CSCI 3366 (Parallel and Distributed Programming), Fall 2021
Homework 3

Credit: 55 points.

1 Reading

Be sure you have read, or at least skimmed, readings from the relevant updated appendices.

2 Overview

Your mission for this assignment is to improve the programs you wrote for Homework 1, to write versions in Java and OpenCL, and to measure their performance and accuracy more systematically.

3 What may need work

There are a couple of things that are typical sources of trouble with what students turn in for Homework 1.

The first thing to note is that calling the two standard C library functions sequence, `srand()` and `rand()`, is by far the most computationally intensive part of the calculation, so it's critical that this be done in a way that's effective and takes advantage of multiple UEs.

In a shared-memory environment, logically it should seem obvious that these functions can only work by saving some of state (possibly the previous value or possibly something larger that can be mapped to possible outputs), and all threads share the same value, leading to either the potential for race conditions if more than one thread at a time is allowed to access this state, or performance slowndowns if the library (or the programmer!) enforces one-at-a-time access. The latter might be acceptable in some situations, but not in when it's so big a part of the calculation. There *are* alternative library functions (`man drand48_r()` will tell you more), but then you have the problem that arises in a distributed memory environment:

In a distributed-memory environment, there isn't any shared state to cause trouble, but then every UE generates the same sequence, which means every UE computes the same result, which means the more UEs you have the less accurate result you get for a given number of points. You could just have each one generate the full sequence and pick out a unique-to-it subsequence, but that won't perform well either.

Instead this assignment asks you to just write your own RNG functions — my long-time collaborator and coauthor Dr. Mattson says it's something every computer scientist do at some point anyway.

4 Details

4.1 Thread-safe RNG

(5 points)

Your first step will be to write a thread-safe random number generator, i.e., one that can be called from multiple threads concurrently without ill effects. To keep this part manageable, I suggest

that you just use a simple technique, LCG (linear congruential generation). It's by no means good enough for cryptography, but it's been used in many library implementations over the years and is simple. The Wikipedia article has a pretty good discussion, but briefly:

This algorithm generates a pseudorandom sequence $x_0, x_1, x_2, \ldots$ from a seed S, constants a, c, and M, and a simple recurrence relation:

$$\begin{aligned} x_0 &= S \\ x_i &= (ax_{i-1} + c) \mod M, \text{for } i > 0 \end{aligned}$$

The Wikipedia article gives values used by many library implementation of this algorithm; to me the most attractive choice is the one cited for two POSIX functions, namely $a = 25214903917$, $c = 11$, and $M = 2^{48}$. (This seems attractive because — if I understand the discussion correctly — it will generate long sequences without duplicates (which we want), and values will be within the range of a 64-bit signed data types, which is available as `int64_t` in standard C and `Long` in Java.) Also, the `mod` part of the calculation is easily done by using bitwise AND with $2^{48} - 1$.) (Note that you will need to `#include stdint.h` to use `int64_t`.)

(If for some reason you want to try a different algorithm, check with me first — there may well be better choices, but there are probably worse choices too.)

You will need two implementations of whatever algorithm you choose, one in C and one in Java. Exactly how you package the algorithm is somewhat up to you, but you want functions analogous to `srand()` and `rand()`, and there needs to be some way to deal with the "state" of the sequence being generated (the current or next x_i) in a way that makes it possible for each thread to have its own state (rather than there being one hidden global state, as with `srand()` and `rand()`).

For C, what I think makes sense is to represent the saved state as an `int64_t` and define two functions that take a pointer to a state as a parameter:

- `void rand_set_seed(long seed, rand_state_t *state);`
- `int64_t rand_next(rand_state_t *state);`

You'll also want to define a constant, with something such as the following:

```
const int64_t RANDMAX = (1LL << 48) - 1;
```

(Note that this is $M - 1$.)

For Java, you'll probably want to define a class analogous to `java.util.Random`, but much simpler, with just a `RANDMAX` constant, a constructor, and a `next` method. I recognize that not all of you are especially fluent in Java, so to get you started here are a skeleton class (called `SimpleLCG`) and a test program that uses it: SimpleLCG.java, TestSimpleLCG.java. `TestSimpleLCG` gets a seed and number of samples from the command line. Run it without arguments and it will remind you what arguments it wants.

4.2 Revised sequential programs

The next step is to replace the current code for generating random numbers in two starter programs, one in C and one in Java, with your RNG code:

- C program: monte-carlo-pi.c. Also requires timer.h. (This is the starter code from Homework 1.)
- Java program: MonteCarloPi.java. (Note that the class this defines is in package `csci3366.hw3`, so it should go in a directory named `csci3366/hw3`.)

4.2.1 Code

(5 points)

Replace the current code for generating random numbers in the two starter sequential programs with calls to your RNG. (If you didn't already test your RNG code, you might temporarily put in some debug-print statements to be sure it's generating reasonable output.) The two programs (C and Java) should now produce the same output (except for execution time).

4.2.2 Results (accuracy)

(5 points)

(You only need to do this for one of your sequential programs, since they should give the same results.) Experiment until you find a seed that seems to give reasonable results, and then measure the relationship between accuracy (difference between the computed value of π and the constant as defined in the math library) and number of samples: Generate output for at least six different values of "number of samples" (I recommend starting with a medium-size number and then repeatedly doubling it, rather than increasing by a fixed amount). Plot the results, by hand or with whatever program you like. (I use `gnuplot`. Short introduction/example below.) You can repeat this for more than one seed and plot all sets of results if you like.

(It's probably worth noting that accuracy starts to max out well short of the number of points you need to measure performance in any meaningful way. I like this problem as an easy first one in the four environments, but it's just not very convincing as a useful application.)

4.3 Parallel programs

4.3.1 Code

(30 points)

Your mission for this step is to produce parallel programs for our four programming environments: C with OpenMP, C with MPI, Java, and C with OpenCL.

- For OpenMP and MPI, you should be able combine what you did for Homework 1 with what you did for the first step (sequential program with your own RNG).
- For Java, you'll have to figure out how to "parallelize" what you did for the first step, but you should be able to adapt the numerical integration example (on the "sample programs" page). As with the numerical integration example, your program should get the number of threads from an additional command-line argument.
- For OpenCL, again you'll have to figure out how to parallelize, but you should be able to adapt the numerical-integration example (on the "sample programs" page). Like that example, your program should take additional command-line arguments that let you vary what can be varied (number of work items, work group size).

So to recap, command-line arguments should be as follows:

- For OpenMP and MPI (same as for Homework 1): number of samples, seed. (The OpenMP program should get the number of threads from the `OMP_NUM_THREADS` variable and the MPI program from `mpirun`.)

- For Java: number of samples, seed, number of threads.
- For OpenCL: number of samples, seed, number of work items, factor that lets you vary workgroup size (to me a reasonable choice here is what I do in the numerical integration example, a factor by which to multiply the "preferred size").

As we noted in class, having all UEs (processes or threads) generate points using the same RNG and seed is not useful. You have two options for dealing with this:

- Use a different seed in each UE. As noted in class, simple methods of combining a "master seed" with UE ID (adding or multiplying them) may produce overlapping sequences, but figuring out how to avoid that is somewhat beyond the scope of this assignment.
- Arrange for each UE to generate only a part of the whole "random" sequence. It turns out that mathematically in principle this should be straightforward: If you want to split the above-described sequence among p UEs, you can do so by generating similar sequences in each UE, but with constants

$$\begin{aligned} a' &= a^p \mod M \\ c' &= c(a^p - 1)/(a - 1) \mod M \end{aligned}$$

  and starting the sequence for the i-th UE at element x_i of the original sequence. This technique is called "leapfrogging" since that's kind of what it does, with each UE generating just the elements of the original sequence it would have if they were assigned round-robin style. (I originally found this in a paper that no longer seems to be freely accessible, but it's repeated here. To me this seems like the right way to go, but it's more work, so I'll give extra credit for trying it. Hints and partial code for a C version below.

Hints for using leapfrogging:

- What I found to make the most sense was to package things up in a slightly different way:

  For Java, a class still makes sense, but I think its constructor should take two more arguments, the number of "streams" (UEs for us) and which stream this object is for. You could put the code to generate the modified constants in the constructor, and I think it's fairly straightforward to do and to get right if in computing the constants you use `BigInteger` for intermediate values and only convert to `Long`s at the end (when the " mod M" step gives you a result you know will fit).

  For C, I thought it made sense to make the "state" a struct and introduce one more function

  ```
  void rand_init_state(int p, int id, rand_state_t *state);
  ```

  that computes and saves the values for the modified constants.

- Computing the modified constants — there may be some way to do this without arbitrary-precision arithmetic, but I didn't think of one so chose to just use the GMP library (a GNU library so not standard, but widely available on UNIX-like systems). I didn't find this so easy so am willing to share most of what I came up with — I've left out a few parts of the code (look for "FIXME") to keep this from being too easy(?), but I'm also including a test program you can use to confirm that what you're doing works:

- leapfrog-lcg.h containing a `struct` and functions. (So you would use `#include "leapfrog-lcg.h"` to include this in your code.)
- test-leapfrog.c containing a test program.
- Makefile containing a make file that may be helpful. Note that if you don't use this you need to remember to compile/link with `-lgmp` to include the GMP library functions.

4.3.2 Results (accuracy)

(5 points)

(You only need to do this for one of your parallel programs, since they should give the same results for the same number of units of execution, where "units of execution" is threads for OpenMP and Java, processes for MPI, and work items for OpenCL.) Experiment until you find a seed and number of samples that seem to give good results, and then measure the relationship between accuracy (difference between the computed value of π and the constant as defined in the math library) and number of UEs. Generate output for at least six different values of "number of UEs" (I recommend powers of two, starting with one). (Since for OpenCL the number of work items has to be a multiple of the minimum work-group size, it might be interesting to make a second plot showing that minimum value and then several multiples of it.) Plot the results, again by hand or with whatever program you like.

4.3.3 Results (performance)

(5 points)

To get meaningful results for performance, you will likely need far more samples than are needed to give reasonably accuracy, though you can use whatever seed seemed to be most effective. Find a number of samples for which both of your sequential programs take at least 2 seconds, and measure execution times for both sequential programs and all four parallel programs. For the parallel programs, measure execution time using different numbers of UEs (start with one and double until you notice that execution time is no longer decreasing). I strongly encourage you to do this on the machines that to me seem most suitable in terms of being able to "scale up" to interesting numbers of UEs: For OpenMP and Java, that would be Dione, for MPI, the Pandora cluster, and for OpenCL, Deimos or one of the Atlas machines. You should do each measurement more than once; if you get wildly different results it probably means you are competing with other work on the machine and should try again another time or using another machine or machines.

Plot the results, again by hand or with whatever program you like:

- For the OpenMP, MPI, and Java programs, plot execution time versus number of UEs, and also show execution time for the sequential program in the same base language (C or Java).
- For the OpenCL program, do as for the others, but also show at least two sets of values for different work-group sizes.

4.4 Hints and tips

- Feel free to borrow code from any of the sample programs linked from the course sample programs page. This page also contains links to my writeups about compiling and running programs on the lab machines. The course "useful links" page has pointers to documentation on all four environments.

- You can develop your programs on any system that provides the needed functionality, but I will test them on the department's Linux classroom/lab machines, so you should probably make sure they work in that environment before turning them in.

5 A very little about `gnuplot`

I often talk about the plotting tool `gnuplot` in class. Here are files for a simple example along the lines of what you need to do for this assignment (plot parallel times as a function of UEs, also showing sequential time):

- Plot input file par.plotin.
- Data files seq-times.dat, par-1-times.dat, par-2-times.dat.

With all these files in a directory, the command `gnuplot < par.plotin` will generate a file `par-times.png` with the plot.

6 What to turn in and how

Turn in the following:

- All source code (your two RNG implementations, revised sequential programs, and parallel programs). Call them whatever you like, as long as it's clear what's what, but please have them get input from their environment and command-line arguments as discussed above.
- Plots (accuracy of sequential program(s), accuracy of parallel program(s), and performance of parallel programs).
- Input data for plots. A text file or text files is fine for this. Also say which machines you used for the performance measurements.

Submit your program source code, plots, and input data by putting them in your "turn-in" folder.

7 Essay and pledge

Include with your assignment the following information.

For programming assignments, please put it a separate file. (I strongly prefer plain text, but if you insist you can put it in a PDF — just no word-processor documents or Google Drive links please.) For written assignments, please put it in your main document.

7.1 Pledge

This should include the Honor Code pledge, or just the word "pledged", *plus* at least one of the following about collaboration and help (as many as apply). Text *in italics* is explanatory or something for you to fill in; you don't need to repeat it!

- I did not get outside help *aside from course materials, including starter code, readings, sample programs, the instructor.*
- I worked with *names of other students* on this assignment.

- I got help with this assignment from *source of help — ACM tutoring, another student in the course, etc. (Here, "help" means significant help, beyond a little assistance with tools or compiler errors.)*
- I got help from *outside source — a book other than the textbook (give title and author), a Web site (give its URL), etc.. (Here too, you only need to mention significant help — you don't need to tell me that you looked up an error message on the Web, but if you found an algorithm or a code sketch, tell me about that.)*
- I provided help to *names of students* on this assignment. *(And here too, you only need to tell me about significant help.)*

7.2 Essay

This should be a brief essay (a sentence or two is fine, though you can write as much as you like) telling me what if anything you think you learned from the assignment, and what if anything you found interesting, difficult, or otherwise noteworthy.