Parallel Computing
Lab Assignment 1

In this lab you will write MPI code to find numbers divisible by a given number x in the range between A and B (inclusive) and test scalability and performance. For example: find numbers divisible by 3 in the range between 2 and 10000,

General notes:

- The name of the source code file is: netID.c where netID is your NetID.
- You compile with *mpicc -std=c99 -Wall -o checkdiv netID.c*
- To execute it: mpiexec -n *p ./checkdiv A B x*
  Where A and B are positive numbers bigger than 1 and less than or equal to 100,000,000 (100 millions); and x is a positive number between 2 and 1000,000 (inclusive).
- The output of your program is a text file *B*.txt
  For example, if I type: mpiexec -n *4* ./checkdiv 2 10 3
  The output must be a text file with the name 10.txt and that file contains:
  3
  6
  9
  one number per line.
- **The numbers must be written in the file in ascending order and one number per line** like this example above.
- You program must also print on screen:
  times of part1 = num1 s
  times part2 = num2 s
  num1 and num2 are double precision floating points representing the time taken in seconds. To know what are part1, etc, continue reading. They are defined next page.
- 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, negative, float, non-numeric, etc).
- We are providing you with a file: skeleton_lab1.c that can help you start. You can use it and just fill-in the gaps.

The parallel code:

Assume you have p processes.

- All processes will get the arguments of the main() function right away, without the need to communicate them.
- Part 1:
  - The range A to B (i.e. B-A+1 numbers) will be divided among the processes with the last process (in terms of rank) taking slightly extra/less work if the range is not divisible by the range. So, if A = 2 and B = 10 and we have two processes, process 0 will work on range 2 to 6 and process 1 will work on the range 7 to 10, getting one less item than proc.
  - Then, each process works on its range and generates its own list.
  - Each process sends its list to process 0.

- Part 2:
    - Finally, process zero creates the file N.txt and writes all the numbers there.

How to measure the performance and scalability of your code?

To see how efficient your implementation is, you need to compare against a version with one process. Therefore, we will use several methods.

The program is divided into two main parts, as we saw above.
part 1 where each process generates its list of divisible numbers and sends it to process 0.
part 2 where process 0 write the data to the disk.

To measure the time of a piece of code, we will use MPI_Wtime() as follows (Note: The following is a pseudo-code. You need to write proper one with correct declarations and header files):

```
start1 = MPI_Wtime();
   part1
end1 = MPI_Wtime();
reduction operation, MAX to get largest end1-start1
start2 = MPI_Wtime();
   part2
end2 = MPI_Wtime();
```

When executing your full program, use the Linux *time* command. It will be used in graph 2 as will be shown in the section "The report" below. For example

```
time mpirun -n 4  ./checdiv 1000000 5
```

Your code must show some speedup, in part 2, relative to running with only one process.
Finally, you will report these numbers in the report as discussed below.

The report

For that report, to generate the graphs, assume x = 91.
We may test your program with several different numbers though.

Write a report that contains the following graphs.

Graph1:

- X-axis with values (100, 1000, 10000, 100000, 1 million, 10 millions, and 100 millions). These values represent B. Assume A = 2.
- The speedup (y-axis) (**time of part 1 with 1 process / time part 1 with p process**).
- For each number in the x-axis, draw three bars: speedup using two processes, four processes, and eight processes.

Graph 2:

- Same values in x-axis and same bars (2, 4, and 8 processes) as graph 1 but the y-axis is the overall speedup generated by the *time* command as described above. That command generates three numbers: user, system, and real. We want the *real* one. The y-axis will then be the real time for one process divided by the real time of p process where p = 2, 4, and 8.

What to submit:

A single zip file. The file name is your **netID.zip** where netID is your, well, NetID.
Inside that zip file you need to have:

- **netID.c**
- pdf file containing the two graphs the file name must be **netID_report.pdf**

Submit the zip file through Brightspace.

Enjoy!