EI1024/MT1024 "Programación Concurrente y Paralela" 2025–26	Entregable para Laboratorio
Nombre y apellidos (1): Maria Pallotti Romero	
Nombre y apellidos (2):	
Tiempo empleado para tareas en casa en formato *h:mm* (obligatorio): 1:30	la10_g

Tema 11. Comunicaciones Punto a Punto en MPI

Tema 12. Comunicaciones Colectivas en MPI

1 Cada proceso dispone de un dato propio y posiblemente distinto en una variable denominada `dato` de tipo entero. Por simplicidad, todos los procesos inicializan dicha variable de la misma forma: La variable `dato` en el proceso `miId` toma el valor `numProcs - miId + 1`.

Se desea que el proceso 0 calcule y obtenga la suma de los valores almacenados en las variables `dato` de todos los procesos, incluyendo el suyo. Al final de la ejecución, cada proceso debe imprimir su valor inicial y, además, el proceso 0 debe imprimir la suma final.

1.1) A partir del fichero `plantilla.c`, implementa el programa `ejer_1_1.c`, en el que todos los procesos envían su valor local al proceso 0, y éste calcula la suma de los valores recibidos en operaciones de comunicación punto a punto,

Comprueba que el programa funciona correctamente con varias configuraciones de procesos. Escribe a continuación la parte del programa principal que realiza tal tarea: la declaración de variables, las comunicaciones y la impresión de resultados.

1.2) A partir del fichero `plantilla.c`, implementa el programa `ejer_1_2.c`, en el que el proceso 0 calcule la suma mediante operaciones de comunicación colectivas.

Comprueba que el programa funciona correctamente con varias configuraciones de procesos.

Escribe a continuación la parte del programa principal que realiza tal tarea: la declaración de variables, las comunicaciones y la impresión de resultados.

2

Cada proceso dispone de un dato propio y posiblemente distinto en una variable denominada `dato` de tipo entero. Por simplicidad, todos los procesos inicializan dicha variable de la misma forma: La variable `dato` en el proceso `miId` toma el valor `numProcs - miId + 1`.

Se desea que el proceso 0 calcule y obtenga la suma de los valores almacenados en las variables `dato` de los **procesos pares**, incluyendo el suyo. Es decir, el valor de `dato` de los procesos impares no deben reflejarse en la suma. Al final de la ejecución, cada proceso debe imprimir su valor inicial y, además, el proceso 0 debe imprimir la suma final.

2.1) A partir del fichero `plantilla.c`, implementa el programa `ejer_2_1.c`, en el que todos los procesos pares envían su valor local al proceso 0, y éste calcula la suma de los valores recibidos en operaciones de comunicación punto a punto.

IMPORTANTE: Solo deben participar en la comunicación los procesos pares.

Comprueba que el programa funciona correctamente con varias configuraciones de procesos.

Escribe a continuación la parte del programa principal que realiza tal tarea: la declaración de variables, las comunicaciones y la impresión de resultados.

2.2) A partir del fichero `plantilla.c`, implementa el programa `ejer_2_2.c`, en el que todos los procesos pares envían su valor local al proceso 0, y éste calcula la suma mediante operaciones de comunicación punto a punto.

IMPORTANTE: Solo deben participar en la comunicación los procesos pares.

IMPORTANTE: Cada proceso solo recibe y envía un mensaje, como máximo.

IMPORTANTE: El proceso 0 no envía mensajes y el mayor proceso par no recibe.

Comprueba que el programa funciona correctamente con varias configuraciones de procesos.

Escribe a continuación la parte del programa principal que realiza tal tarea: la declaración de variables, las comunicaciones y la impresión de resultados.

2.3) A partir del fichero `plantilla.c`, implementa el programa `ejer_2_3.c`, en el que el proceso 0 calcule la suma mediante operaciones de comunicación colectivas.

Es obligatorio que todos los procesos participen en una operación de comunicación colectiva, ya que, en caso contrario, el programa se bloquea, pero en este caso se desea que **los valores de los procesos impares no participen en la suma**.

La solución más sencilla a este problema se consigue **utilizando una variable auxiliar** para realizar la operación que se inicializa convenientemente: Los procesos pares con el valor de su variable `dato`, mientras que los procesos impares la inicializan a cero.

Comprueba que el programa funciona correctamente con varias configuraciones de procesos.

Escribe a continuación la parte del programa principal que realiza tal tarea: la declaración de variables, las comunicaciones y la impresión de resultados.

3 Se desea implementar un programa en el que todos los procesos colaboren en el cálculo de la suma resultante de aplicar una función a los elementos de un vector de números reales de doble precisión que aparece en el proceso 0. Para ello, en primer lugar el vector debe ser distribuido entre los procesos, tras lo cual cada proceso calcula una suma local. Finalmente las sumas locales se acumulan sobre el proceso 0.

El vector a distribuir se denomina `vectorInicial` y su dimensión se guarda en la variable `dimVectorInicial`, obtenido como parámetro de entrada del programa. Por simplicidad, sólo se consideran tamaños de vector inicial que sea divisibles por el número de procesos. Antes de proceder al reparto, el proceso 0 inicializa el vector inicial, y calcula la suma resultante de aplicar una función a los componentes en `sumaInicial`.

En el reparto de `vectorInicial` se utiliza una distribución por bloques, en la que el proceso

0 también se queda con un bloque de datos. Todos los procesos, deben guardar sus respectivos datos en un vector denominado `vectorLocal`, cuya dimensión se almacena en la variable `dimVectorLocal`.

La suma resultante de aplicar la función a los elementos de `vectorLocal` que realiza cada proceso se almacena sobre `sumaLocal`, mientras que la acumulación de estos valores debe quedar en `sumaFinal` del proceso 0.

Como punto de partida debes tomar el siguiente código, que aparecen en el fichero `vector_3_1.c`, el cual deberás compilar incluyendo la opción **-lm** al final del comando `mpicc`, que informa al enlazador que tiene que incorporar la librería matemática:

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

//#define IMPRIME 1
//#define COSTOSA 1

// ==========================================================================

double evaluaFuncion( double x ) {
#ifdef COSTOSA
  return sin( exp( -x ) + log10( 1 + x ) );
#else
  return 2.5 * x;
#endif
}

void inicializaVectorX ( double vectorX [ ], int dim ) {
  int i;
  if( dim == 1 ) {
    vectorX[ 0 ] = 0.0;
  } else {
    for( i = 0; i < dim; i++ ) {
      vectorX[ i ] = 10.0 * ( double ) i / ( ( double ) dim - 1 );
    }
  }
}

// ==========================================================================
int main( int argc, char *argv[] ) {
  int      dimVectorInicial, dimVectorLocal, i, prc;
  int      miId, numProcs;
  double   *vectorInicial = NULL, *vectorLocal = NULL;
  double   sumaInicial, sumaLocal, sumaFinal;
  double   t1, t2, tSec, tDis, tPar;
  MPI_Status st;

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

  // En primer lugar se comprueba si el numero de parametros es valido
  if( argc != 2 ) {
    if ( miId == 0 ) {
      fprintf( stderr, "\n" );
      fprintf( stderr, "Uso: a.out dimension\n");
      fprintf( stderr, "\n" );
```

```
    }
    MPI_Finalize();
    return( -1 );
  }

  // Todos los procesos deben comprobar que la dimension de vectorInicial "n"
  // es multiplo del numero de procesos.
  dimVectorInicial = atoi(argv[1]);
  if( ( dimVectorInicial % numProcs ) != 0 ) {
    if( miId == 0 ) {
      fprintf( stderr, "\n" );
      fprintf( stderr,
          "ERROR: La dimension %d no es multiplo del numero de procesos: %d\n",
          dimVectorInicial, numProcs );
      fprintf( stderr, "\n" );
    }
    MPI_Finalize();
    exit( -1 );
  }

  // El proceso 0 crea e inicializa "vectorInicial".
  if( miId == 0 ) {
    vectorInicial = ( double * ) malloc( dimVectorInicial * sizeof( double ) );
    inicializaVectorX ( vectorInicial, dimVectorInicial );
  }

#ifdef IMPRIME
  // El proceso 0 imprime el contenido de "vectorInicial".
  if( miId == 0 ) {
    for( i = 0; i < dimVectorInicial; i++ ) {
      printf( "Proc: %d.  vectorInicial[ %3d ] = %lf\n",
              miId, i, vectorInicial[ i ] );
    }
  }
#endif

  // El proceso 0 suma todos los elementos de vectorInicial
  if( miId == 0 ) {
    // Calculo en secuencial de la reduccion sin temporizacion
    sumaInicial = 0;
    for( i = 0; i < dimVectorInicial; i++ ) {
      sumaInicial += evaluaFuncion( vectorInicial[ i ] );
    }

    // Inicio del calculo de la reduccion en secuencial y de su coste (tSec)
    // t1 = ... ; // ... (A)
    sumaInicial = 0;
    for( i = 0; i < dimVectorInicial; i++ ) {
      sumaInicial += evaluaFuncion( vectorInicial[ i ] );
    }
    // Finalizacion del calculo de la reduccion en secuencial y de su coste (tSec)
    // t2 = ... ; // ... (B)
    tSec = t2 - t1;
  }

  // Todos los procesos crean e inicializan "vectorLocal".
  // La siguiente linea no es correcta. Debes arreglarla.
  // dimVectorLocal = ... ; // ... (C)
  vectorLocal = ( double * ) malloc( dimVectorLocal * sizeof( double ) );
  for( i = 0; i < dimVectorLocal; i++ ) {
    vectorLocal[ i ] = -1.0;
```

```
  }

  MPI_Barrier( MPI_COMM_WORLD );
  // Distribucion por bloques de "vectorInicial" y calculo de su coste (tDis).
  // Al final de esta fase, cada proceso debe tener sus correspondientes datos
  // propios en "vectorLocal", y el proceso 0 debe saber cuanto ha costado (tDis).
  // ... (D)

#ifdef IMPRIME
  // Todos los procesos imprimen su vector local.
  for( i = 0; i < dimVectorLocal; i++ ) {
    printf( "Proc: %d.  vectorLocal[ %3d ] = %lf\n",
            miId, i, vectorLocal[ i ] );
  }
#endif

  MPI_Barrier( MPI_COMM_WORLD );
  // Inicio del calculo de la reduccion en paralelo y de su coste (tPar).
  // ... (E)

  // Cada proceso suma la aplicacion de la funcion sobre los elementos de vectorLocal
  // ... (F)

  // Se acumulan las sumas locales de cada procesador en sumaFinal sobre el proceso 0
  // ... (G)

  // Finalizacion del calculo de la reduccion en paralelo y de su coste (tPar).
  // ... (H)

  // El proceso 0 imprime la sumas, los costes y los incrementos
  if ( miId == 0) {
    // Imprimir Sumas(sumaInicial, sumaFinal, diferencia)
    printf( "Proc: %d , sumaInicial = %lf , sumaFinal = %lf , diff = %lf\n",
                miId, sumaInicial, sumaFinal, sumaInicial - sumaFinal);
    printf( "Proc: %d , tSec = %lf , tPar = %lf , tDis = %lf\n",
                miId, tSec, tPar, tDis);
    // Imprimir Incrementos(tSec vs tPar , tSec vs (tDis+tPar) )
    // printf ( "Proc: %d , incSec_Par = %lf , incSec_DisPar = %lf\n",
    // ... (I)
  }

  // El proceso 0 borra el vector inicial.
  if( miId == 0 ) {
    free( vectorInicial );
  }

  // Todos los procesos borran su vector local.
  free( vectorLocal );

  // Finalizacion de MPI.
  MPI_Finalize();

  // Fin de programa.
  printf( "Proc: %d Final de programa\n", miId );
  return 0;
}
```

3.1) En este apartado debes calcular el valor de la variable `dimVectorLocal`, y realizar el reparto del vector `VectorInicial`, utilizando únicamente envíos y recepciones **punto a punto**.
Además debes incluir las órdenes que permiten calcular el coste de la ejecución secuencial (tSec) y las que permiten calcular el coste de la distribución de los datos (tDis).
Estas líneas se deben insertar a continuación de las líneas marcadas con "(A)-(D)".
Escribe a continuación la parte del programa principal que realiza estas tareas: la inicialización de variables y las comunicaciones.

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

3.2) Haz una copia del anterior programa y denomínala `vector_3_2.c`

Modifica el programa para que, después de recibir los datos, todos los procesos sumen el resultado de `evaluaFuncion` sobre sus valores locales en `sumaLocal`. A continuación, todos los procesos deben enviar su `sumaLocal` al proceso 0, el cual debe acumular todos los valores recibidos junto a su `sumaLocal` para obtener `sumaFinal`.

Además debes incluir las órdenes que permiten calcular el coste de la ejecución paralela (tPar), así como la impresión de los costes y los resultados.

Comprueba que el resultado de la suma paralela es correcta, comparando el valor de las variables `sumaInicial` y `sumaFinal`.

En este apartado sólo deben emplearse comunicaciones **punto a punto**.

Estas líneas se deben insertar a continuación de las líneas marcadas con "(E)-(I)".

Escribe a continuación la parte del programa principal que realiza tal tarea: la acumulación de resultados, las comunicaciones y la impresión de resultados.

3.3) Haz una copia del anterior programa y denomínala `vector_3_3.c`

Modifica el programa del ejercicio anterior para que tanto el **reparto** de los elementos del vector como la **recogida** de los valores de `sumaLocal` se realicen utilizando operaciones de **comunicación colectiva**.

En este ejercicio **no se pueden emplear operaciones de reducción**.

Estas líneas deben sustituir a las ya insertadas a continuación de las líneas "(D)" y "(G)".

Escribe a continuación la parte del programa principal que realiza tal tarea: la declaración de variables, el reparto de los datos, la reunión de los resultados y la acumulación final.

Recuerda que en lenguaje `C`, la declaración y creación de un vector que contenga `N` números reales de doble precisión se realiza utilizando las siguientes instrucciones:

```
double *vector = NULL;
vector = (double *) malloc ( sizeof ( double ) * N );
```

y la destrucción del vector cuando ya no es útil, como sigue:

```
free( vector ); vector = NULL;
```

3.4) Haz una copia del anterior programa y denomínala `vector_3_4.c`
Modifica el programa del ejercicio anterior para que la suma de los valores de `sumaLocal` se realice con una operación de **comunicación colectiva de reducción**. El resultado debe quedar en el proceso 0.
Estas líneas deben sustituir a las ya insertadas a continuación de la línea "(G)".
Escribe a continuación la parte del programa principal que realiza tal tarea: la declaración de variables (si fuese necesario) y la acumulación final.

3.5) Quita el comentario para activar `COSTOSA`, evalúa el programa `vector_3_2.c` en la cola de karen, y completa la tabla con 4 decimales.

$n = 1\ 200\ 000$				
Número de procesos	2	4	6	8
Coste Secuencial (tSec)	0.1127	0.1128	0.1126	0.1126
Coste Paralelo (tPar)	0.0567	0.0286	0.0205	0.0147
Incremento Paralelo (tSec vs tPar)	1.9862	3.9357	5.4695	7.6411
Coste Distribución (tDis)	0.0352	0.0550	0.0618	0.0659
Incremento Global (tSec vs (tPar + tDis))	1.2253	1.3485	1.3661	1.3967

Examina con detalle los valores y justifica los resultados.

Hacer las comunicaciones paralelas, sean con el número que sean dan mejor resultado siempre, eso si cuantas más hebras hay el incremento paralelo mejora con diferencia.
Pero al ver el incremento global (teniendo en cuenta el coste de distribución), vemos que el incremento de dos en dos en las hebras no es muy alto.

3.6) Quita el comentario para activar `COSTOSA`, evalúa el programa `vector_3_3.c` en la cola de karen, y completa la tabla con 4 decimales.

$n = 1\ 200\ 000$				
Número de procesos	2	4	6	8
Coste Secuencial (tSec)	0.1130	0.1133	0.1133	0.1129
Coste Paralelo (tPar)	0.0567	0.0290	0.0194	0.0151
Incremento Paralelo (tSec vs tPar)	1.9908	3.8978	5.8147	7.4677
Coste Distribución (tDis)	0.0273	0.0583	0.0623	0.0680
Incremento Global (tSec vs (tPar + tDis))	1.3443	1.2972	1.3828	1.3577

Examina con detalle los valores y justifica los resultados.

Hacer las comunicaciones paralelas, sean con el número que sean dan mejor resultado siempre, eso si cuantas más hebras hay el incremento paralelo mejora con diferencia. Pero al ver el incremento global (teniendo en cuenta el coste de distribución), vemos que el incremento de dos en dos en las hebras no es muy alto.

3.7) Quita el comentario para activar `COSTOSA`, evalúa el programa `vector_3_4.c` en la cola de karen, y completa la tabla con 4 decimales.

$n = 1\ 200\ 000$				
Número de procesos	2	4	6	8
Coste Secuencial (tSec)	0.1128	0.1118	0.1126	0.1127
Coste Paralelo (tPar)	0.0568	0.0287	0.0194	0.0148
Incremento Paralelo (tSec vs tPar)	1.9856	3.8874	5.8054	7.6055
Coste Distribución (tDis)	0.0296	0.0560	0.0632	0.0698
Incremento Global (tSec vs (tPar + tDis))	1.3054	1.3185	1.3638	1.3307

Examina con detalle los valores y justifica los resultados.

Hacer las comunicaciones paralelas, sean con el número que sean dan mejor resultado siempre, eso si cuantas más hebras hay el incremento paralelo mejora con diferencia. Pero al ver el incremento global (teniendo en cuenta el coste de distribución), vemos que el incremento de dos en dos en las hebras no es muy alto.

3.8) Comparando el tiempo de implementación de cada apartado y las tablas de resultados, responde a las siguientes preguntas referidas a transferencias en las que estén involucrados todos los procesos de un programa:

¿Es más sencillo utilizar comunicaciones punto a punto o comunicaciones colectivas?

¿Es más eficiente utilizar comunicaciones punto a punto o comunicaciones colectivas?

Es mucho más sencillo utilizar las comunicaciones colectivas, ya que simplifica y reduce mucho el código.

Es más eficiente hacer las operaciones colectivas también, ya que los procesos no es necesario que vayan en orden.