Práctica 3

Salvador Romero Cortés

Productores-Consumidores

Los cambios realizados en el código fuente inicial son los necesarios para poder adaptarlo a varios consumidores y varios productores.

Para ello primero definimos constantes globales para la gestión de varios procesos, como son el número de consumidores, el número de productores, el id del buffer... Además de etiquetas que sirvan para diferenciar procesos consumidores de procesos productores.

```
const int np = 4;
const int nc = 5;
//0 <-> np-1 son productores
// np es el buffer
// np+1 <-> np+nc son consumidores

const int tag_prod = 0;
const int tag_cons = 1;

const int
   id_buffer = np,
   num_procesos_esperado = np+nc+1 ,
   num_items             = np*nc,
   tam_vector            = 10;
```

Además ahora en las funciones productoras tenemos un parámetro nuevo, orden, que identifica al productor en las salidas por pantalla y sirve para calcular el rango de datos que generarán los productores.

El siguiente cambio es en `funcion_productor` y en `producir`.

- `funcion_productor`

  El bucle `for` recorre desde 0 hasta el número de items dividido por el número de productores.

  ```
  void funcion_productor(int orden)
  {
     for ( int i=0; i < (num_items/np) ; i++ )
     {
        // producir valor
        int valor_prod = producir(orden);
        // enviar valor
        cout << "Productor va a enviar valor " << valor_prod << endl << flush;
        MPI_Ssend( &valor_prod, 1, MPI_INT, id_buffer, tag_prod, MPI_COMM_WORLD
  );
     }
  }
  ```

- `producir`

La variable contador ahora se inicializa a `orden*(num_items/np)`

```
int producir(int orden)
{
   static int contador = orden*(num_items/np) ;
   sleep_for( milliseconds( aleatorio<10,100>()) );
   contador++ ;
   cout << "Productor ha producido valor " << contador << endl << flush;
   return contador ;
}
```

Otro cambio es en la función `consumir` puesto que ahora hay varios procesos que consumen, se debe dividir el número de items entre estos. Consisten en un simple cambio en el bucle.

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

Como podemos ver, los Ssend y los Recv también han cambiado, como se mencionó al principio: ahora se usan las etiquetas para identificar los distintos grupos de procesos.

A continuación, en la función que ejecuta el buffer:

Aquí los cambios se basan principalmente en el cambio de ids por etiquetas:

```
/*
... declaraciones de variables
*/
for( unsigned int i=0 ; i < num_items*2 ; i++ )
   {
      // 1. determinar si puede enviar solo prod., solo cons, o todos
      if ( num_celdas_ocupadas == 0 ){               // si buffer vacío
         // id_emisor_aceptable = id_productor ;       // $~~~$ solo prod.
         tag_emisor_aceptable = tag_prod;
      }
      else if ( num_celdas_ocupadas == tam_vector ){ // si buffer lleno
         // id_emisor_aceptable = id_consumidor ;      // $~~~$ solo cons.
         tag_emisor_aceptable = tag_cons;
      }
      else{                                          // si no vacío ni lleno
        // id_emisor_aceptable = MPI_ANY_SOURCE ;     // $~~~$ cualquiera
         tag_emisor_aceptable = MPI_ANY_TAG;
      }
      // 2. recibir un mensaje del emisor o emisores aceptables

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

```
      // 3. procesar el mensaje recibido

      switch( estado.MPI_TAG ) // leer emisor del mensaje en metadatos
      {
         case tag_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 tag_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, tag_cons,
MPI_COMM_WORLD);
            break;
      }
   }
```

Finalmente, el main queda así:

```
int id_propio, num_procesos_actual;
   int orden = 0;

   // 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 < np )
         funcion_productor(id_propio);
      else if ( id_propio == id_buffer )
         funcion_buffer();
      else if (id_propio > id_buffer)
         funcion_consumidor();
   }
   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;
```

Captura de ejecución:

```
Productor ha producido valor 1
Productor va a enviar valor 1
Buffer ha recibido valor 1
Buffer va a enviar valor 1
Consumidor ha recibido valor 1
Productor ha producido valor 6
Productor va a enviar valor 6
Buffer ha recibido valor 6
Buffer va a enviar valor 6
Consumidor ha recibido valor 6
Productor ha producido valor 2
Productor va a enviar valor 2
Buffer ha recibido valor 2
Buffer va a enviar valor 2
Consumidor ha recibido valor 2
Productor ha producido valor 16
Productor va a enviar valor 16
Buffer ha recibido valor 16
Buffer va a enviar valor 16
Consumidor ha recibido valor 16
Productor ha producido valor 7
Productor va a enviar valor 7
Buffer ha recibido valor 7
Buffer va a enviar valor 7
Productor ha producido valor 11
Productor va a enviar valor 11
Buffer ha recibido valor 11
Consumidor ha recibido valor 7
Buffer ha recibido valor 17
Productor ha producido valor 17
Productor va a enviar valor 17
Productor ha producido valor 3
Productor va a enviar valor 3
Buffer ha recibido valor 3
Buffer ha recibido valor 12
Productor ha producido valor 12
Productor va a enviar valor 12
Buffer ha recibido valor 8
Productor ha producido valor 8
Productor va a enviar valor 8
Buffer va a enviar valor 11
Consumidor ha consumido valor 1
Consumidor ha recibido valor 11
Buffer ha recibido valor 18
Productor ha producido valor 18
Productor va a enviar valor 18
```

```
Buffer ha recibido valor 4
Productor ha producido valor 4
Productor va a enviar valor 4
Consumidor ha consumido valor 16
Consumidor ha recibido valor 17
```

Cena de los filósofos

El interbloqueo se produce porque todos los filósofos cogen los tenedores en el mismo orden. Esto hace que los filósofos cojan sólo 1 tenedor (todos empiezan cogiendo el tenedor izquierdo por lo que si dos filósofos contiguos cogen a la vez su tenedor izquierdo ya bloqueará al otro, esto se repite con todos los filósofos y se produce el interbloqueo) y por tanto se quedan bloqueados.

La solución consiste en hacer que uno de los filósofos coja los tenedores en el orden inverso. Esto rompe la simetría y se asegura de que los filósofos cojan dos tenedores y que el que queda esperando no tenga ninguno por lo que no puede encadenar bloqueos.

En mi caso, hago que el primer filósofo sea el que pida primero el tenedor derecho y luego el izquierdo, mientras que el resto piden primero el izquierdo y luego el derecho.

Código del primer filósofo:

```
void funcion_filosofo_asimetrico(int id)
{
   int id_ten_izq = (id + 1) % num_procesos,                //id. tenedor izq.
       id_ten_der = (id + num_procesos - 1) % num_procesos; //id. tenedor der.
   int valor;
   while (true)
   {
      cout << "Filósofo " << id << " solicita ten. der." << id_ten_der << endl;
      // ... solicitar tenedor derecho (completar)
      MPI_Ssend(&valor, 1, MPI_INT, id_ten_der, id_ten_der, MPI_COMM_WORLD);

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

      cout << "Filósofo " << id << " comienza a comer" << endl;
      sleep_for(milliseconds(aleatorio<50, 300>()));

      cout << "Filósofo " << id << " suelta ten. izq. " << id_ten_izq << endl;
      // ... soltar el tenedor derecho (completar)
      MPI_Ssend(&valor, 1, MPI_INT, id_ten_der, id_ten_der, MPI_COMM_WORLD);
      cout << "Filosofo " << id << " comienza a pensar" << endl;
      // ... soltar el tenedor izquierdo (completar)
      MPI_Ssend(&valor, 1, MPI_INT, id_ten_izq, id_ten_izq, MPI_COMM_WORLD);
      cout << "Filósofo " << id << " suelta ten. der. " << id_ten_der << endl;
      sleep_for(milliseconds(aleatorio<50, 300>()));
   }
}
```

Función del resto de filósofos:

```
void funcion_filosofos(int id)
{
   int id_ten_izq = (id + 1) % num_procesos,                //id. tenedor izq.
```

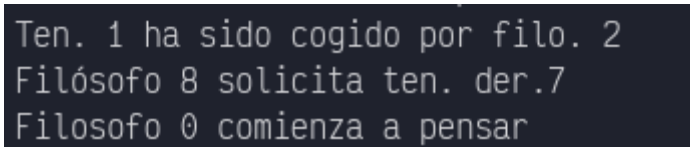
```
        id_ten_der = (id + num_procesos - 1) % num_procesos; //id. tenedor der.
    int valor;
    while (true)
    {
       cout << "Filósofo " << id << " solicita ten. izq." << id_ten_izq << endl;
       // ... solicitar tenedor izquierdo (completar)
       MPI_Ssend(&valor, 1, MPI_INT, id_ten_izq, id_ten_izq, 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, id_ten_der, MPI_COMM_WORLD);
       cout << "Filósofo " << id << " comienza a comer" << endl;
       sleep_for(milliseconds(aleatorio<50, 300>()));

       cout << "Filósofo " << id << " suelta ten. izq. " << id_ten_izq << endl;
       // ... soltar el tenedor izquierdo (completar)
       MPI_Ssend(&valor, 1, MPI_INT, id_ten_izq, id_ten_izq, 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, id_ten_der, MPI_COMM_WORLD);
       cout << "Filosofo " << id << " comienza a pensar" << endl;
       sleep_for(milliseconds(aleatorio<50, 300>()));
    }
}
```

Captura de ejecución:

```
Filósofo 8 solicita ten. izq.9
Filósofo 2 solicita ten. izq.3
Filósofo 4 solicita ten. izq.5
Filósofo 0 solicita ten. der.9
Filósofo 8 solicita ten. der.7
Filósofo 8 comienza a comer
Ten. 3 ha sido cogido por filo. 2
Filósofo 2 solicita ten. der.1
Filósofo 2 comienza a comer
Ten. 5 ha sido cogido por filo. 4
Ten. 9 ha sido cogido por filo. 8
Filósofo 4 solicita ten. der.3
Ten. 1 ha sido cogido por filo. 2
Ten. 7 ha sido cogido por filo. 8
Filósofo 6 solicita ten. izq.7
Filósofo 2 suelta ten. izq. 3
Filósofo 2 suelta ten. der. 1
Ten. 1 ha sido liberado por filo. 2
Filosofo 2 comienza a pensar
Ten. 3 ha sido liberado por filo. 2
Ten. 3 ha sido cogido por filo. 4
Filósofo 4 comienza a comer
Filósofo 8 suelta ten. izq. 9
Filósofo 8 suelta ten. der. 7
Filosofo 8 comienza a pensar
Ten. 9 ha sido liberado por filo. 8
Ten. 9 ha sido cogido por filo. 0
Filósofo 0 solicita ten. izq.1
Filósofo 6 solicita ten. der.5
Ten. 7 ha sido liberado por filo. 8
Ten. 7 ha sido cogido por filo. 6
Filósofo 0 comienza a comer
Ten. 1 ha sido cogido por filo. 0
Filósofo 2 solicita ten. izq.3
Ten. 5 ha sido liberado por filo. 4
Filósofo 2 solicita ten. der.1
Filósofo 4 suelta ten. izq. 5
Filósofo 4 suelta ten. der. 3
Filosofo 4 comienza a pensar
Filósofo 6 comienza a comer
Ten. 5 ha sido cogido por filo. 6
Ten. 3 ha sido liberado por filo. 4
Ten. 3 ha sido cogido por filo. 2
Filósofo 8 solicita ten. izq.9
Filósofo 0 suelta ten. izq. 1
Ten. 1 ha sido liberado por filo. 0
```



Cena de filósofos con camarero

Para este ejercicio surge un nuevo tipo de proceso, el camarero. Este se encarga de sentar y levantar a los filósofos. En caso de que haya 4 filósofos sentados no se permitirá ninguno más. Los filósofos se pueden levantar tras acabar de comer sin problema.

Surge una nueva función: `funcion_camarero`:

```
void funcion_camarero()
{
    int num_filosofos_sentados = 0, valor;
    int flag;
    MPI_Status estado;
    int id_esperando;
    int tag_aceptable;
    while (true){

        if (num_filosofos_sentados < 4){
            tag_aceptable = MPI_ANY_TAG;
        } else
        {
            tag_aceptable = etiq_levantarse;
        }

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

        switch(estado.MPI_TAG){
            case etiq_sentarse:
            cout << "Camarero sienta al filósofo " << estado.MPI_SOURCE << endl;
            num_filosofos_sentados++;
            break;
            case etiq_levantarse:
            cout << "Camarero levanta al filósofo " << estado.MPI_SOURCE << endl;
            num_filosofos_sentados--;
            break;
        }
    }
}
```

En la que seguimos un esquema similar al usado en el buffer de productores-consumidores.

También cambia la función de los filófos y el main.

Cambios en los filósofos:

```
//nuevos ids de tenedores
int id_ten_izq = (id + 1) % (num_procesos-1), //id. tenedor izq.
id_ten_der = (id + num_procesos - 2) % (num_procesos-1);
//... código anterior al bucle while sigue igual
cout << "Filósofo " << id << " solicita sentarse" << endl;
MPI_Ssend(&valor, 1, MPI_INT, id_camarero, etiq_sentarse, MPI_COMM_WORLD);
cout << "Filósofo " << id << " se sienta" << endl;
//... resto del código desde que coge los tenedores hasta que los suelta se
mantiene igual
 cout << "Filósofo " << id << " se va a levantar" << endl;
MPI_Ssend(&valor,1, MPI_INT, id_camarero, etiq_levantarse, MPI_COMM_WORLD);
cout << "Filósofo " << id << " se levanta de la mesa" << endl;
//... resto de codigo sigue igual
```

En main cambia la asignación de las funciones a cada proceso:

```
if (id_propio == id_camarero)
    funcion_camarero();         // si es 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
```

Captura de ejecución:

```
Filósofo 2 solicita sentarse
Filósofo 8 solicita sentarse
Filósofo 0 solicita sentarse
Filósofo 4 solicita sentarse
Filósofo 6 solicita sentarse
Filósofo 2 se sienta
Filósofo 2 solicita ten. izq.3
Filósofo 2 solicita ten. der.1
Ten. 3 ha sido cogido por filo. 2
Filósofo 4 se sienta
Filósofo 4 solicita ten. izq.5
Camarero sienta al filósofo 2
Camarero sienta al filósofo 4
Camarero sienta al filósofo 6
Camarero sienta al filósofo 8
Filósofo 2 comienza a comer
Filósofo 8 se sienta
Filósofo 8 solicita ten. izq.9
Ten. 1 ha sido cogido por filo. 2
Filósofo 6 se sienta
Filósofo 6 solicita ten. izq.7
Filósofo 4 solicita ten. der.3
Ten. 5 ha sido cogido por filo. 4
Ten. 7 ha sido cogido por filo. 6
Filósofo 6 solicita ten. der.5
Filósofo 8 solicita ten. der.7
Ten. 9 ha sido cogido por filo. 8
Filósofo 2 suelta ten. izq. 3
Filósofo 2 suelta ten. der. 1
Filósofo 2 se va a levantar
Filósofo 2 se levanta de la mesa
Filosofo 2 comienza a pensar
Ten. 3 ha sido liberado por filo. 2
Ten. 3 ha sido cogido por filo. 4
Camarero levanta al filósofo 2
Camarero sienta al filósofo 0
Filósofo 0 se sienta
Filósofo 0 solicita ten. izq.1
Filósofo 4 comienza a comer
Ten. 1 ha sido liberado por filo. 2
Filósofo 0 solicita ten. der.9
Ten. 1 ha sido cogido por filo. 0
Ten. 5 ha sido liberado por filo. 4
Ten. 5 ha sido cogido por filo. 6
Filósofo 4 suelta ten. izq. 5
Filósofo 4 suelta ten. der. 3
```

```
Filósofo 4 se va a levantar
Filósofo 4 se levanta de la mesa
Filosofo 4 comienza a pensar
```