

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( MPLCOMM_WORLD, & numProcs );
    MPI_Comm_rank( MPLCOMM_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( MPLCOMM_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( MPLCOMM_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;
}

```


- 3.6) 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)	0,1127	0,1127	0,1127	0,112
Coste Paralelo (tPar)	0,0568	0,286	0,02112	0,01545
Incremento Paralelo (tSec vs tPar)	1,9834	3,9406	5,3353	7,2509
Coste Distribución (tDis)	0,02964	0,05637	0,06473	0,0681
Incremento Global (tSec vs (tPar + tDis))	1,3033	1,3264	1,3126	1,3407

Examina con detalle los valores y justifica los resultados.

Como en el anterior ejercicio, el cálculo escala bien con el número de procesos. Ahora la distribución utiliza Scatter, que es más eficiente (log N en lugar de N) porque envía menos mensajes.

El único problema es que el proceso 0 tiene que sumar todas las sumas parciales una a una.

- 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)	0,1132	0,1127	0,1127	0,1127
Coste Paralelo (tPar)	0,05673	0,02889	0,01941	0,01509
Incremento Paralelo (tSec vs tPar)	1,9961	3,9007	5,8083	7,465
Coste Distribución (tDis)	0,03299	0,05685	0,06374	0,06766
Incremento Global (tSec vs (tPar + tDis))	1,2622	1,3144	1,3555	1,3616

Examina con detalle los valores y justifica los resultados.

Esta versión aprovecha el reduce para calcular directamente la suma, evitando la suma secuencial de sumas parciales.

- 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?

Utilizar comunicaciones colectivas es más sencillo y eficiente porque reduce la cantidad de código que necesitamos evitando así errores y todas las funciones están optimizadas para resolver un problema concreto.

Puede ser más sencillo de comprender el funcionamiento con comunicaciones punto a punto pero una vez comprendido vale la pena utilizar comunicaciones colectivas.