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	la09_g

Tema 11. Comunicaciones Punto a Punto en MPI

Tema 12. Comunicaciones Colectivas en MPI

Se dispone de un código secuencial en `vectorPrimos.c` que calcula el número de primos que aparecen en un vector de enteros, y se desea paralelizarlo para aumentar sus prestaciones.

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

#define VECTOR_CORTO

// ================================================================
// Declaracion de prototipos locales.

int esPrimo( long long num );
void construyeVector( int * numElementos, long long ** vectorNumeros );

// ================================================================
int main( int argc, char *argv[] ) {
  int          i, miId, numProcs, numPrimosSec;
  double       t1, t2, ttSec;
  long long    *vectorNumeros;
  char         miNombre[ MPI_MAX_PROCESSOR_NAME ];
  int          dimVectorNumeros, lonMiNombre;

  // Comienza el codigo.
  MPI_Init( & argc, & argv );
  MPI_Comm_size( MPI_COMM_WORLD, & numProcs );
  MPI_Comm_rank( MPI_COMM_WORLD, & miId );

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

  // El proceso 0 construye el vector de numeros que se va a estudiar.
  if( miId == 0 ) {
    construyeVector( & dimVectorNumeros, & vectorNumeros );
    printf( "El vector consta de %d numeros.\n", dimVectorNumeros );
    printf( "\n" );
  }
  fflush( stdout );

  //
  // Implementacion secuencial (realizada en P0).
  // ===========================================
```

```
  //
  if(( miId == 0 ) && (argc == 1)) {
    t1 = MPI_Wtime();
    numPrimosSec = 0;
    for( i = 0; i < dimVectorNumeros; i++ ) {
      if( esPrimo( vectorNumeros[ i ] ) ) {
        numPrimosSec++;
      }
    }
    t2 = MPI_Wtime(); ttSec = t2 - t1;
    printf( "Implementacion secuencial.  Tiempo (s): %lf\n", ttSec );
    printf( "Numero de primos en el vector: %d\n", numPrimosSec );
    printf( "\n" );
  }
  fflush( stdout );
/*
  // Comprueba que hay al menos 2 procesos para la version paralela.
  // La version paralela necesita al menos 2 procesos:
  // 1 coordinador y al menos 1 trabajador.
  if( numProcs < 2 ) {
    fprintf( stderr, "ERROR: Debe haber al menos 2 procesos.\n" );
    MPI_Finalize();
    exit( -1 );
  }

  //
  // Implementacion Paralela.
  // ========================
  //
  int         numPrimosParLocal, numPrimosPar;
  long long   num;
  char        peticion;
  double      ttPar;
  MPI_Status  s;

  MPI_Barrier( MPI_COMM_WORLD );
  t1 = MPI_Wtime();
  numPrimosPar = 0; numPrimosParLocal = 0;

  // Codigo del Ejercicio 1 o del Ejercicio 2
  // ...

  MPI_Barrier( MPI_COMM_WORLD );
  t2 = MPI_Wtime(); ttPar = t2 - t1;
  if( miId == 0 ) {
    printf( "Implementacion paralela.    Tiempo (s): %lf\n", ttPar );
    if (argc == 1)
      printf( "Implementacion paralela.    Incremento: %lf\n", ... );
    printf( "Numero de primos en el vector: %d\n", numPrimosPar );
  }
*/
  // El proceso 0 destruye el vector de primos.
  if( miId == 0 ) {
    free( vectorNumeros );
  }

  // Fin de programa.
  if( miId == 0 ) {
    printf( "\n" ); fflush( stdout );
  }
  MPI_Barrier( MPI_COMM_WORLD );
```

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

// ================================================================
int esPrimo( long long num ) {
  int primo;
  if( num < 2 ) {
    primo = 0;
  } else {
    primo = 1;
    long long i = 2;
    while( ( i < num )&&( primo ) ) {
      primo = ( num % i != 0 );
      i++;
    }
  }
  return( primo );
}

// ================================================================
void construyeVector( int * numElementos, long long ** vectorNumeros ) {
  int       numElems;
  long long  * vecNums;

#ifdef VECTOR_CORTO
  // Vector corto para pruebas cortas.
  numElems = 20;
  vecNums  = ( long long * ) malloc( numElems * sizeof( long long ) );

  vecNums[  0 ] = 3LL;  vecNums[  1 ] = 15LL; vecNums[  2 ] =  2LL;
  vecNums[  3 ] = 4LL;  vecNums[  4 ] =  4LL; vecNums[  5 ] = 27LL;
  vecNums[  6 ] = 7LL;  vecNums[  7 ] = 17LL; vecNums[  8 ] = 13LL;
  vecNums[  9 ] = 4LL;  vecNums[ 10 ] =  4LL; vecNums[ 11 ] = 21LL;
  vecNums[ 12 ] = 4LL;  vecNums[ 13 ] =  8LL; vecNums[ 14 ] =  3LL;
  vecNums[ 15 ] = 4LL;  vecNums[ 16 ] = 19LL; vecNums[ 17 ] =  4LL;
  vecNums[ 18 ] = 5LL;  vecNums[ 19 ] =  7LL;

#else
  // Vector largo para pruebas largas.
  numElems = 100;
  vecNums  = ( long long * ) malloc( numElems * sizeof( long long ) );

  vecNums[  0 ] = 2097593LL;   vecNums[  1 ] = 4LL;
  vecNums[  2 ] = 16785407LL;  vecNums[  3 ] = 27644437LL;
  vecNums[  4 ] = 16785407LL;  vecNums[  5 ] = 27LL;
  vecNums[  6 ] = 17LL;        vecNums[  7 ] = 27644437LL;
  vecNums[  8 ] = 16785407LL;  vecNums[  9 ] = 4LL;
  vecNums[ 10 ] = 4LL;         vecNums[ 11 ] = 2097593LL;
  vecNums[ 12 ] = 4LL;         vecNums[ 13 ] = 4LL;
  vecNums[ 14 ] = 27644437LL;  vecNums[ 15 ] = 2097593LL;
  vecNums[ 16 ] = 4LL;         vecNums[ 17 ] = 7LL;
  vecNums[ 18 ] = 2097593LL;   vecNums[ 19 ] = 16785407LL;
  vecNums[ 20 ] = 4LL;         vecNums[ 21 ] = 2097593LL;
  vecNums[ 22 ] = 27644437LL;  vecNums[ 23 ] = 4LL;
  vecNums[ 24 ] = 27644437LL;  vecNums[ 25 ] = 2097593LL;
  vecNums[ 26 ] = 4LL;         vecNums[ 27 ] = 27644437LL;
  vecNums[ 28 ] = 2097593LL;   vecNums[ 29 ] = 16785407LL;
  vecNums[ 30 ] = 5LL;         vecNums[ 31 ] = 16785407L;
  vecNums[ 32 ] = 4LL;         vecNums[ 33 ] = 2097593LL;
```

```
  vecNums[ 34 ] = 27644437LL;  vecNums[ 35 ] = 27LL;
  vecNums[ 36 ] = 16785407LL;  vecNums[ 37 ] = 7LL;
  vecNums[ 38 ] = 27644437LL;  vecNums[ 39 ] = 16785407LL;
  vecNums[ 40 ] = 4LL;         vecNums[ 41 ] = 27644437LL;
  vecNums[ 42 ] = 16785407LL;  vecNums[ 43 ] = 4LL;
  vecNums[ 44 ] = 27644437LL;  vecNums[ 45 ] = 16785407LL;
  vecNums[ 46 ] = 5LL;         vecNums[ 47 ] = 16785407LL;
  vecNums[ 48 ] = 16785407LL;  vecNums[ 49 ] = 16785407LL;
  vecNums[ 50 ] = 4LL;         vecNums[ 51 ] = 2097593LL;
  vecNums[ 52 ] = 4LL;         vecNums[ 53 ] = 2097593LL;
  vecNums[ 54 ] = 27644437LL;  vecNums[ 55 ] = 27LL;
  vecNums[ 56 ] = 16785407LL;  vecNums[ 57 ] = 7LL;
  vecNums[ 58 ] = 27644437LL;  vecNums[ 59 ] = 16785407LL;
  vecNums[ 60 ] = 4LL;         vecNums[ 61 ] = 27644437LL;
  vecNums[ 62 ] = 16785407LL;  vecNums[ 63 ] = 4LL;
  vecNums[ 64 ] = 27644437LL;  vecNums[ 65 ] = 16785407LL;
  vecNums[ 66 ] = 5LL;         vecNums[ 67 ] = 4099LL;
  vecNums[ 68 ] = 1025LL;      vecNums[ 69 ] = 4LL;
  vecNums[ 70 ] = 4LL;         vecNums[ 71 ] = 2097593LL;
  vecNums[ 72 ] = 16785407LL;  vecNums[ 73 ] = 27644437LL;
  vecNums[ 74 ] = 16785407LL;  vecNums[ 75 ] = 27LL;
  vecNums[ 76 ] = 17LL;        vecNums[ 77 ] = 7LL;
  vecNums[ 78 ] = 27644437LL;  vecNums[ 79 ] = 4LL;
  vecNums[ 80 ] = 4LL;         vecNums[ 81 ] = 2097593LL;
  vecNums[ 82 ] = 4LL;         vecNums[ 83 ] = 4LL;
  vecNums[ 84 ] = 27644437LL;  vecNums[ 85 ] = 2097593LL;
  vecNums[ 86 ] = 4LL;         vecNums[ 87 ] = 7LL;
  vecNums[ 88 ] = 2097593LL;   vecNums[ 89 ] = 16785407LL;
  vecNums[ 90 ] = 5LL;         vecNums[ 91 ] = 2097593LL;
  vecNums[ 92 ] = 4LL;         vecNums[ 93 ] = 4LL;
  vecNums[ 94 ] = 2049LL;      vecNums[ 95 ] = 27LL;
  vecNums[ 96 ] = 4099LL;      vecNums[ 97 ] = 7LL;
  vecNums[ 98 ] = 4LL;         vecNums[ 99 ] = 2097593LL;
#endif

  // Devuelve el vector y su dimension.
  * numElementos = numElems;   * vectorNumeros = vecNums;
}
```

1 A partir del código anterior, desarrolla una versión paralela (`vectorPrimos_Ej1.c`) en la que sólo intervengan dos procesos: el proceso 0 que dispone de todos los números a procesar, y el proceso 1 que realizará todo el procesamiento, ya que sólo éste ejecutará la función `esPrimo`. Aunque esta versión paralela no será nada eficiente, será muy útil como una primera aproximación.

En esta implementación, **el proceso 0 envía todos los números** que desea evaluar al proceso 1 de uno en uno, es decir, **cada mensaje solo incluye un número**. Una vez se hayan enviado todos los números al proceso 1, **el proceso 0 debe enviar un mensaje especial de terminación (mensaje envenenado)** para indicar al proceso 1 que debe terminar. Cuando el proceso 1 haya terminado de procesar todos los números, debe envíar al proceso 0 el número de primos encontrado, y éste lo imprimirá en pantalla. Este funcionamiento se resume como sigue:

```
// Ejercicio 1
// El proceso 0 envia todos los numeros y un veneno
// ...
// El proceso 1 recibe numeros hasta que recibe un veneno
// ...
// El proceso 1 envia cuantos primos habia al proceso 0, que lo recibe
// ...
```

Existen dos detalles que hay que considerar en la implementación de la versión paralela. En primer lugar, decir que el tipo base de los elementos del vector es `long long`, es decir, un entero de 8 octetos, y que su correspondiente tipo en MPI es `MPI_LONG_LONG_INT`, Respecto de la comunicación, los mensajes donde se incluyen números a procesar irán marcados con la etiqueta 22, mientras que el mensaje envenenado que avisa al trabajador (proceso 1) que debe terminar, irá marcado con la etiqueta 33.

No olvides comprobar que el número de primos obtenido por la versión paralela coincide con el obtenido por la versión secuencial.

Para facilitar la implementación, se valida su funcionamiento con un vector corto con números no muy grandes. Por tanto, asegúrate de que al principio del código aparece la siguiente línea:

```
#define VECTOR_CORTO
```

Escribe a continuación la parte de tu código que realiza la implementación paralela.

Ten en cuenta que el programa debe funcionar para cualquier número de procesos.

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

2 Crea una nueva versión paralela (`vectorPrimos_Ej2.c`) en la que se emplee un número de procesos trabajadores igual al número de procesos empleados en el programa MPI menos uno.

Como la labor de averiguar si un número es primo tiene un coste muy heterogéneo (en unos casos es muy rápido y en otros le puede costar muchísimo), se propone emplear una distribución dinámica, con un único proceso coordinador (el proceso 0) y `(numProcs - 1)` procesos trabajadores. En esta distribución, el proceso 0 no participará en el procesamiento de los elementos del vector, sino que se limitará a distribuir las tareas a los procesos trabajadores **bajo demanda**. Para ello, el coordinador asignará un nuevo elemento del vector a un trabajador, cuando reciba de éste una nueva petición.

A continuación se describe con un poco más de detalle las funciones de los dos tipos de procesos:

- **Proceso coordinador**: Este es el papel del **proceso 0**, el cual se encarga de recibir peticiones de trabajo y contestarlas.

  El coordinador siempre debe estar a la espera de recibir peticiones de cualquier otro proceso. Cuando le llegue una petición de un trabajador, el coordinador debe responder al mismo trabajador enviándole un nuevo número del vector para que sea examinado.

  El coordinador puede averiguar el proceso origen de una petición a través de la variable de tipo `MPI_Status` que es uno de los parámetros de la operación de recepción.
- **Procesos trabajadores**: Este es papel de todos los procesos que no son el proceso 0, los cuales se encargan de enviar peticiones al coordinador, recibir el trabajo y procesarlo.

  Para pedir un número al coordinador, el proceso trabajador debe enviarle una petición. Posteriormente, deberá procesar la tarea recibida, antes de realizar una nueva petición. Un proceso trabajador finaliza su labor cuando recibe un mensaje envenenado.

La petición de un número por parte de un trabajador se realiza mediante el envío de un único dato (con cualquier valor) del tipo `char` (su correspondiente tipo en MPI es `MPI_CHAR`). Posteriormente, el coordinador responde con el envío del trabajo asignado, es decir, le envía un único número entero del tipo `long long`.

El coordinador irá enviando, de uno en uno, todos los números de su vector hasta que se acaben. Entonces, el coordinador debe enviar **un mensaje envenenado a cada uno de los procesos trabajadores** para indicarles que deben terminar. Este envío se realiza **como respuesta a la petición de trabajo** de los trabajadores.

Para señalar un mensaje envenenada, el proceso coordinador utilizará la etiqueta 44, mientras que para el resto de mensajes utilizará la etiqueta 55. En todos los casos, el proceso coordinador enviará un número, pero aquellos que se incluyen en un mensaje envenenado no serán procesados, por lo que se puede enviar cualquier valor, como por ejemplo, el primer elemento del vector.

Para las peticiones y el reparto de los números deben emplearse únicamente operaciones de **comunicación punto a punto**. En cambio, para la acumulación final del número de primos de todos los procesos deben emplearse operaciones de **comunicación colectiva**.

Este funcionamiento se puede resumir como sigue:

```
// Ejercicio 2
// El proceso 0 recibe peticiones y responde con numeros o venenos
// ...
// Los procesos trabajadores envian peticiones y reciben numeros o un veneno
// ...
// Todos los procesos colaboran para obtener el numero de primos
// ...
```

Para detectar errores más fácilmente, es obligatorio emplear en este ejercicio la rutina `MPI_Ssend` en lugar de la rutina habitual `MPI_Send`. Su funcionamiento y sus parámetros son idénticos, pero su diferencia estriba en que `MPI_Ssend` es un envío bloqueante, mientras que `MPI_Send` puede no serlo, dependiendo del espacio disponible. Al utilizar una rutina bloqueante, resulta más sencillo detectar si existen mensajes enviados y que nadie ha recibido.

Escribe a continuación la parte de tu código que realiza la implementación paralela.

3 Una vez se compruebe que funciona correctamente el programa anterior, modifícalo para que emplee la rutina `MPI_Send` en lugar de la rutina `MPI_Ssend`. La primera es más eficiente pero la detección de errores es más compleja.

Comprueba que el programa continúa funcionando correctamente.

Si, el programa sigue funcionando correctamente.

4 Completa la siguiente tabla con la última versión paralela.

Desarrolla el programa en el ordenador del aula, mientras que la producción de tiempos e incrementos debes realizarlas en karen. Redondea los tiempos dejando sólo tres decimales y redondea los incrementos dejando dos decimales.

Para la obtención de los tiempos es conveniente que el programa procese un vector grande. Para ello, asegúrate de que al principio del código aparece la siguiente línea:

```
#undef VECTOR_CORTO
```

Ten en cuenta que en karen **el código secuencial sólo debe ser ejecutado en una única ejecución**. Es muy importante que el código secuencial no sea ejecutado para cada ejecución (para 2 procesos, para 4 procesos, etc.) para de esa forma agilizar las ejecuciones y no saturar la cola de trabajos. Para ello, simplemente debes añadir un parámetro cuando ejecutes el programa.

Implementación	Tiempo	Incremento
Secuencial	11.256	—
Paralela con 2 procesos	10.989	1.02
Paralela con 4 procesos	4.008	2,80
Paralela con 6 procesos	2.614	4,30
Paralela con 8 procesos	2.001	5,62

Justifica los resultados obtenidos.

El vector analizado al ser muy largo, la implementación secuencial le cuesta analizar cada dato.
Al repartir el trabajo entre varios procesos, el tiempo de ejecución se reduce considerablemente.
A medida que van creciendo el número de procesos disminuye el tiempo hasta un tope.

5 ¿Cuál es el máximo incremento que se puede conseguir?

Alrededor de 1'30'' y 1'50'' ya que aunque los procesos tienen cada uno menos carga, el proceso 0 sigue teniendo que enviar la información a los demás procesos y ese tiempo no se reduce.

Nota: Esta entrega forma parte de la evaluación de la asignatura. Debe ser guardado por el estudiantado junto con el resto de entregas en una carpeta. El profesorado podrá pedir al estudiantado que le entregue dicha carpeta en cualquier momento.