EI1024/MT1024 "Programación Concurrente y Paralela" 2023–24	Entregable para Laboratorio
Nombre y apellidos (1):	
Nombre y apellidos (2):	la08_g
Tiempo empleado para tareas en casa en formato *h:mm* (obligatorio):	

Tema 10. Programación de Multicomputadores o MMD

Tema 11. Comunicaciones Punto a Punto en MPI

1 El siguiente código inicializa MPI, obtiene el número de procesos activos (`numProcs`) y el identificador del proceso (`miId`), tras lo cual imprime estas dos informaciones y finaliza MPI.

```
#include <stdio.h> // Definicion de rutinas para E/S
#include <mpi.h>   // Definicion de rutinas de MPI

// Programa principal
int main(int argc, char *argv[])
{
  // Declaracion de variables
  int miId, numProcs;

  // Inicializacion de MPI
  MPI_Init(&argc, &argv);

  // Obtiene el numero de procesos en ejecucion
  MPI_Comm_size(MPI_COMM_WORLD, &numProcs); // Obtiene el numero de procesos en ejecucion
  // Obtiene el identificador del proceso
  MPI_Comm_rank(MPI_COMM_WORLD, &miId); // Obtiene el identificador del proceso

  // ------- PARTE CENTRAL DEL CODIGO (INICIO) -------------------------------

  // Impresion de un mensaje en el terminal
  printf("Hola, soy el proceso %d de %d\n", miId, numProcs);

  // ------- PARTE CENTRAL DEL CODIGO (FINAL) --------------------------------

  // Finalizacion de MPI
  MPI_Finalize();

  return 0;
}
```

Para poder probar este código, primero hay que compilarlo y luego ejecutarlo, utilizando los siguientes comandos:

```
mpicc -o hola hola.c ; mpirun -np 4 ./hola
```

Si se ejecuta varias veces el código, ¿tiene siempre el mismo comportamiento? ¿Por qué?

Si se ejecuta varias veces el código, no tiene siempre el mismo comportamiento, porque el orden en que se imprimen los mensajes puede variar según la disponibilidad de los recursos y el planificador del sistema operativo. Sin embargo, el resultado final es el mismo, es decir, se imprime un mensaje por cada proceso indicando su identificador y el número total de procesos.

2 Realiza las siguientes tareas.

2.1) Escribe un programa en MPI en el que el proceso 0 lea un valor del teclado y lo almacene en la variable `n`. Una vez el proceso 0 haya leído del teclado el valor, **todos los procesos** deberán imprimir el contenido de la variable `n`. Es decir, cada proceso debe imprimir en una misma línea su identificador y el contenido de la variable `n`, tal y como sigue:

```
Proceso <i> con n = <n>
```

Escribe a continuación únicamente la parte central del código.

```
// Declaración e inicialización de la variable n
int n = 0;

// El proceso 0 lee un valor del teclado y lo almacena en n
if (mild == 0) {
  printf("Introduzca un valor para n: ");
  scanf("%d", &n);
}

// Todos los procesos imprimen el contenido de n
printf("Proceso %d con n = %d\n", mild, n);
```

2.2) ¿Todos los procesos tienen el valor leído por el proceso 0 en sus variables `n`? ¿Por qué?

No, todos los procesos no tienen el valor leído por el proceso 0 en sus variables n, porque la variable n es local a cada proceso y no se ha comunicado entre ellos. Por lo tanto, solo el proceso 0 tendrá el valor introducido por el usuario, mientras que el resto de procesos tendrán el valor inicial de 0.

2.3) Modifica el anterior programa para que una vez el proceso 0 haya leído el número, lo envíe él mismo al resto de procesos. Para ello deberá utilizar operaciones de comunicación punto a punto, enviando el contenido de la variable `n` al proceso 1, en primer lugar, luego al proceso 2, y continuando con el resto.
Así, tras esta fase de comunicaciones, todos los procesos deberían tener el valor leído por el proceso 0 en la variable `n`. Finalmente, cada proceso debe imprimir en una misma línea su identificador y el contenido de `n`, tal y como se comentó con anterioridad.
Escribe a continuación únicamente la parte central del código.

```
// Declaración e inicialización de la variable n
int n = 0;

// El proceso 0 lee un valor del teclado y lo almacena en n
if (miId == 0) {
  printf("Introduzca un valor para n: ");
  scanf("%d", &n);
}

// El proceso 0 envía el valor de n al resto de procesos usando comunicaciones punto a punto
if (miId == 0) {
  for (int i = 1; i < numProcs; i++) {
    MPI_Send(&n, 1, MPI_INT, i, 0, MPI_COMM_WORLD); // Envía n al proceso i con etiqueta 0
  }
}

// El resto de procesos reciben el valor de n del proceso 0 usando comunicaciones punto a punto
if (miId != 0) {
  MPI_Recv(&n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // Recibe n del proceso 0 con
etiqueta 0
}

// Todos los procesos imprimen el contenido de n
printf("Proceso %d con n = %d\n", miId, n);
```

2.4) ¿Todos los procesos tienen el valor leído por el proceso 0 en sus variables `n`? ¿Por qué?

Sí, todos los procesos tienen el valor leído por el proceso 0 en sus variables n, porque el proceso 0 ha enviado el valor de n al resto de procesos usando operaciones de comunicación punto a punto, y el resto de procesos han recibido el valor de n del proceso 0 usando operaciones de comunicación punto a punto. Por lo tanto, todos los procesos han sincronizado el valor de n.

3 En este ejercicio se va a implementar el algoritmo ping-pong para medir la latencia y el ancho de banda de la red de comunicaciones que interconecta dos procesos.

Puedes aprovechar el siguiente código:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

// ================================================================================
int main( int argc, char * argv[] ) {
  // Declaracion de variables.
  MPI_Status  s;
  int         numProcs, miId, numArgs, vecArgs[ 5 ] = { 0, 0, 0, 0, 0 };
  int         numMensajes, minTam, maxTam, incTam, tam, i, j;
  char        * ptrWorkspace;
  double      t1, t2, tiempoTotal, tiempoPorMensajeEnMicroseg,
              anchoDeBandaEnMbs;
  char        miNombreProc[ MPI_MAX_PROCESSOR_NAME ];
  int         longNombreProc;

  // Inicializacion de MPI.
  MPI_Init( & argc, & argv );
  MPI_Comm_size( MPI_COMM_WORLD, & numProcs );
  MPI_Comm_rank( MPI_COMM_WORLD, & miId );

  // Comprobacion del numero de procesos.
  if( numProcs < 2 ) {
    if ( miId == 0 ) {
      fprintf( stderr, "\nError: Al menos se deben iniciar dos procesos\n\n" );
    }
    MPI_Finalize();
    return( -1 );
  }

  // Imprime el nombre de los procesadores.
  MPI_Get_processor_name( miNombreProc, & longNombreProc );
  printf( "Proceso %d  Se ejecuta en: %s\n", miId, miNombreProc );

  // El proceso 0 inicializa las cinco variables.
  if( miId == 0 ) {
    numArgs     = argc;
    numMensajes = ( numArgs > 1 )? atoi( argv[ 1 ] ): -1;
    minTam      = ( numArgs > 2 )? atoi( argv[ 2 ] ): -1;
    if( numArgs == 5 ) {
      maxTam = atoi( argv[ 3 ] );
      incTam = atoi( argv[ 4 ] );
    } else {
      maxTam = minTam;
      incTam = 1;
    }
  }

  // El proceso 0 prepara el vector con las cinco variables.
  if( miId == 0 ) {
    vecArgs[ 0 ] = numArgs;
    vecArgs[ 1 ] = numMensajes;
    vecArgs[ 2 ] = minTam;
    vecArgs[ 3 ] = maxTam;
    vecArgs[ 4 ] = incTam;
  }
```

```
// Difusion del vector vecArgs con operaciones punto a punto.
// ... (A)

// El resto de procesos inicializan las cinco variables con la
// informacion del vector. El proceso 0 no tiene que hacerlo porque
// ya habia inicializado las variables.
if( miId != 0 ) {
  numArgs     = vecArgs[ 0 ];
  numMensajes = vecArgs[ 1 ];
  minTam      = vecArgs[ 2 ];
  maxTam      = vecArgs[ 3 ];
  incTam      = vecArgs[ 4 ];
}

// Todos los procesos comprueban el numero de argumentos de entrada.
if( ( numArgs != 3 )&&( numArgs != 5 ) ) {
  if ( miId == 0 ) {
    fprintf( stderr, "\nUso: a.out numMensajes minTam [ maxTam incTam ]\n\n" );
  }
  MPI_Finalize();
  return( -1 );
}

// Imprime los parametros de trabajo.
if( miId == 0 ) {
  printf( "  Numero de procesos:  %5d\n", numProcs );
  printf( "  Numero de mensajes:  %5d\n", numMensajes );
  printf( "  Tamanyo inicial   :  %5d\n", minTam );
  printf( "  Tamanyo final     :  %5d\n", maxTam );
  printf( "  Incremento        :  %5d\n", incTam );
}

// Crea un vector capaz de almacenar el espacio maximo.
if( maxTam != 0 ) {
  ptrWorkspace = ( char * ) malloc( maxTam );
  if( ptrWorkspace == NULL ) {
    if ( miId == 0 ) {
      fprintf( stderr, "\nError en Malloc: Devuelve NULL.\n\n");
    }
    MPI_Finalize();
    return( -1 );
  }
} else {
  ptrWorkspace = NULL;
}

// Imprime cabecera de la tabla.
if ( miId == 0 ) {
  printf( " Comenzando bucle para envio de informacion\n\n" );
  printf( " Tamanyo(bytes)   tiempoTotal(s.)" );
  printf( "  tiempoPorMsg(microsec.)  AnchoBanda(MB/s)\n");
  printf( " ------------------------------------------" );
  printf( "-----------------------------------------\n");
}

// Sincronizacion de todos los procesos
MPI_Barrier( MPI_COMM_WORLD );

// Bucle para pruebas de tamanyos.
for( tam = minTam; tam <= maxTam; tam += incTam ) {
```

```
      // Sincronizacion de todos los procesos
      MPI_Barrier( MPI_COMM_WORLD );

      // Bucle de envio/recepcion de "numMensajes" de tamanyo "tam" y toma de tiempos.
      // ... (B)

      // Calculo de prestaciones: tiempoTotal, tiempoPorMensajeEnMicroseg,
      // anchoDeBandaEnMbs.
      // ... (C)

      // Escritura de resultados.
      if ( miId == 0 ) {
        printf("    %8d", tam );
        if( tiempoTotal >= 0.0) {
          printf("    %15.6f", tiempoTotal );
          printf("    %15.3f", tiempoPorMensajeEnMicroseg );
          printf("    %21.2f", anchoDeBandaEnMbs );
          printf("\n");
        } else {
          printf(": No se han realizado los calculos.\n" );
        }
      }
    }

    // Imprime final de la tabla.
    if ( miId == 0 ) {
      printf( " ----------------------------------------" );
      printf( "----------------------------------------\n");
    }

    // Liberacion del espacio.
    if( maxTam != 0 ) {
      free( ptrWorkspace );
    }

    // Cierre de MPI.
    MPI_Finalize();

    if ( miId == 0 ) {
      printf( "Fin del programa\n" );
    }
    return 0;
  }
```

```
// Difusion del vector vecArgs con operacion de difusion colectiva.
// El proceso 0 es el que tiene el vector original y lo difunde al resto de procesos.
// Todos los procesos llaman a la misma funcion con los mismos parametros.
MPI_Bcast(vecArgs, 5, MPI_INT, 0, MPI_COMM_WORLD);
```

3.1) Introduce en el programa anterior, el código que permite que el proceso 0 envíe el vector vecArgs al resto de procesos. Busca la definición del vector en el código para identificar su tamaño y el tipo base de sus elementos.

Fíjate que estas líneas se deben insertar a continuación de la línea marcada con "(A)".

Para comprobar el correcto funcionamiento del programa, compila y ejecuta el código:

```
mpicc -o anchoBanda anchoBanda.c
mpirun -np 4 ./anchoBanda 2000 1024
```

Escribe a continuación la parte de tu código que realiza tal tarea:

```
// Difusion del vector vecArgs con operaciones punto a punto.
if (miId == 0){{^1^][1][^2^][2]
 // El proceso 0 envia el vector a los demas procesos, uno por uno.
  for (i = 1; i < numProcs; i++) {
   // Envia el vector vecArgs al proceso i, con la etiqueta 0 y el comunicador MPI_COMM_WORLD.
   MPI_Send(vecArgs, 5, MPI_INT, i, 0, MPI_COMM_WORLD);
  }
} else {
  // El resto de procesos reciben el vector del proceso 0, con la etiqueta 0 y el comunicador MPI_COMM_WORLD.
  MPI_Recv(vecArgs, 5, MPI_INT, 0, 0, MPI_COMM_WORLD, &s);
}
```

ATENCIÓN: Los ejercicios anteriores deben realizarse en casa. Los siguientes, en el aula.

3.2) Introduce en el programa anterior, el código que permite que el proceso 0 envíe `num` mensajes de tamaño `tam` bytes al proceso 1 y que éste devuelva un mensaje de tamaño 0 bytes cuando reciba el último de ellos.

Incluye también las líneas que permite al proceso 0 identificar cuando se inician (`t1`) y finalizan (`t2`) las operaciones de comunicación, utilizando la rutina `MPI_Wtime`.

Fíjate que estas líneas se deben insertar a continuación de la línea marcada con "(B)".

Escribe a continuación la parte de tu código que realiza tal tarea:

```
// Bucle de envio/recepcion de "numMensajes" de tamanyo "tam" y toma de tiempos.
if (miId == 0) {
  // Process 0 sends messages to process 1 and waits for an acknowledgment
  t1 = MPI_Wtime(); // Start the timer
  for (i = 0; i < numMensajes; i++) {
    // Send a message of size tam bytes to process 1, with the tag i and the communicator
MPI_COMM_WORLD
    MPI_Send(ptrWorkspace, tam, MPI_CHAR, 1, i, MPI_COMM_WORLD);
  }
  // Receive a message of size 0 bytes from process 1, with any tag and the communicator
MPI_COMM_WORLD
  MPI_Recv(ptrWorkspace, 0, MPI_CHAR, 1, MPI_ANY_TAG, MPI_COMM_WORLD, &s);
  t2 = MPI_Wtime(); // Stop the timer
  tiempoTotal = t2 - t1; // Calculate the total time
} else if (miId == 1) {
  // Process 1 receives messages from process 0 and sends an acknowledgment when it receives the last
one
  for (i = 0; i < numMensajes; i++) {
    // Receive a message of size tam bytes from process 0, with the tag i and the communicator
MPI_COMM_WORLD
    MPI_Recv(ptrWorkspace, tam, MPI_CHAR, 0, i, MPI_COMM_WORLD, &s);
    // If it is the last message, send a message of size 0 bytes to process 0, with the tag i and the
communicator MPI_COMM_WORLD
    if (i == numMensajes - 1) {
      MPI_Send(ptrWorkspace, 0, MPI_CHAR, 0, i, MPI_COMM_WORLD);
    }
  }
```

3.3) Introduce en el programa anterior, el código que permite al proceso 0 medir el coste de cada comunicación (en segundos), así como la duración media del envío de cada mensaje (en microsegundos) y el ancho de banda de la comunicación (en Megabytes por segundo). Fíjate que estas líneas se deben insertar a continuación de la línea marcada con "(C)". Escribe a continuación la parte de tu código que realiza tal tarea:

```
// Calculo de prestaciones: tiempoTotal, tiempoPorMensajeEnMicroseg,[^1^][1]
// anchoDeBandaEnMbs[^2^][2][^3^][3].
tiempoPorMensajeEnMicroseg = (tiempoTotal / numMensajes) * 1e6; // Convert seconds to
microseconds
anchoDeBandaEnMbs = (tam * numMensajes * 8) / (tiempoTotal * 1e6); // Convert bytes to megabits
```

3.4) Verifica que el código funciona correctamente incluyendo el número de mensajes a enviar y su tamaño como argumento en la línea de órdenes. Por ejemplo, la siguiente orden,

```
mpirun -np 4 ./anchoBanda 2000 1024
```

realiza el envío de 2000 mensajes de tamaño 1024 bytes.
Escribe el resultado de esta ejecución:

3.5) Verifica que el código funciona incluyendo todos los parámetros: el número de mensajes a enviar, el tamaño mínimo y máximo de los mensajes, así como el incremento en el tamaño del mensaje. Así, la siguiente orden

```
mpirun -np 4 ./anchoBanda 2000 0 10240 1024
```

realizará el envío de 2000 mensajes de tamaño 0 (0K), 2000 mensajes de tamaño 1024 (1K), 2000 mensajes de tamaño 2048 (2K), y así sucesivamente hasta enviar 2000 mensajes de tamaño 10240 (10K).

Ejecuta la prueba anterior en patan y completa la siguiente tabla, calculando el ancho de banda en Megabytes por segundo y redondeando el resultado con dos decimales.

Tamaño	Tiempo por mensaje (microseg.)	Ancho de banda (MB/s)
0		
1024		
2048		
3072		
4096		
5120		
6144		
7168		
8192		
9216		
10240		

Justifica los resultados.

. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

3.6) ¿Cuál es la latencia de las comunicaciones? ¿Cómo lo has calculado?
¿Cómo influye el tamaño de mensaje en el ancho de banda?
¿Qué valor tomarías como el ancho de banda real?

. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .