2

Se pide escribir un programa en lenguaje C y utilizando MPI que, a partir de un array de números enteros, calcule la cantidad de estos números que son divisibles por cada número del intervalo [1,9]. De esta forma, se debe calcular la cantidad de números divisibles por 1, la cantidad de números divisibles por 2, etc.

El programa pedido deberá distinguir entre dos tipos de procesos: proceso *master* y procesos *worker*. En cada ejecución existirá un único proceso *master* y, al menos, dos procesos *worker*.

El proceso *master* debe distribuir el contenido del array de números enteros a procesar, el cual será **generado aleatoriamente** utilizando la función `createNumbers()`. El tamaño de este array viene determinado por el valor de la constante `MAX_NUMBERS`. Los procesos *worker* realizarán el cómputo para calcular cuántos de estos números son divisibles por los números del intervalo [1, 9]. Al finalizar el programa, el proceso *master* deberá imprimir por pantalla el resultado (ver ejemplos al final del enunciado).

La ejecución por la línea de comandos se realiza de la siguiente forma:

```
mpiexec -hostfile machines -np numProc progMPI
```

donde `machines` es el fichero con las direcciones IP de los ordenadores en los cuales se ejecutará el programa, `numProc` es el número de procesos que intervienen en la ejecución del programa y `progMPI` es el fichero ejecutable del programa.

Consideraciones a tener en cuenta:

- El sistema distribuido donde se ejecutará el programa es heterogéneo. Es decir, los recursos de cada ordenador pueden ser diferentes.
- El array de números enteros se genera en la función *main*, no es necesario volver a generar los números aleatorios en la función `executeMaster()`.
- Se tendrán en cuenta, de forma positiva, las soluciones que opten por un algoritmo dinámico para repartir la carga. En este caso, la constante GRAIN indica el número máximo de elementos que el proceso *master* puede enviar a un proceso *worker* en cada iteración del reparto de carga.
- Se puede suponer que `(GRAIN * (numProc-1)) <= MAX_NUMBERS`.
- No se permite enviar el contenido completo del array generado por el proceso *master* a todos los procesos *worker*.

Se pide implementar las dos funciones descritas a continuación:

```
void executeMaster (int *array, int numProc);
```

Esta función deberá ser invocada por el proceso *master*. El parámetro *array* contiene los números que deben procesarse, mientras que *numProc* indica el número total de procesos que intervienen en la ejecución.

```
void executeWorker ();
```

Esta función deberá ser invocada por los procesos *worker*.

La siguiente porción de código muestra la estructura de la función principal del programa.

```
/** Amount of numbers to be processed */
#define MAX_NUMBERS 15

/** Array size to allocate the results */
#define RESULTS_SIZE 9

/** Workload to be processed by each worker */
#define GRAIN 4

/** Master process */
#define MASTER 0

/** End-of-processing flag */
#define END_OF_PROCESSING -1


// Funtion prototypes
void createNumbers (int* vector, int maxNum);
void executeMaster (int *array, int numProc);
void executeWorker ();

int main(int argc, char *argv[]){

   int numProc, rank;
   int totalNum;
   int *array;

      // Init MPI
      MPI_Init (&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &numProc);

      // Check
      if (numProc < 3){
         printf ("Wrong number of parameters\n");
         MPI_Abort (MPI_COMM_WORLD, 0);
      }

      // Check the number of arguments
      if (argc != 1){
         printf ("Wrong number of parameters\n");
         MPI_Abort (MPI_COMM_WORLD, 0);
      }

      if (GRAIN*(numProc-1) > MAX_NUMBERS){
         printf ("Wrong number\n");
         MPI_Abort (MPI_COMM_WORLD, 0);
      }

      // Master process
      if (rank == MASTER){

         // Allocate memory and create random numbers
         array = (int*) malloc (MAX_NUMBERS * sizeof(int));
         createNumbers (array, MAX_NUMBERS);

         // Execute Master!
         executeMaster (array, numProc);
      }

      // Worker process
      else{
         executeWorker ();
      }

      // End MPI nvironment
      MPI_Finalize();

  return 0;
}
```