Slide 1

Administrivia

- (None?)

Slide 2

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(!), though not all students' did.
- 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 3

Parallelizing RNGs

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

Slide 4

Approaches to Parallelizing RNGs — Central Server

- Use one UE to generate sequence, have it distribute results to other UEs or let them request them.
- Can produce results identical to sequential computation (depending on how central server distributes results), but potentially a bottleneck.

Slide 5

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".
- Seems like this should be good with regard to getting results that don't vary depending on number of UEs. But can it "parallelize" well?
- 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).

Slide 6

Approaches to Parallelizing RNGs — Parameterization

- Parameterization — e.g., "cycle parameterization" exploits property that some (most? all?) RNGs can generate different cycles depending on seed. Idea is to "parameterize" algorithm so UEs generate different cycles.
- 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 7

Sidebar: “Thread Safety”

- Some functions said to be “thread-safe”.
- What does that even mean? Can be called from more than one thread at the same time without possibility of race conditions.

Slide 8

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. Could work for some computations, where generating the “random” numbers is a small part of the whole computation, but that isn’t the case here.
- Starting different processes with different seeds also seems promising. Can’t guarantee that sequences don’t overlap “too much”, but for this assignment it could be good enough. (Picking good seeds for parallel RNGs is apparently a complex topic. I tried to read up on it and got nowhere!)

Slide 9

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. Again, if this wasn't such a big part of the computation, this would work, but not for this application.
- 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, but if we write our own, we have total control — a little along the lines of computing a global sum by computing local sums and combining.

Slide 10

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 . . . Which is what Homework 2 asks you to do. But in real-world situations, it's probably better to investigate good third-party libraries, commercial if need be.

Slide 11

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?

  (Only an issue if you inadvertently ended up with code where all UEs generate the same sequence — e.g., in a naive MPI version.)

  (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 12

Homework 2 — 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:

  Knuth includes algorithm for generating just selected elements of LCG, based on modifying a and c.

  I got interested a while back . . .

Slide 13

Homework 2 — 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 14

Minute Essay

- Questions?