Parallel Computing
MPI Programming Assignment

In this lab you will write MPI code to generate the prime numbers between 2 and N (inclusive) and test scalability and performance.

General notes:

- The name of the source code file is: primes.c
- Write your program in such a way that to execute it I type: *mpiexec -n x ./primes N*
  Where N is a positive number bigger than 2 and less than or equal to 10,000,000 (10Millions). "x" is the number of processes.
- The output of your program is a text file *N*.txt (N is the number entered as argument to your program).
  For example, if I type: mpiexec -n 2 ./primes 10
  The output must be a text file with the name 10.txt and that file contains: 2 3 5 7
  You need to put a single space between each two prime numbers in the file. The prime numbers in the file must be ordered.
- You can assume that we will not do any tricks with the input (i.e. We will not deliberately test your program with wrong values of N).
- As a way to help you, the following table contains the number of prime numbers below x.

x	number of primes
10	**4**
100	**25**
1,000	**168**
10,000	**1,229**
100,000	**9,592**
1,000,000	**78,498**
10,000,000	**664,579**

Even though there are several algorithms for generating prime numbers, some are even more efficient than this one, you will implement the following algorithm for generating prime numbers.
This is the sequential version.

Input: N
1. Generate all numbers from 2 to N.
2. First number is 2, so remove all numbers that are multiple of 2 (i.e. 4, 6, 8, … N). Do not remove the 2 itself.
3. Following number is 3, so remove all multiple of 3 that have not been removed from the previous step. That will be: 9, 15, … till you reach N.
4. The next number that has not been crossed so far is 5. So, remove all multiple of 5 that have not been crossed before, till you reach N.
5. Continue like this till floor((N+1)/2).
6. The remaining numbers are the prime numbers.

Example:

Suppose N = 20

floor of (20+1)/2 = 10 ← where we stop.

Initially we have:
2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

Let's cross all multiple of 2 (but leave 2): [4 is crossed but the line is not shown due to the way 4 is written]
2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13, ~~14~~, 15, ~~16~~, 17, ~~18~~, 19, ~~20~~

Next number is 3, so we cross all multiple of 3 that have not been crossed:
2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ~~18~~, 19, ~~20~~

Next number that has not been crossed is 5, so we will cross multiple of 5 (i.e. 10, 15, and 20). As you see below, they are all already crossed. Note that 5 itself is not crossed.
2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ~~18~~, 19, ~~20~~

Next number that has not been crossed is 7, so we will cross multiple of 7 (i.e. 14). As you see below, they are all already crossed.
2, 3, ~~4~~, ~~5~~, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ~~18~~, 19, ~~20~~

The next number that has not been crossed is 11. This is bigger than 10, so we stop here.

The numbers that have not been crossed are the prime numbers:
2, 3, 5, 7, 11, 13, 17, 19

The file that your program generates is 20.txt and looks like (only spaces between each two numbers, no commas):

2 3 5 7 11 13 17 19

Your MPI Implementation

- You need to design your program in such a way as to distribute an equal amount of work, as much as possible, among the processes. If we have two processes, for example, then the first one will take care of the first (N-1)/2 numbers. That is, to find the prime numbers in them. And the second process will take care of the second half. If we have t processes, then process 0 will take care of the first (N-1)/t, process 1 will take the following (N-1)/t, etc. Of course, you need to take care of the case when (N-1) is not divisible by t. This may result in a process taking care of a few extra numbers, which is not a problem and won't affect work balance a lot.
- Once each process gets its assigned numbers, it will apply the algorithms described above to get the prime numbers in its range. This is a bit tricky. So, read the next point.
- Suppose we have three processes and N = 100. Then each process needs to check for primality by checking the remainder of the divisions by 2, then by 3, … till floor((100+1)/2) = 50 as explained earlier. Each process will be responsible for ceil((100+1)/3) = 34 numbers. Remember that 34*3 = 102. So, the last process will do a bit less work. *Each* process will check the remainder of the divisions by 2 to 50. You can optimize a little bit here because all the numbers below the divisor do not need to be divided. For example, any number below 30, for instance, does not need to be divided by 30. And all the numbers that have been crossed and removed from an earlier iteration do not need to be divided.
- The prime numbers must be listed in N.txt in order.
- Feel free to make any optimizations regarding the data structure, communication, and computation. However, you need to follow the main algorithm mentioned above.

How to measure the performance and scalability of your code?

Your code must be timed, using MPI_Wtime() as explained in slide 35 from the lecture "MPI-3". Let process 0 write the data to the file and print the time measured on the screen. Measure the time taken by your code after the initialization step (i.e., MPI_Init …), and end the timing before writing to the file. That is, the timing must include all computations, all communications, and dynamic allocation of any data structure you may use.

To see how efficient your implementation is, you need to compare the performance with a different number of processes.
Therefore, do the following:

1) Prepare three tables as follows (rows represent #processes and columns represent N):

N→	1000	10,000	100,000	1000,000	10,000,000
#processes 1					
2					
5					
10					

2) So, one table for times (in seconds), one for speedup (relative to execution with one process), and the third is for efficiency (speedup/ #processes).
3) In these tables, make N = 1000, 10000, 100000, 1000000, and 10000000

4) The number of processes = 1, 2, 5, and 10
5) Answer the following questions:
   a) What is the pattern that you see in speedup as number of processes increase, for different problem sizes?
   b) Explain why we get the pattern you explained in the question above.
   c) As the number of processes increases, how is the efficiency affected, for different problem sizes?
   d) Justify your answer to question c above.

What do you have to submit:
A single zip file. The file name is your lastname.firstname.zip
Inside that zip file you need to have:

- primes.c
- pdf file containing the tables and your answers.

Submit the zip file through Brightspace as usual.
Note: As a way to help you, we provide a list of the first 1 million prime numbers for you to check your implementation for correctness. This file is for checking your results, but it is not formatted in the correct format required for the file N.txt

Enjoy!