Administrivia

- (Homework 2 coming soon.)

Numerical Integration Results, Yet Again

- Performance of parallel Java version is — mysterious? and seems to vary with details of platform in a way that the other versions don't.
- Why? based on my experiments, it has to do with just-in-time compilation . . .

Homework 1 Revisited — Sequential Programs

- First step is probably to run sequential programs a few times. (Using what machines? what parameters?)
- Do results vary depending on seed? (Yes.)
- Are results better for more samples? (Sometimes.)
- 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? from machine to machine? (Yes.)

Homework 1 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 pi.
- 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?

Homework 1 Revisited — Parallel Programs, Continued

- Probably should repeat sequential-program experiments, right? with same inputs, but varying numbers of UEs. (How many UEs should we use?)
- And if we do that, results can be — “interesting”?
  - Different answers depending on number of UEs. (How can that be? Is the answer the same for OpenMP, Java, and MPI?)
  - Disappointing performance (but maybe not for all three versions?)
- What’s going on? well, maybe we should step back and talk about “generating random numbers” . . .

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.

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?

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.

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

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?)

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 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.
- Java programs can exhibit either behavior, depending on how you approach the problem.

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?

Parallelizing RNGs

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

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?

Approaches to Parallelizing RNGs — Central Server, Continued

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

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?

Approaches to Parallelizing RNGs — Cycle Division, Continued

- Same sequence, split the same way, but could be other problems – subsequences might not be “random”. Also could be very inefficient.

Approaches to Parallelizing RNGs — Parameterization

- 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

- Depends on being able to parameterize in a way that cycles don't overlap. Related to choice of seed in the first place.

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"?)

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.)

RNG Functions Revisited

- C library function `rand` 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 `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 2 will ask you to do. But in real-world situations, it’s probably better to investigate good third-party libraries, commercial or not.)

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?

  (Measure carefully / systematically.)

Not-So-Simple Point-to-Point Communication in MPI

- For not-too-long messages and when readability is more important than performance, `MPI_Send` and `MPI_Recv` are probably fine.
- If messages are long, however, buffering can be a problem, and can even lead to deadlock. Also, sometimes it's nice to be able to overlap computation and communication.
- Therefore, MPI offers several other kinds of send/receive functions, including:
  - Synchronous (`MPI_Ssend`, `MPI_Recv`) — blocks both sender and receiver until communication can occur.
  - Non-blocking send/receive (`MPI_Isend`, `MPI_Irecv`, `MPI_Wait`) — doesn't block, program must explicitly test/wait.
  - Which is faster/better? probably best to try them and find out. (Sample programs `exchange*`.)

Minute Essay

- None — sign in.