

EI1024/MT1024 “Programación Concurrente y Paralela” 2022–23 Nombre y apellidos (1): Nombre y apellidos (2): Tiempo empleado para tareas en casa en formato <i>h:mm</i> (obligatorio):	Entregable para Laboratorio la08_g
---	---

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(MPLCOMM_WORLD, &numProcs);
    // Obtiene el identificador del proceso
    MPI_Comm_rank(MPLCOMM_WORLD, &miId);

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

    // Finalizacion de MPI
    MPI_Finalize();

    return 0;
}
```

Para poder probar este código, primero hay que compilarlo, a través del comando:

```
mpicc -o hola hola.c
```

Mientras que para ejecutarlo con 4 procesos, se utiliza el siguiente comando:

```
mpirun -np 4 ./hola
```

Cuando se ejecuta varias veces el anterior comando, ¿siempre tiene el mismo comportamiento?
¿Por qué?

.....

2 Realiza las siguientes tareas.

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

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

// =====
int main( int argc , char * argv[] ) {
    int numProcs, miId;

    MPI_Init( & argc , & argv );

    MPI_Comm_size( MPLCOMM_WORLD, & numProcs );
    MPI_Comm_rank( MPLCOMM_WORLD, & miId );

    // ----- INICIO CODIGO A MODIFICAR -----
    int n = ( miId + 1 ) * numProcs;

    if ( miId == 0 ) {
        printf ( "Dame un numero —> \n" ); scanf ( "%d" , &n );
    }
    printf ( "Proceso <%d> con n = %d\n" , miId , n );

    // ----- FIN CODIGO A MODIFICAR -----

    MPI_Finalize();

    printf( "Fin de programa\n" );
    return 0;
}
```

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

.....

.....

.....

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 utiliza el código que aparece a continuación:

```
// ----- INICIO CODIGO A MODIFICAR -----
int n = ( miId + 1 ) * numProcs, i;
MPI_Status st;

if ( miId == 0 ) {
    printf ( "Dame un numero —> \n" ); scanf ( "%d" , &n );
    for ( i=1; i<numProcs; i++ )
        MPI_Send ( &n, 1, MPI_INT, i , 88, MPLCOMM_WORLD );
} else
    MPI_Recv ( &n, 1, MPI_INT, 0, 88, MPLCOMM_WORLD, &st );
printf ( "Proceso <%d> con n = %d\n" , miId , n );

// ----- FIN CODIGO A MODIFICAR -----
```

En el código, el proceso 0 realiza un envío (**MPI_Send**) al proceso 1, luego al proceso 2, y luego al resto de procesos. Por su parte, el resto de procesos reciben un único mensaje (**MPI_Recv**) del proceso 0.

Seguidamente se comenta el significado de algunos parámetros de las rutinas de MPI:

- El primer parámetro indica la dirección donde se encuentra la información a enviar o donde se debe almacenar el resultado. Si se desea enviar un escalar, debe añadirse el operador &, cuestión que no debe realizar cuando se envían vectores.
- El segundo y tercer parámetro indican el número de elementos a enviar/recibir y el tipo base de estos elementos. En el código anterior, se comunica 1 número entero.
- El cuarto parámetro identifica el proceso destino u origen del mensaje. Es por ello que en los `MPI_Recv` siempre aparezca un 0.

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

.....

.....

.....

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 ( 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( 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 ( miId == 0 ) {
        printf( "      %d", 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;
}

```

- 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)”.

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

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

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

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 realice `numMensajes` envíos de tamaño `tam` bytes al proceso 1, y que éste devuelva un único mensaje de tamaño 0 bytes tras recibir el último de ellos.

Incluye también las líneas que permite al proceso 0 identificar cuando se inician (**t1**) y finalizan (**t2**) todas 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:

- 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 orden

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

realizará 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 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?

.....

.....

.....

.....

.....