cpp
7  // Implementación del problema del productor-consumidor con
8  // un proceso intermedio que gestiona un buffer finito y recibe peticiones
9  // en orden arbitrario
10 // (versión con un único productor y un único consumidor)
11 //
12 // Historial:
13 // Actualizado a C++11 en Septiembre de 2017
14 //
       ------------------------------------------------------------------------
15
16 #include <iostream>
17 #include <thread> // this_thread::sleep_for
18 #include <random> // dispositivos, generadores y distribuciones aleatorias
19 #include <chrono> // duraciones (duration), unidades de tiempo
20 #include <mpi.h>
21
22 using namespace std;
```

```cpp
using namespace std::this_thread ;
using namespace std::chrono ;

const int
   num_items             = 6, //cambiamos los datos para que sean
      divisibles entre productor y consumidor
   tam_vector            = 10,
   n_p = 3,
   n_c = 2,
   num_procesos_esperado = n_p + n_c + 1,
   k = num_items/n_p;

//variables para el test de produccion de valores
int valores_producidos[n_p] = {0};
int valores_consumidos [n_c] = {0};

const int etiq_productor = 0, // etiqueta de productor
    //etiq_consumidor = 1; // etiqueta de consumidor
    etiq_consumidor_par = 1, // etiqueta de consumidor par
      etiq_consumidor_impar = 2, // etiqueta de consumidor impar
      etiq_consumidor = 3; // etiqueta de consumidor


//**********************************************************************
// 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 secuencia (1,2,3,....)
// y lleva espera aleatorio
int producir(int numero_productor) {
   static int contador = 0;
   sleep_for(milliseconds(aleatorio<10, 100>()));
   contador++;
   cout << "Productor " << numero_productor << " ha producido valor " <<
      contador << endl << flush;
   valores_producidos[numero_productor]++;
   return contador;
}
// ---------------------------------------------------------------------

void funcion_productor(int numero_productor) {
   for (int i = 0; i < k; i++) {
      int valor_prod = producir(numero_productor);
      MPI_Ssend(&valor_prod, 1, MPI_INT, n_p, etiq_productor,
         MPI_COMM_WORLD);
```

```
74        cout << "Productor " << numero_productor << " envió valor " <<
              valor_prod << endl << flush;
75     }
76  }
77  // ---------------------------------------------------------------------
78
79  void consumir( int valor_cons )
80  {
81     // espera bloqueada
82     sleep_for( milliseconds( aleatorio<110,200>()) );
83     cout << "Consumidor ha consumido valor " << valor_cons << endl << flush
           ;
84  }
85  // ---------------------------------------------------------------------
86
87  void funcion_consumidor(int numero_consumidor)
88  {
89     int         peticion = 1,
90                 valor_rec;
91     MPI_Status  estado ;
92
93     for( unsigned int i=0 ; i < num_items/n_c; i++ )
94     {
95        MPI_Ssend( &peticion,  1, MPI_INT, n_p, etiq_consumidor,
              MPI_COMM_WORLD); //n_p hace referencia al id de buffer
96
97        MPI_Recv ( &valor_rec, 1, MPI_INT, n_p, MPI_ANY_TAG, MPI_COMM_WORLD,&
              estado );
98        if( estado.MPI_TAG == etiq_consumidor_par ){
99           cout << "Consumidor con rol " << numero_consumidor << " ha
                 recibido valor " << valor_rec << " que es par" << endl << flush
                  ;
100       }
101       else if( estado.MPI_TAG == etiq_consumidor_impar ){
102          cout << "Consumidor con rol " << numero_consumidor << " ha
                 recibido valor " << valor_rec << " que es impar" << endl <<
                 flush ;
103       }
104       else{
105          cout << "ERROR, HA RECIBIDO DEL PRODUCTOR, .... ABORTAR....";
106          exit(0);
107       }
108       //cout << "Consumidor con rol " << numero_consumidor << " ha recibido
               valor " << valor_rec << endl << flush ;
109       consumir( valor_rec );
110       valores_consumidos[numero_consumidor]++;
111    }
112 }
113 // ---------------------------------------------------------------------
114
115 void funcion_buffer()
116 {
117    int         buffer[tam_vector],      // buffer con celdas ocupadas y vací
           as
118                valor,                   // valor recibido o enviado
119                primera_libre       = 0, // índice de primera celda libre
```

```
120             primera_ocupada     = 0, // índice de primera celda ocupada
121             num_celdas_ocupadas = 0; // número de celdas ocupadas
122    MPI_Status estado ;                 // metadatos del mensaje recibido
123
124    int etiq_aceptable; // etiqueta aceptable, almacena el valor de la
           etiqueta del mensaje recibido
125
126    for( unsigned int i=0 ; i < num_items*2 ; i++ )
127    {
128       etiq_aceptable = (num_celdas_ocupadas == 0) ? etiq_productor
129                        : (num_celdas_ocupadas == tam_vector) ?
                              etiq_consumidor
130                        : MPI_ANY_TAG;
131
132       // 2. recibir un mensaje del emisor o emisores aceptables
133
134       MPI_Recv( &valor, 1, MPI_INT, MPI_ANY_SOURCE, etiq_aceptable,
              MPI_COMM_WORLD, &estado );
135
136
137       // 3. procesar el mensaje recibido
138
139       switch( estado.MPI_TAG ) // leer emisor del mensaje en metadatos
140       {
141          case etiq_productor: // si ha sido el productor: insertar en
                 buffer
142             buffer[primera_libre] = valor ;
143             primera_libre = (primera_libre+1) % tam_vector ;
144             num_celdas_ocupadas++ ;
145             cout << "Buffer ha recibido valor " << valor << endl ;
146             break;
147
148          case etiq_consumidor: // si ha sido el consumidor: extraer y
                 enviarle
149             valor = buffer[primera_ocupada] ;
150             primera_ocupada = (primera_ocupada+1) % tam_vector ;
151             num_celdas_ocupadas-- ;
152             cout << "Buffer va a enviar valor " << valor << endl ;
153             if(valor%2 == 0){
154                cout << "Buffer va a enviar valor " << valor << " a
                       consumidor par" << endl ;
155                MPI_Ssend( &valor, 1, MPI_INT, estado.MPI_SOURCE ,
                       etiq_consumidor_par, MPI_COMM_WORLD);
156             }
157             else{
158                cout << "Buffer va a enviar valor " << valor << " a
                       consumidor impar" << endl ;
159                MPI_Ssend( &valor, 1, MPI_INT, estado.MPI_SOURCE ,
                       etiq_consumidor_impar, MPI_COMM_WORLD);
160             }
161             break;
162       }
163    }
164 }
165
```

```
166 //Función extra para verificar que cada productor ha producido la cantidad
        correcta de ítems
167 void test_produccion() {
168
169     for (int i = 0; i < n_p; i++)
170     {
171         if(valores_producidos[i] == k){
172             cout << endl << "
                    ------------------------------------------------------------" <<
                    endl;
173             cout << "| Productor " << i << " ha producido la cantidad correcta
                     de ítems |" << endl;
174             cout << "
                    ------------------------------------------------------------" <<
                    endl;
175         }
176     }
177
178     for (int i = 0; i < n_c; i++)
179     {
180         if (valores_consumidos[i] == num_items/n_c)
181         {
182             cout << endl << "
                    ------------------------------------------------------------" <<
                    endl;
183             cout << "| Consumidor " << i << " ha consumido la cantidad
                    correcta de ítems |" << endl;
184             cout << "
                    ------------------------------------------------------------" <<
                    endl;
185         }
186     }
187
188
189
190
191 }
192
193
194 // ---------------------------------------------------------------------
195 int main( int argc, char *argv[] )
196 {
197     int id_propio, num_procesos_actual;
198
199     // inicializar MPI, leer identif. de proceso y número de procesos
200     MPI_Init( &argc, &argv );
201     MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
202     MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual );
203
204     if(num_procesos_esperado == num_procesos_actual) {
205         if(id_propio < n_p) {
206             funcion_productor(id_propio);
207
208         }
209         else if (id_propio == n_p) {
210             funcion_buffer();
```

```
211
212         }
213         else {
214             funcion_consumidor(id_propio - n_p - 1); // debemos de restarlo
                    para que coincida con el numero del proceso consumidor
215         }
216     } else {
217         if ( id_propio == 0 ) // solo el primero escribe error, indep. del
                rol
218         { cout << "el número de procesos esperados es:    " <<
                num_procesos_esperado << endl
219             << "el número de procesos en ejecución es: " <<
                    num_procesos_actual << endl
220             << "(programa abortado)" << endl ;
221         }
222     }
223
224     // al terminar el proceso, finalizar MPI
225     MPI_Finalize( );
226
227     test_produccion();
228     return 0;
229 }
```

4 Examen de modificar camarero y filósofos

4.1. Enunciado

El archivo plantilla proporcionado contiene una solución básica al problema de los filósofos con camarero. En esta solución, el camarero se encarga de controlar el acceso a la mesa, permitiendo que se sienten a la mesa un máximo de 4 filósofos. Hay 5 filósofos.

Modificar este archivo como sigue:

- El filósofo 4, cuyo rango es 8, tendrá un comportamiento algo diferente del resto. El filósofo 4 no pide ni suelta ningún tenedor ya que come con las manos. No obstante, este filósofo sí debe pedir permiso para sentarse y levantarse al camarero.
- Como el filósofo 4 no puede comer si no se siente muy acompañado, el camarero solo permite que se siente cuando hay 4 filósofos sentados en ella.
- El camarero además no permite levantarse a ninguno de ellos mientras está comiendo el filósofo 4, para que se sienta acompañado.
- Como ahora ya no hay interbloqueo, podría haber 5 sentados en la mesa. Han de usarse unas etiquetas especiales para el filósofo 4.

El camarero deberá, en un bucle infinito:

- Paso 1 (Usar Iprobe)

- IF (hay menos de 4 sentados)
  - Sondeo en un bucle de las peticiones de sentarse o levantarse de cualquier filósofo distinto al 4, hasta detectar una.
  - Recibe el primer mensaje detectado.
- ELSE IF (hay 4)
  - Sondeo en un bucle de las peticiones de levantarse de los filósofos distintos al 4 y de sentarse del 4, hasta detectar una.
  - Recibe el primer mensaje detectado.
- ELSE
  - Recibe la petición de levantarse del filósofo 4.

- Paso 2
  - Actualiza el número de sentados (en función de la etiqueta recibida).

4.2. Solución

4.2.1. Partimos de la plantilla

```
1
2  #include <mpi.h>
3  #include <thread> // this_thread::sleep_for
4  #include <random> // dispositivos, generadores y distribuciones aleatorias
5  #include <chrono> // duraciones (duration), unidades de tiempo
6  #include <iostream>
7
8  using namespace std;
9  using namespace std::this_thread ;
10 using namespace std::chrono ;
11
12 const int
13    num_filosofos = 5 ,              // número de filósofos
14    num_filo_ten  = 2*num_filosofos, // número de filósofos y tenedores
15    num_procesos  = num_filo_ten+1 ;   // número de procesos total
16 const int camarero= num_filo_ten;
17 const int tag_sentar=0, tag_levantar=1;
18
19
20 //**********************************************************************
21 // plantilla de función para generar un entero aleatorio uniformemente
22 //----------------------------------------------------------------------
23 template< int min, int max > int aleatorio()
24 {
25   static default_random_engine generador( (random_device())() );
26   static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
27   return distribucion_uniforme( generador );
28 }
29
30 // ----------------------------------------------------------------------
31
32 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.
  int  peticion=0, valor;

  while ( true )
  {

    cout <<"++++++++++++++++++++++++  Filosofo " <<id << " solicita
        SENTARSE." <<endl;
    MPI_Ssend(&peticion, 1, MPI_INT,camarero,tag_sentar,MPI_COMM_WORLD);


    cout <<"Filosofo " <<id << " solicita ten. izq." <<id_ten_izq <<endl;
    MPI_Ssend(&peticion, 1, MPI_INT,id_ten_izq,0,MPI_COMM_WORLD);

    cout <<"Filosofo " <<id <<" solicita ten. der." <<id_ten_der <<endl;
    MPI_Ssend(&peticion, 1, MPI_INT,id_ten_der,0,MPI_COMM_WORLD);

    cout <<endl<<"Filosofo " <<id <<" comienza a COMER" <<endl ;
    sleep_for( milliseconds( aleatorio<1000,1800>() ) );

    cout <<"Filosofo " <<id <<" suelta ten. izq. " <<id_ten_izq <<endl;
    MPI_Ssend(&peticion, 1, MPI_INT,id_ten_izq,0,MPI_COMM_WORLD);

    cout<< "Filosofo " <<id <<" suelta ten. der. " <<id_ten_der <<endl;
    MPI_Ssend(&peticion, 1, MPI_INT,id_ten_der,0,MPI_COMM_WORLD);

    cout <<"++++++++++++++++++++++++++  Filosofo " <<id << " solicita
        LEVANTARSE." <<endl;
    MPI_Ssend(&peticion, 1, MPI_INT,camarero,tag_levantar,MPI_COMM_WORLD);


    cout<<endl << "Filosofo " << id << " comienza a PENSAR" << endl;
    sleep_for( milliseconds( aleatorio<1000,1800>() ) );
 }
}


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

  while ( true )
  {
     MPI_Recv(&valor, 1, MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&estado);
     id_filosofo=estado.MPI_SOURCE;
     cout <<".....................................Ten. " <<id <<
                       " ha sido COGIDO por filo. " <<id_filosofo <<endl;

     MPI_Recv(&valor, 1, MPI_INT,id_filosofo,0,MPI_COMM_WORLD,&estado);
     cout <<".....................................Ten. "<< id<<
                       " ha sido SOLTADO por filo. " <<id_filosofo <<endl
                          ;
```

```
86    }
87  }
88
89  // ---------------------------------------------------------------------
90  void funcion_camarero( )
91  {
92    int sentados=0, id_filosofo, tag, valor;
93    MPI_Status estado ;
94
95    while ( true )
96    {
97       if (sentados<4)
98            MPI_Recv(&valor, 1, MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,
                 MPI_COMM_WORLD,&estado);
99        else
100           MPI_Recv(&valor, 1, MPI_INT,MPI_ANY_SOURCE,tag_levantar,
                 MPI_COMM_WORLD,&estado);
101
102       id_filosofo=estado.MPI_SOURCE; tag=estado.MPI_TAG;
103
104       if (tag==tag_sentar)
105          sentados++;
106       else
107          sentados--;
108      cout <<endl<<"########################################################
            CAMARERO DICE QUE HAY  " <<sentados <<
109                                              " FILOSOFOS EN LA MESA " <<endl
                                                    <<endl;
110
111   }
112 }
113 // ---------------------------------------------------------------------
114
115 int main( int argc, char** argv )
116 {
117    int id_propio, num_procesos_actual ;
118
119    MPI_Init( &argc, &argv );
120    MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
121    MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual );
122
123
124    if ( num_procesos == num_procesos_actual )
125    {
126     if (id_propio==camarero)
127       funcion_camarero();
128     else if ( id_propio % 2 == 0 )
129       funcion_filosofos( id_propio );
130     else
131       funcion_tenedores( id_propio );
132    }
133    else
134    {
135       if ( id_propio == 0 ) // solo el primero escribe error, indep. del
          rol
```

```
136       { cout << "el numero de procesos debería ser:    " << num_procesos <<
              endl
137              << "   y el numero de procesos es: " << num_procesos_actual <<
                  endl
138              << "(programa abortado)" << endl ;
139       }
140    }
141
142    MPI_Finalize( );
143    return 0;
144 }
145
146 // ---------------------------------------------------------------------
```

4.2.2. Solución

```
1
2  #include <mpi.h>
3  #include <thread> // this_thread::sleep_for
4  #include <random> // dispositivos, generadores y distribuciones aleatorias
5  #include <chrono> // duraciones (duration), unidades de tiempo
6  #include <iostream>
7
8  using namespace std;
9  using namespace std::this_thread ;
10 using namespace std::chrono ;
11
12 const int
13    num_filosofos = 5 ,              // número de filósofos
14    num_filo_ten  = 2*num_filosofos, // número de filósofos y tenedores
15    num_procesos  = num_filo_ten+1 ;   // número de procesos total
16 const int camarero= num_filo_ten;
17 const int tag_sentar=0, tag_levantar=1;
18 const int fil4 = 8;
19 const int tag_sentarsef4=2, tag_levantarsef4=3;
20
21
22 //**********************************************************************
23 // plantilla de función para generar un entero aleatorio uniformemente
24 //----------------------------------------------------------------------
25 template< int min, int max > int aleatorio()
26 {
27   static default_random_engine generador( (random_device())() );
28   static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
29   return distribucion_uniforme( generador );
30 }
31
32 // ---------------------------------------------------------------------
33
34 void funcion_filosofos( int id )
35 {
36   int id_ten_izq = (id+1)              % num_filo_ten, //id. tenedor izq.
37       id_ten_der = (id+num_filo_ten-1) % num_filo_ten; //id. tenedor der.
38   int  peticion=0, valor;
```

```
39
40  while ( true )
41  {
42
43    if(id!=fil4){
44      cout <<"++++++++++++++++++++++++  Filosofo " <<id << " solicita
            SENTARSE." <<endl;
45      MPI_Ssend(&peticion, 1, MPI_INT,camarero,tag_sentar,MPI_COMM_WORLD);
46
47
48      cout <<"Filosofo " <<id << " solicita ten. izq." <<id_ten_izq <<endl;
49      MPI_Ssend(&peticion, 1, MPI_INT,id_ten_izq,0,MPI_COMM_WORLD);
50
51      cout <<"Filosofo " <<id <<" solicita ten. der." <<id_ten_der <<endl;
52      MPI_Ssend(&peticion, 1, MPI_INT,id_ten_der,0,MPI_COMM_WORLD);
53
54      cout <<endl<<"Filosofo " <<id <<" comienza a COMER" <<endl ;
55      sleep_for( milliseconds( aleatorio<1000,1800>() ) );
56
57      cout <<"Filosofo " <<id <<" suelta ten. izq. " <<id_ten_izq <<endl;
58      MPI_Ssend(&peticion, 1, MPI_INT,id_ten_izq,0,MPI_COMM_WORLD);
59
60      cout<< "Filosofo " <<id <<" suelta ten. der. " <<id_ten_der <<endl;
61      MPI_Ssend(&peticion, 1, MPI_INT,id_ten_der,0,MPI_COMM_WORLD);
62
63      cout <<"++++++++++++++++++++++++++  Filosofo " <<id << " solicita
            LEVANTARSE." <<endl;
64      MPI_Ssend(&peticion, 1, MPI_INT,camarero,tag_levantar,MPI_COMM_WORLD)
            ;
65
66
67      cout<<endl << "Filosofo " << id << " comienza a PENSAR" << endl;
68      sleep_for( milliseconds( aleatorio<1000,1800>() ) );
69    }
70    else if (id == fil4){
71      //si es el filósofo 4, no pide ni suelta ningún tenedor, ya que come
            con las manos
72      cout <<"++++++++++++++++++++++++  Filosofo ESPECIAL" <<id << "
            solicita SENTARSE." <<endl;
73      MPI_Ssend(&peticion, 1, MPI_INT,camarero,tag_sentarsef4,
            MPI_COMM_WORLD); //usamos una etiqueta especial para que la
            reconozca el camarero
74
75      cout <<endl<<"Filosofo ESPECIAL" <<id <<" comienza a COMER" <<endl ;
76      sleep_for( milliseconds( aleatorio<1000,1800>() ) );
77
78      cout <<"++++++++++++++++++++++++++  Filosofo ESPECIAL" <<id << "
            solicita LEVANTARSE." <<endl;
79      MPI_Ssend(&peticion, 1, MPI_INT,camarero,tag_levantarsef4,
            MPI_COMM_WORLD); //usamos una etiqueta especial para que la
            reconozca el camarero
80
81      cout<<endl << "Filosofo ESPECIAL" << id << " comienza a PENSAR" <<
            endl;
82      sleep_for( milliseconds( aleatorio<1000,1800>() ) );
83
```

```
84      }
85
86   }
87  }
88
89
90  // -----------------------------------------------------------------------
91  void funcion_tenedores( int id )
92  {
93    int valor, id_filosofo ;  // valor recibido, identificador del filósofo
94    MPI_Status estado ;       // metadatos de las dos recepciones
95
96    while ( true )
97    {
98       MPI_Recv(&valor, 1, MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&estado);
99       id_filosofo=estado.MPI_SOURCE;
100      cout <<".....................................Ten. " <<id <<
101                        " ha sido COGIDO por filo. " <<id_filosofo <<endl;
102
103      MPI_Recv(&valor, 1, MPI_INT,id_filosofo,0,MPI_COMM_WORLD,&estado);
104      cout <<".....................................Ten. "<< id<<
105                        " ha sido SOLTADO por filo. " <<id_filosofo <<endl
                             ;
106   }
107 }
108
109 // -----------------------------------------------------------------------
110 void funcion_camarero( )
111 {
112   int sentados = 0, id_filosofo, tag, valor;
113   MPI_Status estado;
114
115   while ( true )
116   {
117     if (sentados < 4) // si hay menos de 4 filósofos sentados
118     {
119       int mensaje_recibido = 0;
120       while (!mensaje_recibido)
121       {
122         MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &
              mensaje_recibido, &estado);
123         if (mensaje_recibido)
124         {
125           MPI_Recv(&valor, 1, MPI_INT, estado.MPI_SOURCE, estado.MPI_TAG,
                MPI_COMM_WORLD, &estado);
126         }
127       }
128     }
129     else if (sentados == 4) // si hay 4 filósofos sentados
130     {
131       int mensaje_recibido = 0;
132       while (!mensaje_recibido)
133       {
134         MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &
              mensaje_recibido, &estado);
```

```
135         if (mensaje_recibido && (estado.MPI_TAG == tag_levantar || (estado.
                MPI_TAG == tag_sentarsef4 && estado.MPI_SOURCE == fil4)))
136         {
137           MPI_Recv(&valor, 1, MPI_INT, estado.MPI_SOURCE, estado.MPI_TAG,
                  MPI_COMM_WORLD, &estado);
138         }
139       }
140     }
141     else // si hay más de 4 filósofos sentados (solo puede ser el filósofo
            4)
142     {
143       MPI_Recv(&valor, 1, MPI_INT, fil4, tag_levantarsef4, MPI_COMM_WORLD,
             &estado);
144     }
145
146     id_filosofo = estado.MPI_SOURCE;
147     tag = estado.MPI_TAG;
148
149     if (tag == tag_sentar || tag == tag_sentarsef4)
150       sentados++;
151     else if (tag == tag_levantar || (tag == tag_levantarsef4))
152       sentados--;
153
154     cout << endl << "#########################################################
            CAMARERO DICE QUE HAY  " << sentados <<
155             " FILOSOFOS EN LA MESA " << endl << endl;
156   }
157 }
158 // ---------------------------------------------------------------------
159
160 int main( int argc, char** argv )
161 {
162    int id_propio, num_procesos_actual ;
163
164    MPI_Init( &argc, &argv );
165    MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
166    MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual );
167
168
169    if ( num_procesos == num_procesos_actual )
170    {
171     if (id_propio==camarero)
172       funcion_camarero();
173     else if ( id_propio % 2 == 0 )
174       funcion_filosofos( id_propio );
175     else
176       funcion_tenedores( id_propio );
177    }
178    else
179    {
180       if ( id_propio == 0 ) // solo el primero escribe error, indep. del
              rol
181       { cout << "el numero de procesos debería ser:    " << num_procesos <<
               endl
182              << "   y el numero de procesos es: " << num_procesos_actual <<
                    endl
```

```
183                 << "(programa abortado)" << endl ;
184         }
185     }
186
187     MPI_Finalize( );
188     return 0;
189 }
190
191 // ---------------------------------------------------------------------
```

5 Examen Realizado del grupo A3 de este curso

5.1. Enunciado

Se iba detallando los pasos que se debía de hacer, en este caso en la función del proceso encargado se debía de consultar el valor de cajas disponibles, en mi caso lo hice así por simplicidad.

5.2. Solución

```
1  //Ismael Sallami Moreno
2  //DNI: 20888108Z
3
4  //para la resolución de este problema vamos a usar la plantilla del
       problema de los filósofos con un camarero
5
6  #include <mpi.h>
7  #include <thread> // this_thread::sleep_for
8  #include <random> // dispositivos, generadores y distribuciones aleatorias
9  #include <chrono> // duraciones (duration), unidades de tiempo
10 #include <iostream>
11
12 using namespace std;
13 using namespace std::this_thread ;
14 using namespace std::chrono ;
15
16 const int
17     cajas = 3, // número de cajas registradoras
18     clientes = 10, // número de clientes
19     num_procesos = 11, // número de procesos
20     id_proceso_encargado = num_procesos-1; // id del proceso encargado
21
22 int peticion_pago = 1; //para solicitar pagar
23 int peticion_fin_pago=2; //para solicitar el fin de pago
24
25 bool caja_vacia[cajas] = {true,true,true};
26
27 int num_caja_vacia(){
28     int val = 0;
29     for(int i=0;i<cajas;i++){
30         if(caja_vacia[i]){
31             val++;
```

```
        }
    }
    return val;
}
int caja_libre(){
    int id_caja_libre=-1;
    for(int i=0;i<cajas;i++){
        if(caja_vacia[i]){
            id_caja_libre = i;
            break;
        }
    }
    return id_caja_libre;
}

//*************************************************************************
// 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_cliente(int id){
    MPI_Status estado;
    int caja_donde_pagar; // valor que recibe del proceso encargado
    int valor = id; // enviar el ID del cliente al encargado

    while(true) {
        sleep_for( milliseconds( aleatorio<10,300>() ) ); // espera
            aleatoria
        cout << "Cliente " << id << ": solicito que se me asigne caja" <<
            endl;
        MPI_Ssend(&valor, 1, MPI_INT, id_proceso_encargado, peticion_pago,
            MPI_COMM_WORLD);
        MPI_Recv(&caja_donde_pagar, 1, MPI_INT, id_proceso_encargado, 0,
            MPI_COMM_WORLD, &estado);
        cout << "Cliente " << id << ": comienzo pago en caja " <<
            caja_donde_pagar << endl;
        sleep_for( milliseconds( aleatorio<10,300>() ) ); // espera
            aleatoria
        cout << "Cliente " << id << ": termino pago en caja " <<
            caja_donde_pagar << endl;
        MPI_Ssend(&caja_donde_pagar, 1, MPI_INT, id_proceso_encargado,
            peticion_fin_pago, MPI_COMM_WORLD);
    }
}
```

```
78
79
80  void funcion_encargado(){
81      int valor, caja_a_liberar; // valor es el id del cliente o la caja que
            libera
82      MPI_Status estado;
83
84      while (true) {
85          MPI_Recv(&valor, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                MPI_COMM_WORLD, &estado); // recibe el mensaje
86
87          //en este caso se pedia que si habia cajas libres que la etiqueta
                fuese cualquiera, pero he decidio hacerlo de esta manera para
                hacerlo mas simple, aunque en el examen se pedía de esa manera.
88
89          if (estado.MPI_TAG == peticion_pago) {
90              int primera_caja_libre = caja_libre();
91              caja_vacia[primera_caja_libre] = false;
92              cout << "Encargado: asigna caja " << primera_caja_libre << " al
                    cliente " << valor << endl;
93              MPI_Ssend(&primera_caja_libre, 1, MPI_INT, estado.MPI_SOURCE,
                    0, MPI_COMM_WORLD);
94          }
95          else if (estado.MPI_TAG == peticion_fin_pago) {
96              caja_a_liberar = valor; // el valor recibido es la caja a
                    liberar
97              caja_vacia[caja_a_liberar] = true;
98              cout << "Encargado: liberada caja " << caja_a_liberar << " por
                    el cliente " << estado.MPI_SOURCE << endl;
99          }
100     }
101 }
102
103
104 int main( int argc, char** argv )
105 {
106     int id_propio, num_procesos_actual ;
107
108     MPI_Init( &argc, &argv );
109     MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
110     MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual );
111
112     if(num_procesos == num_procesos_actual){
113         if(id_propio < clientes){
114             funcion_cliente(id_propio);
115         }
116         else if(id_propio == id_proceso_encargado){
117             funcion_encargado();
118         }
119     }
120     else
121     {
122         if ( id_propio == 0 ) // solo el primero escribe error, indep. del
                rol
123         { cout << "el número de procesos esperados es:    " << num_procesos
                << endl
```

```
124                 << "el número de procesos en ejecución es: " <<
                        num_procesos_actual << endl
125                 << "(programa abortado)" << endl ;
126         }
127     }
128
129     MPI_Finalize( );
130     return 0;
131 }
```

6 Materiales

Los materiales se encuentran en mi página web (pincha aqui.)