Administrivia

- (None?)

Slide 1

Homework 2 Recap/Review/Revisited

- How did you approach this problem? i.e., what was your overall strategy?
- Did you get good results (close to π)? (If not, why do you think you didn't?)
- Did you get good speedups? (If not, why do you think you didn't?)

Slide 2

Slide 3

Homework 2 Review, Continued

- My idea was that you would do something very similar to what we did with numerical integration, with one task per simulated dart.

  Unfortunately, there's a small problem: The tasks aren't really independent, because of how library functions to generate "random numbers" really work.
- Depending on what you did about this problem (ignore it, try to solve it, etc.), you probably got either not-so-good results or disappointing performance — or both.

Slide 4

A Little About Random Numbers

- (Sources: Knuth, Quinn, SPRNG Web site.)
- 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.), programming (to generate data, also some algorithms), 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, though, and besides 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 5

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": Consider treating sequence as sequence of points in 2D, 3D, etc., space. Are the points spread out evenly?

Slide 6

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 7

Parallelizing RNGs

- RNGs are used in some applications that are compute-intensive and thus appealing candidates for parallelization. How to do this?
- Naive approach — identical calculations in each UE (thread/process), use same RNG with same seed. Assuming no sharing of internal state, what happens?

  And what happens if there *is* sharing of internal state? Is the function “thread-safe”? i.e., if two threads call it at the same time, what happens? obviously potentially a problem if “state” is saved in global variable hidden from users.

Slide 8

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?
- 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?
- 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?

Slide 9

Some Popular RNG Algorithms

- Linear Congruential Generator (LCG).

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

  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 $+$ (additive LFG) or $\times$ (multiplicative LFG). Again, k must be carefully chosen. Must also choose "enough" initial elements.

Slide 10

RNG Functions

- C library function `random` and friends: Variant of LFG. Can specify seed, but internal state apparently hidden.
- C library function `drand48` and friends: LCG. Can specify seed. One variant allows keeping internal state in user-provided buffer.
- Java library class `RandomGenerator`: LCG. Can specify seed. Not known whether different instances share internal state, but seems unlikely.
- Or one can write one's own . . .

Slide 11

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.
- "Leapfrog" approach seems attractive.

  Naive implementation would just have each process generate whole sequence and ignore elements it doesn't want. Good idea?

  Knuth includes algorithm for generating just selected elements of LCG, based on modifying a and c.
- Starting different processes with different seeds seems good. But how do you guarantee that sequences don't overlap too much?

Slide 12

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 — seems attractive. For us, might be effective, especially if we generate elements two at a time. Efficient?
- Other approaches similar to distributed-memory case, but require that each thread have its own "internal state". Could be a problem if using library functions.

Slide 13

Possible Homework 2 Revisions

- Improve results (all versions): Experiment with different RNGs, and/or determine what seed(s) give good results.
- Improve performance (OpenMP and Java): Figure out how to get thread safety without synchronization overhead.
- Improve packaging (Java): Figure out how to avoid ugly global variables.
- Next homework — revise Homework 2, improving as much as you can in all of these areas. I will write up requirements . . .

Slide 14

Minute Essay

- Based on what we talked about today, what would you change first about each of your Homework 2 programs? Be as specific as you can.