Slide 1

Administrivia

- Homework 2 to be on Web Monday.

Slide 2

Minute Essay From Last Lecture

- Question was: What kind of experiments might be useful in figuring out whether a random sequence is "good" for the Monte Carlo pi problem?
- Answers:

  Check whether it gives a good approximation of pi.

  Try a lot of seeds (script this) and see which ones come closest.

  Graph (?) numbers and check distribution.

  Check whether pairs cover dartboard / fall proportionally inside and outside the circle. (Similar to first answer?)

  Compare how random numbers are over long execution.

  Try different functions for generating random numbers. ("Guess and check.")
- (Or study up really fast on what "the literature" recommends.)

Slide 3

Homework 1 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 and Java 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 4

A Little About Random Numbers, Recap

- Canonical reference is in volume 2 of Knuth's *The Art of Computer Programming*? look at man page for `rand`, documentation of `java.util.Random`.

  Another good reference — *Numerical Recipes*.
- Many applications need a "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, Recap

- "Randomness" (though defining that precisely may not be easy).
- Uniformity — for each element, equal probability of getting any of the possible values. (Some applications do need other distributions, but can usually generate them from uniformly-distributed sequence.)

  For some applications, also need to consider "uniformity in higher dimensions": If sequence is treated as a 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

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 8

Some RNG Library Functions

- C library function `random` and friends: Variant of LFG.

  (Where are previous values stored?)

- Java library class `Random`: LCG.

  (Where is previous value stored?)

Slide 9

RNGs and Homework 1

- Does this explain why accuracy of result might depend on choice of seed?
- Does it explain why results for C and Java programs are different?
- Does it explain why results can vary depending on number of threads? (Is the explanation the same for the different programming environments?)
- Does it explain why performance of OpenMP and Java programs can be disappointing?

Slide 10

Parallelizing RNGs

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

Slide 11

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? (Same sequence, but maybe not distributed same way. Could be inefficient / bottleneck.)

- 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? (Same sequence, split the same way, but could be other problems – subsequences might not be "random". Also could be very inefficient.)

- 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.

Slide 12

Reproducible? Efficient? Other problems? (Depends on being able to parameterize in a way that cycles don't overlap. Related to choice of seed in the first place.)

Slide 13

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? (Sometimes, but probably not for the Homework 1 problem.)

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

Slide 14

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 1 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 15

RNG Functions Revisited

- 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 16

Improving on Homework 1 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.)
- Is there a "think outside the box" solution that might not require a careful parallel RNG?

  (Maybe — idea of "geometric decomposition".)
- And how will we know a revised solution is better?

Slide 17

Minute Essay

- (This might be a more-than-a-minute essay, so feel free to think about it over the weekend and send me an answer by e-mail. Extra class-participation points for particularly good answers.)
- Sketch out a plan for comparing different solutions to the Homework 1 problem — both accuracy and performance.

  What tools might help with this? (I think simple shell scripts and redirecting output. If this is foreign to you, tell me and we can review briefly.)