Intel MKL Library
STAT 624: Big Data and Databases
Prof: Dr. Schmiediche
Student: Carson Hanel

This is an introduction to highly efficient random number generation utilizing Intel's Math Kernel Library. It's just a quick and dirty overview before I actually do a deep dive performance benchmark for Intel's MKL for research purposes. I do research for the NSF and Texas A&M's IRL focusing on streaming and number generation in a distributed framework, and it seems Intel's MKL library is our lead competition. Here's a quick guide to get you going.

Running the code, evaluating speed

- **Library Import:** module load intel2018/b
- **Compile Code:** icc -o terra_mt terra_mt.c -O2 -xHost -lmkl_rt -lpthread -lm -ldl
- **Run Code:** time ./terra_mt
- **Runtime:** 2.534 seconds for 1 billion numbers, 1000 at a time.

Comparison with the GNU compiler

- **Compile Code:** gcc -o terra_mt terra_mt.c -O2 -xHost -lmkl_rt -lpthread -lm -ldl
- **Run Code:** time ./terra_mt
- **Runtime:** 3.342 seconds for 1 billion numbers, 1000 at a time.

Evaluation

While icc was notably faster, I'm surprised that gcc both was able to compile utilizing the same flags, and was still within a decent time frame of execution. Much better results and comparison than the ifort Fortran compiler results. Again, however, this reinforces the idea that Intel designs the most optimal compilers (well, for Intel hardware).

Comparison between RNG Methods

- **Generalized Feedback Shift Register Generator**
  - **Stream instantiation:** vslNewStream(&stream, VSL_BRNG_R250, 777);
  - **Runtime:** 3.402 seconds for 1 billion numbers, 1000 at a time.
- **59-bit Multiplicative Congruential Generator**
  - **Stream instantiation:** vslNewStream(&stream, VSL_BRNG_MCG59, 777);
  - **Runtime:** 2.486 seconds for 1 billion numbers, 1000 at a time.
- **Mersenne Twister**
  - **Stream instantiation:** vslNewStream(&stream, VSL_BRNG_MT19937, 777);
  - **Runtime:** 2.534 seconds for 1 billion numbers, 1000 at a time.

Mersenne Twister performance over varying batchloads

- **10 numbers at a time:**
  - **Runtime:** 19.802 seconds for 1 billion numbers, 10 at a time.
- **1,000 numbers at a time:**

    - **Runtime:** 2.564 seconds for 1 billion numbers, 1,000 at a time.
- **1,000,000 numbers at a time:**
    - **Runtime:** 2.882 seconds for 1 billion numbers, 1,000,000 at a time.

Final Evaluation

To explain the differences in speed, the first configuration is much slower because it prohibits vectorization from really being able to help out, and the function call to generate random numbers happens much more often. This interaction really extends the runtime. Now, for a billion at a time, the reason that it's much slower than 1,000 is due to temporal and spatial locality. For the million at a time scenario, the OS had to first, grab all of that RAM, the generator wrote an entire million, and then the work was allowed to be done. In the shorter case, as soon as 1,000 were generated, the consumer was allowed to work. This "producer-consumer" interaction allows for better vectorization and chunking of work, which is why the medium batch is the most optimal choice for both temporal and spatial locality and requirements.