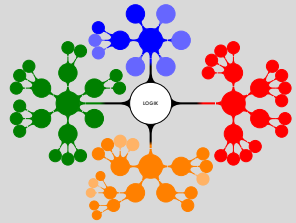
# Práctica 3

## Implementación de algoritmos distribuidos con MPI

Asignatura *Sistemas Concurrentes y Distribuidos*

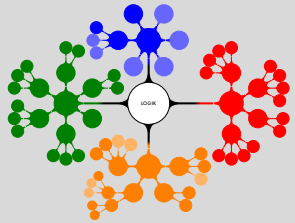
Fecha 19 noviembre 2019

Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada



# Objetivos

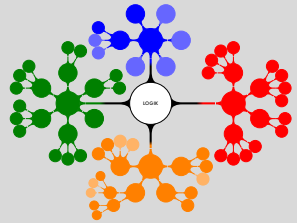
- Iniciarse en la programación de algoritmos distribuidos
- Conocer varios problemas sencillos de sincronización y su solución distribuida mediante el uso de la interfaz de paso de mensajes `MPI`:
  - Diseñar una solución distribuida al problema del productor: consumidor con buffer acotado, para varios productores y varios consumidores. El planteamiento del problema es similar al ya visto para múltiples hebras en memoria compartida
  - Diseñar diversas soluciones al problema de la cena de los filósofos
- Diseñar diversas soluciones al problema de la cena de los filósofos.



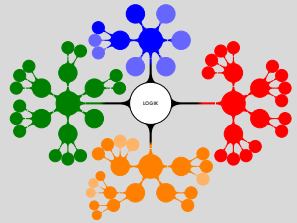
# Configuración de `openmpi`

Para instalar `Openmpi` correctamente:

1. Bajar el comprimido de:
   `https://www.open-mpi.org/software/ompi/v4.0/`
   (openmpi-4.0.2.tar.gz)
2. Hacer (en Linux): `tar -xvf openmpi-*` y cambiarse `cd openmpi-4.0.2`
3. `./configure - -prefix=/home/$USER/.openmpi`
4. `make install`
5. Hay que incluir en nuestro entorno el directorio de instalacion (˜ .openmpi) acabado en '/bin:'
   `export PATH="$PATH:/home/$USER/.openmpi/bin"`
6. Hay que incluir el directorio_de_instalacion_lib:
   `export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/home/$USER/.openmpi/lib/"`
7. Para hacerlos permanentes, teclear: `echo export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/home/$USER/.openmpi/lib/"» /home/$USER/.bashrc` y
   `echo export PATH="$PATH:/home/$USER/.openmpi/bin"»/home/$USER/.bashrc`



# Compilación de programas en `Openmpi`



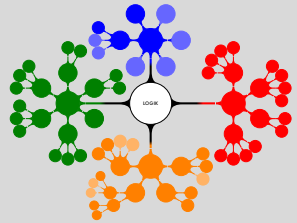
OpenMPI es una implementación portable y de código abierto del estándar MPI-2, llevada a cabo por una serie de instituciones de ámbito tanto académico y científico como industrial.
OpenMPI ofrece varios scripts necesarios para trabajar con programas aumentados con llamadas a funciones de MPI. Los más importantes son estos dos:

- `mpicxx`: para compilar y/o enlazar programas C++ que hagan uso de MPI
- `mpirun`: para ejecutar programas MPI. El programa `mpicxx` puede utilizarse con las mismas opciones que el compilador de C/C++ usual, p.e.:
  - `$ mpicxx -std=c++11 -c ejemplo.cpp`
  - `$ mpicxx -std=c++11 -o ejemplo ejemplo.o`

# Compilación de programas en `Openmpi` -II

La forma más usual de ejecutar un programa MPI es :

- `$ mpirun -np 4 ./ejemplo`
- El argumento `-np` sirve para indicar cuántos procesos ejecutarán el programa ejemplo. En este caso, se lanzarán cuatro procesos ejemplo
- Como no se indica la opción `-machinefile`, OpenMPI lanzará dichos 4 procesos en el mismo ordenador donde se ejecuta `mpirun`
- Con la opción `machinefile`, podemos realizar asociaciones de procesos a distintos ordenadores
- Si al ejecutar `mpirun` aparece el error: There are not enough slots available in the system . . .:
  - Crearse un archivo *hostfile* en el directorio de trabajo que contenga 1 sola línea con `localhost slots=40` (o un número mayor de slots necesarios para `-np`)
  - Cambiar la ejecución del programa a: `$ mpirun -hostfile hostfile -np 4 ./ejemplo`

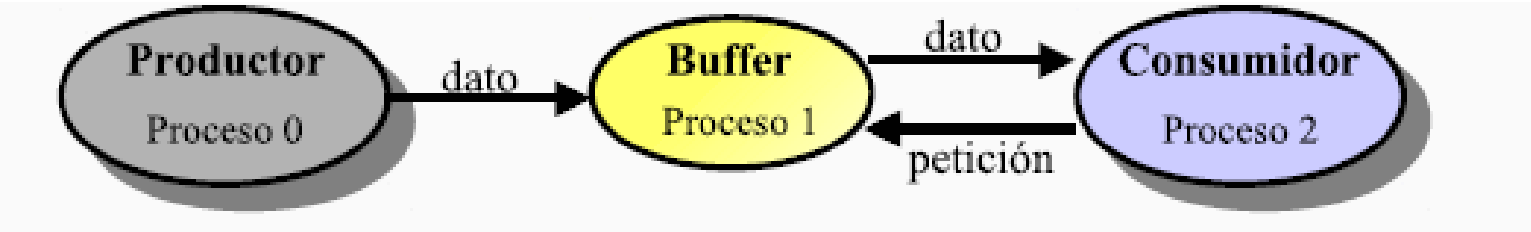


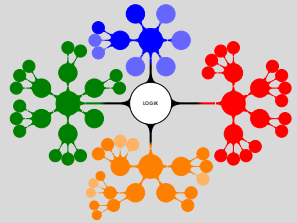
# Solución inicial

En la solución distribuida habrá (inicialmente) tres procesos:

- **Productor**: produce una secuencia de datos (números enteros, comenzando en 0), y los envía al proceso buffer
- **Buffer**: Recibe (de forma alterna) enteros del proceso productor y peticiones del consumidor. Responde al consumidor enviándole los enteros recibidos, en el mismo orden.
- **Consumidor**: realiza peticiones al proceso buffer, como respuesta recibe los enteros y los consume

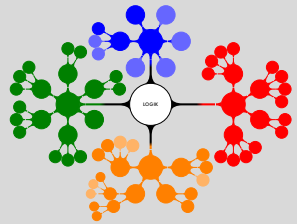
El esquema de comunicación entre estos procesos se muestra a continuación:



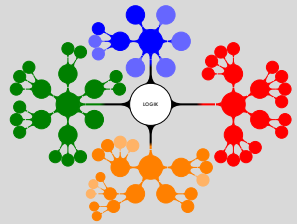
# Solución inicial: estructura del programa

```
1  include .../includesinclude <mpi.h> //includes de MPI
2  using ..... ; //using varios
3  //contantes: asignacion de identificadores a roles
4    const int id_productor = 0, //identificador del
5                                //proceso productor
6    id_buffer = 1,//identificador del proceso buffer
7    id_consumidor = 2,//identificador del
8                      //proceso consumidor
9    num_procesos_esperado = 3,//numero total de
10                             //procesos esperado
11   num_iteraciones = 20;  //numero de datos producidos
12                    //o consumidos; funciones auxiliares
13   int producir() { ... } //produce un valor
14                               //(usada por productor)
15   void consumir( int valor ){ ... }//consume un valor
16                               //(usada por consumidor)
17 // funciones ejecutadas por los procesos en cada rol:
18   void funcion_productor() { ... }//funcion ejecutada
19                              //por proceso productor
20   void funcion_consumidor() { ... }//funcion ejecutada
21                             //por proceso consumidor
22   void funcion_buffer() { ... }//funcion ejecutada por
23                               //proceso buffer
24 // funcion main (punto de entrada comun
25 // a todos los procesos del programa)
26 int main( int argc, char *argv[] ) { ... }
```



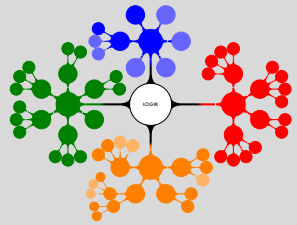
# Función `main()`

```
1  int main( int argc, char *argv[] ){
2  int id_propio, num_procesos_actual;//ident. propio,
3                     //numero de procesos del programa
4  MPI_Init( &argc, &argv );
5  MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
6  MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual );
7  if (num_procesos_esperado == num_procesos_actual){
8    if (id_propio == id_productor) //si mi identificador
9                                   //es el del productor
10     funcion_productor();//ejecutar funcion del productor
11     else if (id_propio == id_buffer)//si mi ↩
           identificador
12                                    //es el del buffer
13       funcion_buffer();//ejecutar funcion buffer
14       else //en otro caso, mi ident es consumidor
15        funcion_consumidor();//ejecutar funcion ↩
              consumidor
16 }
17 else if ( id_propio == 0 )//si hay error,
18                           //el proceso 0 informa:
19       cerr << "error: numero de procesos distinto del
20                esperado." << endl ;
21 MPI_Finalize( );
22 return 0;
23 }
```



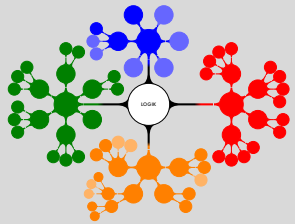
# Solución inicial: Productor y consumidor

```
1  void funcion_productor(){
2   for ( unsigned i = 0 ; i < num_items ; i++ ){
3    int valor_prod = producir();//producir (espera
4                                //bloqueado tiempo aleatorio↩
                                     )
5    cout << "Productor va a enviar valor "
6         << valor_prod << endl;
7    MPI_Ssend( &valor_prod, 1, MPI_INT, id_buffer,
8               0, MPI_COMM_WORLD );
9   }
10 }
11 void funcion_consumidor(){
12   int peticion, valor_rec = 1; MPI_Status estado;
13   for( unsigned i = 0 ; i < num_items; i++ ){
14    MPI_Ssend( &peticion, 1,MPI_INT, id_buffer,
15               0, MPI_COMM_WORLD);
16    MPI_Recv ( &valor_rec, 1,MPI_INT, id_buffer,
17               0, MPI_COMM_WORLD, &estado);
18    cout << "Consumidor ha recibido valor " <<
19         valor_rec << endl;
20    consumir( valor_rec );//consumir (espera
21                          //bloqueado tiempo aleatorio)
22   }
23 }
```



# Solución inicial: Proceso buffer

```
1  void funcion_buffer(){
2    int valor, peticion ;
3    MPI_Status estado ;
4    for ( unsigned int i = 0 ; i < num_items ; i++ ){
5    //recibir valor del productor
6    MPI_Recv( &valor, 1, MPI_INT, id_productor, 0,
7              MPI_COMM_WORLD, &estado);
8    cout << "Buffer ha recibido valor " << valor << endl ;
9  //recibir peticion de consumidor, enviarle el dato
10   MPI_Recv( &peticion, 1, MPI_INT, id_consumidor, 0,
11             MPI_COMM_WORLD, &estado);
12   cout << "Buffer va a enviar " << valor << endl;
13   MPI_Ssend( &valor, 1,MPI_INT, id_consumidor, 0,
14             MPI_COMM_WORLD);
15  }
16 }
```

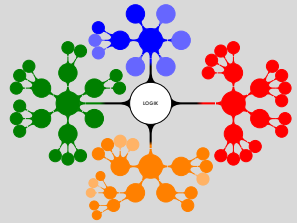


# Valoración de la solución inicial

Es correcta pero esta solución, sin embargo, fuerza una excesiva sincronización entre productor y consumidor
A largo plazo, el tiempo promedio empleado en producir será similar al empleado en consumir, sin embargo:

- En cada llamada puede haber diferencias arbitrarias entre ambos tiempos
- Frecuentemente la hebra productora o la hebra consumidora quedarán esperando un tiempo hasta que el buffer puedo procesar su envío o solicitud para recibir datos
- Si las hebras consumidora y productora se ejecutan en dos procesadores distintos y exclusivos (no compartidos con otra hebra), esos procesadores quedarán sin utilizar (es decir, “desocupados”) una fracción del tiempo total y el programa puede tarda más en acabar que una versión secuencial del mismo

Necesitamos algún mecanismo de reducción de las esperas

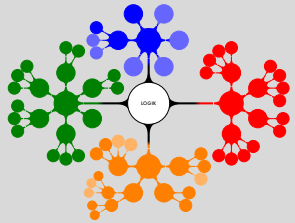


# Nueva solución con no-determinismo en el búfer

Para lograr nuestro objetivo, permitimos que el proceso buffer acomode diferencias temporales en la duración de producir y consumir:

- El proceso buffer puede guardar un vector de valores pendientes de consumir, en lugar de un único valor
- De esta forma: el productor puede producir varios valores seguidos (sin esperar al consumidor), y el consumidor también puede consumir (sin esperar al productor)
- Las esperas se reducen, las CPUs están menos tiempo desocupadas
- Por consiguiente, el tiempo total de ejecución del programa se reduce

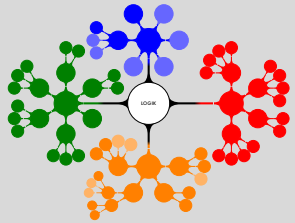
Para lograr esto, necesitamos que el proceso buffer pueda recibir una petición cuando haya valores pendientes de enviar, y a la vez pueda recibir un valor cuando haya huecos en el búfer



# Implementación de la *espera selectiva* en MPI

El comportamiento del buffer que queremos implementar es parecido al mecanismo denominado “espera selectiva”

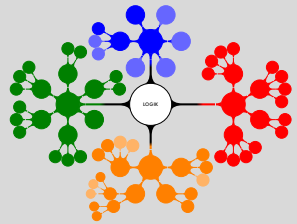
- En cada iteración, el búfer podrá aceptar un mensaje exclusivamente del productor (si el vector está vacío), exclusivamente del consumidor (si el vector está lleno), o de ambos (si no está vacío ni lleno)
- MPI permite implementar este comportamiento usando para ello la posibilidad de especificar, en cada operación de recepción, un emisor concreto o cualquier emisor (igualmente con las etiquetas)
- Por tanto, el buffer aceptará, en función del estado del vector que representa al búfer, un mensaje solo del productor, solo del consumidor, o de cualquier emisor (es decir, de ambos)



# Estructura del proceso búfer

El proceso buffer tiene esta estructura (archivo `prodcons2.cpp`)

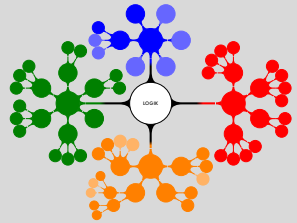
```
1  void funcion_buffer(){
2    int buffer[tam_vector], // buffer con celdas ocupadas
3                            // y vacias
4    valor,                    // valor recibido o enviado
5    primera_libre= 0,     // indice de primera celda libre
6    primera_ocupada= 0,// indice de primera celda ocupada
7    num_celdas_ocupadas= 0, //  numero de celdas ocupadas
8    id_emisor_aceptable;   // identificador de emisor
9                           // aceptable
10   MPI_Status estado;//metadatos del mensaje recibido
11   for( unsigned int i=0 ; i < num_items*2 ; i++ )
12   {
13   // 1. determinar si puede enviar solo productor,
14   // solo el consumidor, o ambos pueden enviar
15 ....
16   // 2. recibir un mensaje del emisor o
17   //    emisores aceptables
18 .....
19   // 3. procesar el mensaje recibido
20 .....
21 }
```



# Recepción de un mensaje

En el cuerpo del bucle, en primer lugar se calcula (en `id_emisor_aceptable`) de que proceso o procesos podemos aceptar un mensaje. Después, lo recibimos:

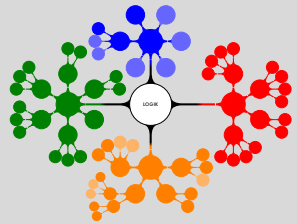
```
1  // 1.Determinar si puede enviar solo el productor,
2  // solo el consumidor, o de ambos pueden enviar
3  if (num_celdas_ocupadas == 0) // si buffer vacio
4    id_emisor_aceptable= id_productor; // solo productor
5  else if (num_celdas_ocupadas == tam_vector) // si buffer
6                                 // esta lleno
7    id_emisor_aceptable= id_consumidor; //solo consumidor
8    else//si el bufer no esta vacio ni lleno
9      id_emisor_aceptable= MPI_ANY_SOURCE;//cualquiera
10 //2.Recibir un mensaje del emisor o emisores aceptables:
11     MPI_Recv( &valor, 1, MPI_INT, id_emisor_aceptable,
12               0, MPI_COMM_WORLD,&estado );
```



# Procesado del mensaje recibido

Una vez recibido el mensaje, el tercer paso es actualizar el buffer en función de qué proceso haya sido el que lo ha enviado:

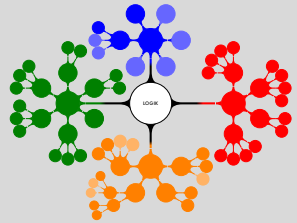
```
1  // 3. procesar el mensaje recibido
2  switch( estado.MPI_SOURCE ){ //leer emisor del mensaje
3                               //en metadatos
4    case id_productor: //si ha sido el productor:
5                       //insertarlo en el bufer
6     buffer[primera_libre]= valor;
7     primera_libre= (primera_libre+1) % tam_vector;
8     num_celdas_ocupadas++ ;
9     cout<<"Buffer ha recibido valor "<<valor<<endl;
10    break;
11   case id_consumidor://si ha sido el consumidor:
12                      //extraer valor y enviarselo
13    valor= buffer[primera_ocupada];
14    primera_ocupada= (primera_ocupada+1) % tam_vector ;
15    num_celdas_ocupadas-- ;
16    cout<<"Buffer va a enviar valor "<<valor<<endl ;
17    MPI_Ssend( &valor, 1, MPI_INT, id_consumidor, 0,
18               MPI_COMM_WORLD);
19    break;
```



# Ejercicio propuesto: múltiples consumidores y productores

Extenderemos el programa anterior (para 1 productor y 1 consumidor) a múltiples productores y consumidores:

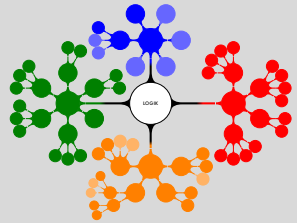
- Habrá `np = 4` procesos productores y `nc = 5` procesos consumidores
- Sigue habiendo un único proceso buffer
- El número `m` total de items a producir o consumir (constante `num_items`) debe ser múltiplo de `np` y múltiplo de `nc`
- Los procesos con identificador entre `0` y `np - 1` son productores
- El proceso con identificador `np` es el buffer
- Los procesos con identificdor entre `np + 1` y `np + nc` son consumidores
- Declarar dos constantes enteras con el núm. de prods. (`np`) y el núm. de consum. (`nc`), asegurarse que el programa es correcto aunque se usen otros valores distintos de 4 y 5 para `np` y `nc`



# Número de orden de los procesos y producción de valores

Puesto que ahora tenemos múltiples productores y consumidores:

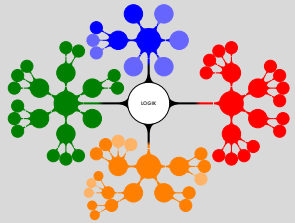
- La función de los productores y la de los consumidores reciben como parámetro el número de orden del productor o del consumidor, respectivamente (esos números son los números de orden en cada rol, comenzando en 0, no son los identificadores de proceso)
- Los números de orden deben calcularse en `main`
- La función de producir dato recibe como parámetro el número de orden del productor que la invoca. Esto permite que los productores usen cada uno su contador (variable contador de `producir_dato`) para producir un rango distinto de valores: el productor con número de orden `i` producirá los valores entre `ik` y `ik + k - 1` (ambos incluidos), donde `k` es el número de valores producidos por cada productor (es decir `k = m/np`)



# Diseño de la solución con etiquetas

Para solucionar el problema con múltiples productores/consumidores:

- Según el estado del buffer, debemos de aceptar un mensaje de cualquier productor, de cualquier consumidor, o de cualquier proceso
- No es posible usar exactamente la misma estrategia que antes: con MPI no es posible restringir el emisor aceptable a cualquiera de un subconjunto de procesos dentro de un comunicador (o aceptamos de un proceso concreto o aceptamos de todos los del comunicador)
- El problema se puede solucionar usando múltiples comunicadores, pero no hemos visto como definirlos
- También se puede solucionar usando dos etiquetas distintas para diferenciar los mensajes de los productores y los consumidores



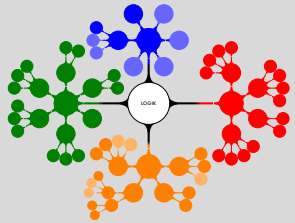
# Actividad para subir a `prado.ugr.es`

Partiendo de `prodcons2.cpp`, crea un nuevo archivo (llamado `prodcons2-mu.cpp`) con tu solución al problema descrito, ahora para múltiples productores y consumidores:

- Diseña una solución basada en el uso de etiquetas, que por lo demás es similar a la que ya hemos programado
- Define constantes enteras para las etiquetas: el programa será mucho más legible. Estas constantes deben tener nombres que comienzen con `etiq_`

Documentación a subir a “prado”:

1. Describir qué cambios se han realizado sobre el programa de partida y el propósito de dichos cambios
2. Incluir el código fuente completo de la solución adoptada
3. Incluir un listado parcial de la salida del programa.

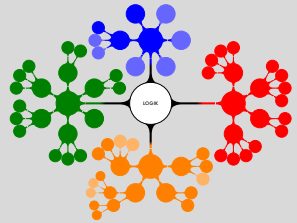


# Cena de los filósofos en MPI

Consideramos un programa `MPI` para el problema de la cena de los filósofos.
En este problema intervienen 5 filósofos y 5 tenedores:

- Los filósofos son 5 procesos (numerados del 0 al 4) que se ejecutan en un bucle indefinido; en cada iteración: comen primero y piensan después (ambas son actividades de duración arbitrariamente larga)
- Los filósofos, para comer, se disponen en una mesa circular donde hay un tenedor entre cada dos filósofos. Cuando un filósofo está comiendo, usa en exclusión mutua sus dos tenedores adyacentes

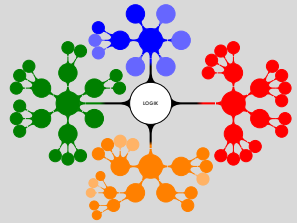
En programación distribuida cada recurso compartido (de uso exclusivo por un único proceso en un instante) debe de implementarse con un proceso gestor adicional (específico para ese recurso), proceso que alterna entre dos estados (libre o en uso)



# Procesos tenedor: sincronización

Para implementar el problema de la cena de los 5 filósofos, debemos de ejecutar **5** procesos de tipo **tenedor**, numerados del **0** al **4**

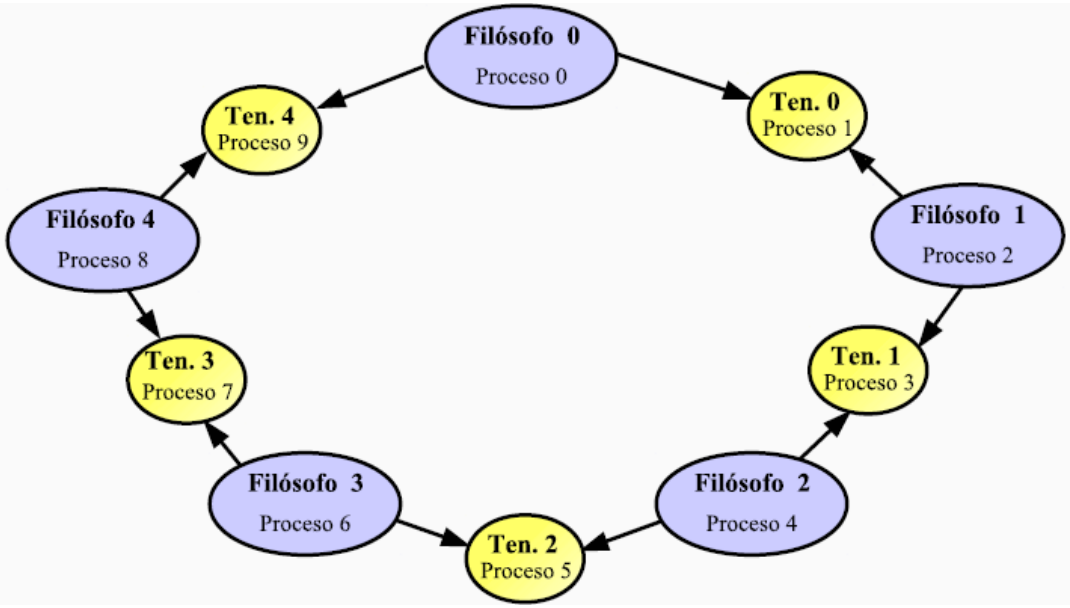
- Cuando un proceso filósofo va a usar un tenedor, debe de enviar (al proceso tenedor correspondiente) un mensaje síncrono antes de usarlo y otro mensaje después de haberlo usado
- Cada proceso tenedor ejecuta un bucle indefinido; al inicio de cada iteración está libre, y realiza estos dos pasos:
  - Espera hasta recibir un mensaje de cualquier filósofo, al recibirlo el tenedor pasa a estar ocupado por ese filósofo
  - Espera hasta recibir un mensaje del filósofo que lo adquirió en el paso anterior. Al recibirlo, pasa a estar libre
- Puesto que el envío (por el filósofo) del mensaje previo al uso es síncrono, supone para dicho filósofo una espera bloqueada hasta adquirir el tenedor en exclusión mutua

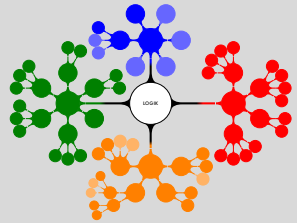


# Identificación de los procesos

Para facilitar la comunicación entre filósofos y tenedores:

- Los procesos filósofos tienen identificadores `MPI` pares, es decir,el filósofo número `i` tendrá identificador `2i`
- Los procesos tenedor tienen identificadores `MPI` impares, es decir, el tenedor número `i` tendrá identificador `2i + 1`

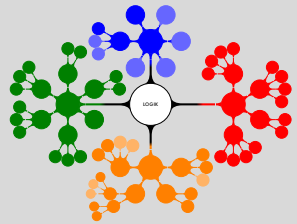


# Cena de los filósofos: programa principal

La función main del programa (archivo `filosofos-plantilla.cpp`) es la siguiente:

```
1  const int num_filosofos = 5 ,
2  num_procesos = 2*num_filosofos ;
3     .....
4  int main( int argc, char** argv )
5  {
6     int id_propio, num_procesos_actual ;
7     MPI_Init(&argc, &argv);
8     MPI_Comm_rank(MPI_COMM_WORLD, &id_propio);
9     MPI_Comm_size(MPI_COMM_WORLD, &num_procesos_actual);
10    if (num_procesos == num_procesos_actual)
11      {if (id_propio%2 == 0) //si es par
12          funcion_filosofos(id_propio);//es un filosofo
13        else //si es impar
14          funcion_tenedores(id_propio);//es un tenedor
15       }
16    else if (id_propio==0)//solo escribe el proceso id==0
17      cerr << "Error: se esperaban 10 procesos.
18               Programa abortado." << endl;
19    MPI_Finalize( );
20 return 0;
21 }
```

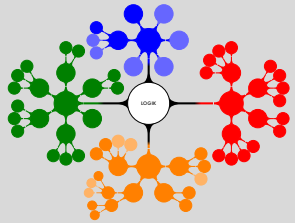


# Procesos filósofos

En cada iteración del bucle un filósofo realiza repetidamente estas acciones:

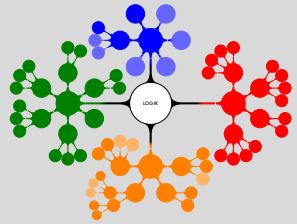
1. Tomar los tenedores (primero el tenedor izquierdo y después el derecho)
2. Comer (bloqueo de duración aleatoria)
3. Soltar tenedores (el orden da igual)
4. Pensar (bloqueo de duración aleatoria)

- Las acciones pensar y comer pueden implementarse mediante un mensaje por pantalla seguido de un retardo durante un tiempo aleatorio
- Las acciones de tomar tenedores y soltar tenedores deben implementarse enviando mensajes síncronos seguros de petición y de liberación a los procesos tenedor situados a ambos lados de cada filósofo



# Esquema de los procesos filósofos

```
1  void funcion_filosofos(int id)
2  { int id_ten_izq= (id+1)%num_procesos,//id. tenedor izq.
3    id_ten_der= (id+num_procesos-1)%num_procesos;
4    //identificador tenedor derecho
5    while ( true ){
6      cout<<"Filosofo "<<id<<" solicita ten. izq."
7           << id_ten_izq <<endl;
8      // ... solicitar tenedor izquierdo (completar)
9      cout<<"Filosofo "<<id<<" solicita ten. der."
10          << id_ten_der <<endl;
11     // ... solicitar tenedor derecho (completar)
12     cout<<"Filosofo "<<id <<" comienza a comer"<<endl ;
13     sleep_for(milliseconds(aleatorio<10,100>()));
14     cout<<"Filosofo "<<id<<" suelta ten. izq. "
15          << id_ten_izq <<endl;
16     // ... soltar el tenedor izquierdo (completar)
17     cout<<"Filosofo "<<id<<" suelta ten. der. "
18          << id_ten_der <<endl;
19     // ... soltar el tenedor derecho (completar)
20     cout<<"Filosofo "<<id<<" comienza a pensar"
21          << endl;
22     sleep_for( milliseconds( aleatorio<10,100>() ) );
23   }
24 }
```

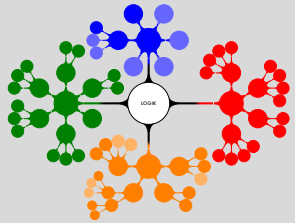


# Procesos tenedor

Cada proceso tenedor ejecutará en un bucle:

1. Esperar hasta recibir un mensaje de cualquier filósofo (lo llamamos mensaje de petición)
2. Esperar hasta recibir un mensaje del mismo filósofo emisor del anterior (lo llamamos mensaje de liberación)

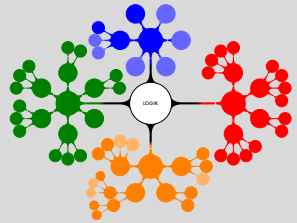
```
1  void funcion_tenedores(int id){// id es el identificador
2                                 // del proceso tenedor
3     int valor, id_filosofo;//valor recibido,
4                            //identificador del filosofo
5     MPI_Status estado;//metadatos de las dos recepciones
6     while ( true ){
7      // ..... recibir peticion de cualquier
8      // filosofo(completar)
9      // ..... guardar en id_filosofo el identificador
10     // del emisor (completar)
11     cout<<"Ten. "<<id <<" cogido por filosofo "
12         << id_filosofo << endl;
13     // ..... recibir liberacion de filosofo
14     //  id_filosofo(completar)
15     cout<<"Ten. "<<id<<" liberado por filo. "
16         <<id_filosofo<<endl;
17    }
18 }
```



# Actividades: soluciones con interbloqueo y sin interbloqueo

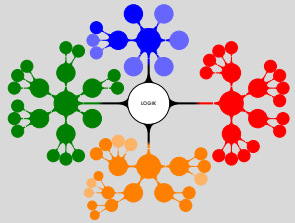
Se propone realizar las siguientes actividades:

1. Implementar una solución distribuida al problema de los filósofos de acuerdo con el esquema descrito en las plantillas. Usar la operación síncrona de envío `MPI_Ssend`. Copia el archivo de la plantilla (`filosofos-plantilla.cpp`) en el archivo `filosofos-interb.cpp` y completa este último archivo
2. El esquema propuesto (cada filósofo coge primero el tenedor de su izquierda y después el de la derecha) puede conducir a interbloqueo:
   - Identifica la secuencia de peticiones de filósofos que conduce a interbloqueo
   - Diseña una modificación que solucione dicho problema
   - Copia `filosofos-interb.cpp` en `filosofos.cpp` e implementa tu solución en este último archivo



# Documentación para subir a Prado

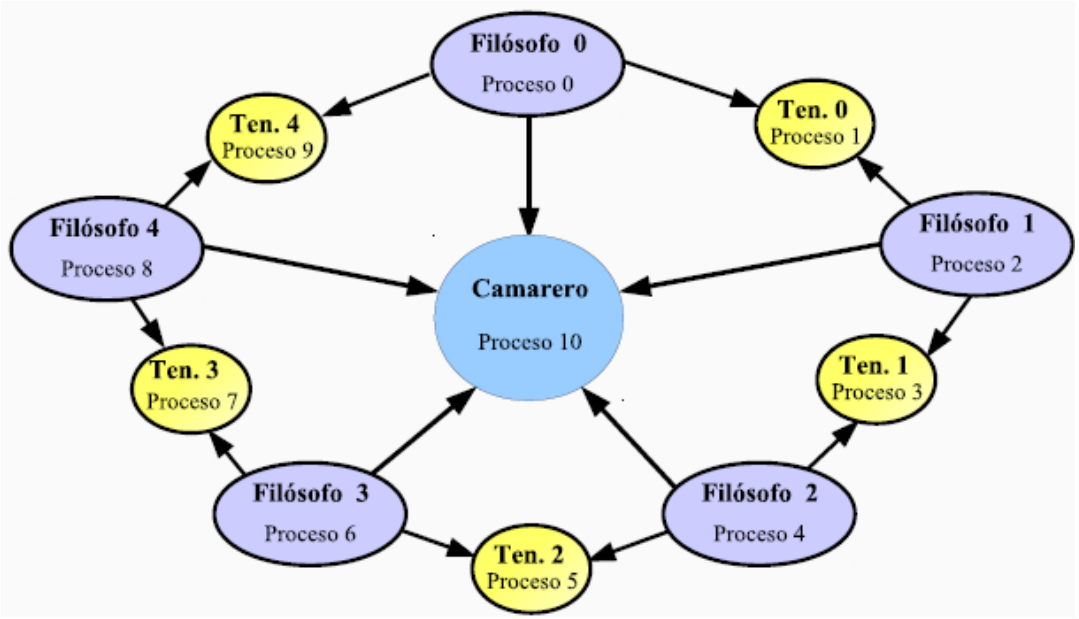
- Describe los aspectos más destacados de tu solución al problema de los filósofos, la situación que conduce al interbloqueo y tu solución al problema del interbloqueo
- Incluye el código fuente completo de la solución adoptada para evitar la posibilidad de interbloqueo
- Incluye un listado parcial de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

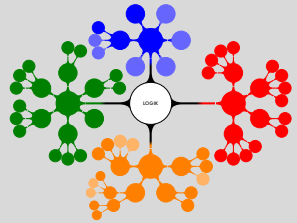


# Cena de los filósofos con camarero

Existe otra opción para solucionar el problema del interbloqueo:

- Se introducen dos pasos nuevos en los filósofos:
  - sentarse en la mesa (antes de coger los tenedores)
  - levantarse de la mesa (después de soltar los tenedores)
  - Un proceso adicional llamado camarero (identificador 10) impedirá que haya 5 filósofos sentados a la vez



# Procesos filósofos y camarero

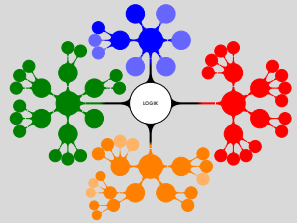
Ahora, cada filósofo ejecutará repetidamente esta secuencia:

1. Sentarse
2. Tomar tenedores
3. Comer
4. Soltar tenedores
5. Levantarse
6. Pensar

Cada filósofo pedirá permiso para sentarse o levantarse haciendo un envío síncrono al camarero. Debemos de implementar de nuevo una espera selectiva en el camarero, el cual:

- llevará una cuenta (`s`) del número de filósofos sentados
- solo cuando `s < 4` aceptará las peticiones de sentarse
- siempre aceptará las peticiones para levantarse

De nuevo, se deben utilizar etiquetas para esta implementación. Definir constantes enteras para etiquetas, cuyos nombres pueden comenzar por `etiq_`.



# Actividades para realizar en esta práctica

Realizar las siguientes actividades:

- Copiar la solución con interbloqueo propuesta (`filosofos-interb.cpp`) sobre un nuevo archivo llamado `filosofos-cam.cpp`
- Implementar, en este último archivo, el método descrito, basado en un proceso camarero con espera selectiva

Documentación para subir a `prado.ugr.es`:
Responder, de forma razonada, a cada uno de los siguientes puntos,

- Describir la solución al problema de los filósofos con camarero central
- Incluir el código fuente completo de la solución adoptada
- Incluir un listado parcial de la salida del programa