**CSE-443/543: High Performance Computing**

**Homework #02**

Max Points: 86

**Objective**: The objective of this exercise is to:

* Gain experience developing a non-trivial parallel code using OpenMP
* Gain experience using the CSE C++ style guidelines
* Build experience with benchmarking parallel programs
* Build experience using spreadsheets for scientific data analysis
* Learn how to safely use pseudorandom number in a multi-threaded program
* Learn about functions in bash scripts
* Learn about the $RANDOM environment variable in bash
* Learn about how to evaluate expressions on the bash command line.

## Background

In class, you have learned about OpenMP and have done a lab to gain experience compiling and running programs in OpenMP. To build experience with OpenMP, you will be tackling a more difficult challenge – writing code from scratch that will be then implemented in parallel via OpenMP.

As you read in section 34.2.2 in the textbook, while all random number generator libraries provide a uniform pseudorandom number generator, not all of them provide a Gaussian distribution generator. Gaussian-distributed random variates are commonly used in simulation, modeling, queuing, etc. so having a way to generate them is important.

The [Box-Muller transform](https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform) is one way of transforming a pair of uniformly distributed random numbers into a pair of Gaussian-distributed numbers. In its basic form, two Gaussian-distributed random numbers (G1 and G2) are generated from two uniformly-distributed random numbers (U1 and U2) via these equations:

If this transform is repeated for many pairs of numbers, the resulting distribution will be Gaussian, with mean 0 and standard deviation 1. To generate a distribution with mean and standard deviation , each number is transformed in this way:

While this transformation is valid, it relies on the computation of the sine and cosine, which could be slow depending on how they are implemented. (Remember Lab 02?) An alternative implementation, attributed to George Marsaglia, can avoid computing the trigonometric functions at the expense of throwing away some results. In this approach, you compute

then discard *U1* and *U2* if *W* is > 1. Once there are two valid values for *W* and the *V*s, the Gaussian-distributed numbers are computed as

For this homework, you will first develop and benchmark serial versions of both the Box-Muller and Marsaglia implementations. Once these are working and have been debugged and tested, you will move on to parallelizing them using OpenMP.

To get you started, I am giving you a program called rng\_omp.cpp. This program implements a thread-safe, system-provided random number generator drand48\_r. You should study this program to see how it is implemented, paying particular attention to:

1. The random number state (state), stored in an instance of a drand48\_data structure. Note that state is declared as a global variable and is noted as being threadprivate. Make sure you understand why.
2. The initialization of state for different threads, performed in the initRNG method.
3. The testme method, which generates a given number of random numbers and sums them.

If you compile and run this with command line arguments 40 1234567 (the number of RNs to generate, and the RNG seed), and run it for 4 threads, you should get the answer 19.4018. Because the initialization routine uses the thread number to generate a unique seed for each thread, the result you get will depend on the number of threads used.

Another problem that arises in many situations, especially in high-performance computing, is how to provide RNG seeds to programs. One approach that we have previously used is to use the built-in function [time](https://linux.die.net/man/2/time). This returns the number of seconds that have elapsed since the Epoch (Midnight on 1/1/1970). See [Lab 02](https://drive.google.com/drive/folders/1n36vDfk2V5CgIAhp7R99cS_3pdOyCoCQ) for example. A problem with this is that if you are running many jobs, there’s no guarantee that they won’t start within the same second. In that case, two or more jobs could have the same RNG seed which is not desirable.

Bonus material: The time function is the cause of a pending problem with Linux-based systems. time returns a value of type time\_t which is equivalent to a 32-bit signed integer whose maximum value is 2,147,483,647. On January 19, 2038, the number of elapsed seconds since the Epoch will surpass the maximum value that can be stored in a signed integer and the time function will begin returning negative numbers, starting at -2,147,483,648 and then moving forward. This will break many programs, and others will function, but display the date as being in December 1901. This is known as the [Year 2038 Problem](https://en.wikipedia.org/wiki/Year_2038_problem) and it’s likely that at least some of you will have to remediate it as part of your jobs in the future.

Another problem with automatically seeding a random number generator with values from time is that during development and debugging it is desirable to repeat runs with the *same sequence* of random numbers. For this reason, it is a good programming practice to provide the random number generator seed as *input* to a program.

If you are running a program only a few times, coming up with a few random numbers to use as seeds isn’t a problem. However, when a program will be run multiple times, or when automation is desired, having a way to automatically generate seeds is important. I am giving you a starter bash shell script called Homework02-starter.bash that demonstrates how you can generate random number seeds. It introduces several new concepts so you may pick up a few tips by studying it:

1. How to write a function in bash. In this shell script, the function is called getSeed.
2. The bash internal let, which is used to evaluate arithmetic expressions acting on shell variables.
3. The Linux [expr](https://linux.die.net/man/1/expr) command, which can evaluate a richer set of expressions acting on shell variables.

You are welcome to use Homework02-starter.bash or the concepts in it to help with running your programs.

Your program will be computing the average and standard deviation of a set of numbers. There are several ways to implement the standard deviation in a way that only requires a single pass through the data. The one I am suggesting is the following:

1. As you generate data, have a variable that contains a running sum of the observations
2. Also keep a variable that contains a running sum of the squares of the observations
3. After generating the entire set of numbers, divide the running totals by the number of observations to get the average of the observations <O> and the average value of the square of the observations <O2>
4. Compute the variance as <O2> - <O>2
5. Computer the standard deviation as the square of the variance.

Note that this simple one-pass approach can have numerical stability issues if the variance is too small. We will be fine for this application, but in general there are [other more stable (and complex) approaches](https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance). Also note that this computes the population standard deviation, not the sample standard deviation, but with 20000000 observations per trial being generated with this program the difference is insignificant.

As with other labs/homeworks, your timing runs must be performed on the lab machines in BEN002. If you have access to another Linux system you can certainly do your development and testing on it. Recall that the computers are reserved during class times (1:15 – 2:35 and 2:50 – 4:10 on Mondays and Wednesdays). Most of the machines are assigned to two people and you should work together to coordinate access to the systems outside of class times to ensure that you don’t interfere with each other’s timing. There is no issue with two people being logged in while you’re doing development and testing.

Finally, one change that goes into effect starting with this homework. I am no longer asking you to include your shell scripts with your homework/lab submissions. I strongly encourage you to continue writing scripts to make your work easier, but if you choose to do everything manually that is now your prerogative.

Some comments and hints:

1. You should write a program called box-muller.cpp which implements the Box-Muller approach to generating Gaussian random numbers. It should have these properties/requirements:
   1. It should take a single, unsigned int command line argument which is the random number generator seed.
   2. You should include these preprocessor macro definitions:
      1. #define RUNLENGTH 20000000
      2. #define N 100

These are, respectively, the number of Gaussian numbers to generate in a single trial and the number of trials to perform.

* 1. It should include a method called doBoxMuller that generates the average and standard deviation of a set of RUNLENGTH Gaussian random numbers. Your definition of doBoxMuller should be:

void doBoxMuller(double \*ave, double \*stdDev, double mu, double sigma)

* 1. The doBoxMuller method should generate RUNLENGTH Gaussian random numbers with mean mu and standard deviation sigma. It should compute the average and standard deviation as described above.
  2. In this serial program, you should declare a global instance of a drand48\_data structure called state, and initialize this in your main method with your random number seed by using srand48\_r.
  3. You should set the average value of the Gaussian-distributed random numbers (mu) to be 42.0, and the standard deviation (sigma) to be 5.0.
  4. Your main method should have a loop running from 0 to N - 1 that invokes the doBoxMuller method. After each iteration it should write out the values of the average and standard deviation computed in the method.
  5. Your box-muller.cpp program should pass a cpplint.py test with no warnings or errors.
  6. You should compile box-muller.cpp using our standard options (-O3 -Wall –std=c++14)
  7. As a test, if you run your box-muller program with the seed 1234567 you should find the first average to be 42.0004 and the first standard deviation to be 4.99939.
  8. When your program is working correctly, compile it and run it 5 times, timing the runs with /usr/bin/time, and entering the timing results (CPU = User + System, and Elapsed, expressed as real numbers without “:”) into the Homework 02 Google spreadsheet. Each run should use a different random number seed. You may use the approach from Homework02-starter.bash for this.

1. Repeat Step 1, but this time implementing the Marsaglia approach. Your program should be called marsaglia.cpp and the method doMarsaglia. If you have implemented this correctly, when you run the marsaglia program with the seed 1234567 you should find the first average to be 42.0007 and the first standard deviation to be 5.00092.
2. Make copies of box-muller.cpp and marsaglia.cpp, calling them box-muller\_omp.cpp and marsaglia\_omp.cpp respectively.
3. Modify the \_omp versions of the program to use OpenMP to parallelize the for loop inside the doBoxMuller or doMarsaglia methods.
   1. You should initialize the random number generator using the approach given in the rng\_omp.cpp program.
   2. You should compile the \_omp programs using our standard options for OpenMP programs (-O3 -Wall –std=c++14 -fopenmp)
   3. To check for a correct implementation, when run with OMP\_NUM\_THREADS=1, and with a random number seed of 1234567 you should get the same values for the first average and standard deviation as for the serial programs.
   4. You should run each parallel program for OMP\_NUM\_THREADS equal to 1, 2, 3, 4, 5, 6, 7 and 8. For each value of the number of threads you should run the program 5 times, timing the runs with /usr/bin/time, and entering the timing results (CPU = User + System, and Elapsed, expressed as an real number without “:”) into the Homework 02 Google spreadsheet. As with the serial program, each run should use a different random number seed. You may use the approach from Homework02-starter.bash for this.
4. After entering your observations for Box-Muller and Marsaglia in the spreadsheet, compute the average CPU times, average elapsed times, and their respective errors in the Analysis portion of the Google spreadsheet, then compute the parallel performance metrics Sn, En, and Pn according to their definitions from [Session 12](https://drive.google.com/open?id=1noK7mMx14QScNMZ7DtF-wdjsF5g6YFvMOoNkeM7mqQY).
5. If your programs are working correctly, it should take approximately 55 minutes to do all of the timing runs for both programs. If the runs are taking much more than this, you likely have a problem in your spreadsheet or bash script. Make sure to do some test runs for the parallel code to ensure that the elapsed time is dropping as you increase the number of threads!

# Submit files to Canvas

When you complete the homework, download your Google spreadsheet file as a Microsoft Excel (.xlsx) file with the naming convention Homework02 – Results MUid.xlsx (example: Homework02 - Results - ferrenam.xlsx)

Then, submit the following files to Canvas:

1. The Microsoft Excel file you downloaded from Google Drive.
2. Your box-muller.cpp program.
3. Your box-muller\_omp.cpp program.
4. Your marsaglia.cpp program.
5. Your marsaglia\_omp.cpp program.