

2 Realiza las siguientes tareas.

2.1) Compila y ejecuta el siguiente código con 4 procesos:

```
#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 identificador del proceso
    MPI_Comm_rank(MPI_COMM_WORLD, &miId);

    // ----- PARTE CENTRAL DEL CODIGO (INICIO) -----
    // Definicion e inicializacion de la variable n
    int n = ( miId + 1 ) * numProcs;

    // El proceso 0 lee un numero desde teclado sobre la variable n
    if ( miId == 0 ) {
        printf ("Dame un numero —> \n"); scanf ("%d", &n);
    }
    // Impresion de la variable n en todos los procesos
    printf ("Proceso <%d> con n = %d\n", miId, n);

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

    // Finalizacion de MPI
    MPI_Finalize();

    return 0;
}
```

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

No, todos los procesos a parte del 0 tendrán $n = (miId+1) * numProcs$ porque no comparten espacio de memoria. Solo el proceso 0 ejecuta el `scanf` que cambia el contenido de **n** y por tanto solo este tendrá la **n** cambiada.

.....

.....

.....

.....

2.3) Modifica el anterior programa para que una vez el proceso 0 haya leído el número, él mismo lo envíe al resto de procesos. Para ello deberá utilizar operaciones de **comunicación punto a punto**, enviando, en primer lugar, el contenido de la variable **n** al proceso 1, 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.

```
.....
int n = 0;
MPI_Status s;
.....
if ( mild == 0) {
    printf ("Dame un numero --> \n");
    scanf ("%d", &n);
.....
    for (int i = 1; i < numProcs; i++) {
        MPI_Send(&n, 1, MPI_INT, i, 33, MPI_COMM_WORLD);
    }
} else {
    MPI_Recv(&n, 1, MPI_INT, 0, 33, MPI_COMM_WORLD, &s);
}
printf ("Proceso <%d> con n = %d\n", mild, n);
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
```

- 2.4) ¿Todos los procesos tienen el valor leído por el proceso 0 en sus variables **n**? ¿Por qué?
 Sí porque los procesos distintos a 0 se esperarán hasta recibir el mensaje con el valor de **n**.
 Una vez recibido ejecutarán el printf con su mild y la misma **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[ MPLMAX_PROCESSOR_NAME ];
    int longNombreProc;
```

```

// Inicializacion de MPI.
MPI_Init( & argc, & argv );
MPI_Comm_size( MPLCOMM_WORLD, & numProcs );
MPI_Comm_rank( MPLCOMM_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( mild == 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 ( mild == 0 ) {
            fprintf( stderr, "\nError en Malloc: Devuelve NULL.\n\n" );
        }
        MPI_Finalize();
        return( -1 );
    }
} else {
    ptrWorkspace = NULL;
}

// Imprime cabecera de la tabla.
if ( mild == 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( MPLCOMM_WORLD );

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

    // Sincronizacion de todos los procesos
    MPI_Barrier( MPLCOMM_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 ( mild == 0 ) {
        printf( "    %8d", tam );
        if( tiempoTotal >= 0.0 ) {
            printf( "    %15.6f", tiempoTotal );
            printf( "    %15.3f", tiempoPorMensajeEnMicroseg );
            printf( "    %21.2f", anchoDeBandaEnMbs );
            printf( "\n" );
        } else {

```


- 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. Tras recibir todos los mensajes, el proceso 1 envía un mensaje de tamaño 0 bytes al proceso 0. Utiliza la función `MPI_Ssend` para estos envíos.

En estas comunicaciones se debe utilizar el `ptrWorkspace`, un vector con `tamMax` elementos de tipo `char`. Por tanto, las comunicaciones deben enviar elementos `MPI_CHAR` o `MPI_BYTE`.

Incluye también las líneas que permitan 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)”.

Cuando hayas verificado que funciona, sustituye `MPI_Ssend` por `MPI_Send`.

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

```
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
```

- 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:

```
.....
.....
.....
.....
.....
.....
.....
.....
```

- 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 ese comando en la cola de karen y completa la siguiente tabla, mostrando 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?

.....

.....

.....

.....

.....