Administrivia

Slide 1

- Reminder: Don't view this one until after turning in Homework 2.

Homework 2 Revisited — Sequential Programs

Slide 2

- First step is probably to run sequential C program a few times. (Using what machines? what parameters?)
- Do results vary depending on seed? (Yes.)
- Are results better for more samples? (Sometimes(!).)
- Next it might be interesting to rewrite in Java ...
- Are results the same for C and Java programs? (No.)
- Does execution time make sense — fairly consistent from run to run, scales with number of samples? (Yes.)

Slide 3

Homework 2 Revisited — Parallel Programs

- My idea was that you would do something very similar to what we did with numerical integration:
  - Consider each “throw a dart” operation as a task.
  - Divide tasks among UEs, with each of them computing a local count.
  - Combine local counts at the end, and then compute π.
- Recall that for numerical integration we got different results for different numbers of UEs because floating-point addition is not associative. Will that happen here? (It shouldn’t!)

Slide 4

Homework 2 Revisited — Parallel Programs, Continued

- Naive strategy doesn’t turn out well here: Programs may produce worse results with more samples, and OpenMP programs may slow down with more threads.
- What’s going on? basically, a naive strategy assumes that calls to `rand()` are independent, just as the computations in the loop in the example are. Is that true?
- Maybe we should step back and talk about “generating random numbers” . . .

Slide 5

A Little About Random Numbers

- (Canonical reference — discussion in volume 2 of Knuth's *The Art of Computer Programming*. Very mathematical. Other references may be easier.)
- Many application areas that depend on "random" numbers (whatever we mean by that) — simulation (of physical phenomena), sampling, numerical analysis (Monte Carlo methods, e.g.), etc.
- Early on, people used physical methods (currently still in use in lotteries), and thought about building hardware to generate "random" results. No good large-scale solution, plus it seemed useful to be able to repeat a calculation.
- Hence need for "random number generator" (RNG) — way to generate "random" sequences of elements from a given set (e.g., integers or doubles). Tricky topic. Many early researchers got it wrong. Many application writers aren't interested in details.

Slide 6

Desirable Properties of RNG — "Randomness"

- Obviously a key goal, if tricky to define. A thought-experiment definition: Suppose we're generating integers in the range from 1 through d, and we let an observer examine as much of the sequence as desired, and ask for a guess for any other element in the sequence. If the probability of the guess being right is more than $1/d$, the sequence isn't random.
- Also want uniformity — for each element, equal probability of getting any of the possible values.
- For some applications, also need to consider "uniformity in higher dimensions": If you consider treating the sequence as sequence of points in 2D, 3D, etc., space., are the points spread out evenly?

Slide 7

Other Desirable Properties of RNG

- Reproducibility. For some applications, not important, or even bad. But for many others, good to be able to repeat an experiment. Usually meet this need with "pseudo random number generator" — algorithm that computes sequence using initial value (seed) and definition of each element in terms of previous element(s).
- Speed. Probably not a major goal, though, since most applications involve lots of other calculations.
- Large cycle length. If every element depends only on the one before, once you get the initial element again what happens? and usually that's not good.

Slide 8

Some Popular RNG Algorithms

- Linear Congruential Generator (LCG).

  $$x_n = (ax_{n-1} + c) \bmod m$$

  m constrains cycle length (period) — usually prime or a power of 2. a and c must be carefully chosen. Results good overall, but least significant bits "aren't very random", which affects how well they work for generating points in 2D, etc., space.
- Lagged-Fibonacci Generator.

  $$x_n = (x_{n-j} \ op \ x_{n-k}) \bmod 2^m, \quad j < k$$

  where op is a binary operator ($+$, $\times$, etc.). j and k must be carefully chosen. Must also choose "enough" initial elements (how many depends on j and k).

Slide 9

Some RNG Library Functions

- C library function `rand()` and friends: Variant of LFG.
  (Where are previous values stored?)
- Java library class `Random`: LCG.
  (Where is previous value stored?)

Slide 10

Homework 2 Results — Recap

- Quality of results can vary depending on seed, but not in any obvious way. Effect seems to decrease as number of samples increases, however.
- OpenMP program can produce different results for different numbers of threads(!).
- OpenMP programs can have very poor performance — times *increase* for more threads.
- MPI program can produce different results for different numbers of threads, but performance is usually good.

Slide 11

RNGs and Homework 2

- Does this explain why accuracy of result might depend on choice of seed? (Yes.)
- Does it explain why results can vary depending on number of UEs? (Is the explanation the same for the different programming environments?)
- Does it explain why performance of OpenMP program can be disappointing?

Slide 12

Parallelizing RNGs

- RNGs are used in some applications that are compute-intensive and thus appealing candidates for parallelization.
- How to do this?

Slide 13

Approaches to Parallelizing RNGs — Central Server

- Use one UE to generate sequence, have it distribute results to other UEs or let them request them.
- Reproducible? Efficient? Other problems?

Slide 14

Approaches to Parallelizing RNGs — Central Server, Continued

- Same sequence, but maybe not distributed same way.
- Could be inefficient / bottleneck.

Slide 15

Approaches to Parallelizing RNGs — Cycle Division

- Cycle division — split elements of original sequence between UEs, having each UE generate "its" elements. Two basic schemes — "leapfrog" and "cycle splitting".
- Reproducible? Efficient? Other problems?

Slide 16

Approaches to Parallelizing RNGs — Cycle Division, Continued

- Same sequence, split the same way.
- Could be other problems – subsequences might not be "random".
- Also could be very inefficient, depending on how each UE computes its elements (e.g., for leapfrogging, simplest approach is just to generate all elements and skip some).

Approaches to Parallelizing RNGs — Parameterization

Slide 17

- Parameterization — e.g., "cycle parameterization" exploits property that some RNGs can generate different cycles depending on seed. Idea is to "parameterize" algorithm so UEs generate different cycles.
- Reproducible? Efficient? Other problems?

Approaches to Parallelizing RNGs — Parameterization, Continued

Slide 18

- Depends on being able to parameterize in a way that cycles don't overlap.
- Related to choice of seed in the first place. Figuring how to do this effectively could be difficult.

Slide 19

Parallel RNG With Distributed Memory

- Thread safety not an issue. But also have no access to shared state, so each process should probably generate sequence independently. (Central server would work, but again, could be a bottleneck.)
- "Leapfrog" approach seems attractive.

  Naive implementation would just have each process generate whole sequence and ignore elements it doesn't want. Good idea? (Sometimes, but probably not for the Homework 2 problem.)

  Knuth includes algorithm for generating just selected elements of LCG, based on modifying a and c.
- Starting different processes with different seeds also seems promising. Is there a situation in which that wouldn't work? (Can you guarantee that sequences don't overlap "too much"?)

Slide 20

Parallel RNG With Shared Memory

- Thread safety an issue, but have access to shared state, which might be attractive.
- Adaptation of "central server" idea — use regular library function, but ensure one-at-a-time access. Good idea? (Maybe for some applications, but probably won't work well for Homework 2 problem.)
- Other approaches similar to distributed-memory case, but require that each thread have its own "internal state". Good idea? doable? (Could be a problem if using library functions.)

Slide 21

RNG Functions Revisited

- C library function `rand()` and friends: Variant of LFG. Can specify seed, but internal state apparently hidden. `rand_r()` allows keeping internal state in user-provided buffer.
- Java library class `Random`: LCG. Can specify seed. Not known whether different instances share internal state, but seems unlikely.
- Or one can write one's own . . . (And that's what Homework 3 will ask you to do. But in real-world situations, it's probably better to investigate good third-party libraries, commercial if need be.)

Slide 22

Improving on Homework 2 Solutions

- How do we improve performance?

  (Should be straightforward — any revised algorithm that doesn't use a shared state should help.)
- How do we improve accuracy?

  (Should be straightforward — any revised algorithm that doesn't generate the same sequence for every UE should help at least a little.)
- And how will we know a revised solution is better?

  (Measure carefully / systematically.)

Slide 23

Homework 3 — Implementing LCG

- Implementing a 48-bit LCG function is doable in both C (with `int64_t` and Java `Long`). Note, however, that the multiplication required to generate the next element can overflow — which is no problem since we only want the value mod 2^{48}, *but* consider what happens if the overflow produces a negative result. Hence my suggestion to compute this with bitwise AND (`&`) rather than with `%`.
- Implementing the described leapfrog scheme is trickier, and beyond the scope of this course, but I got interested a while back . . .

Slide 24

Homework 3 — Implementing LCG With Leapfrogging

- Algorithm seems straightforward (compute and use modified constants a' and b', but when I tried implementing in Scala, I found I needed `BigInt` to compute them correctly (they're 48 bits and can be `Long`s, but some intermediate results apparently need to be larger?).
- Same implementation works in Java, using its analogous classes (e.g., `BigInteger`).
- And in C . . . Same approach works, using GMP library for arbitrary-precision arithmetic.
- You don't have to do this, though if you're curious and want some extra points, go for it. Assignment includes a little starter code for the C version.

Slide 25

Minute Essay

- Does this discussion help you understand better anything about your Homework 2 results?