Slide 1

Administrivia

- Homework 2 to be on the Web soon. I will send mail.

Slide 2

Minute Essay From Last Lecture

- Some interesting comments (about MPI), though nothing really stood out.
- One person did mention that `mpicc` just seems to call `gcc` with added flags. True!
- It seems that a potential source of confusion is not realizing the `mpirun -np N ./mypgm` starts `N` copies of `mypgm`, each with a different “rank” (process ID) but otherwise identical.

Slide 3

Multithreaded Programming in Java, Recap/Revisited

- Basic functionality — threads, synchronization, wait/notify — has been in the language from the start.
- At some point in Java's evolution, `java.util.concurrent` was added to the library. *Many* useful features, many of which we won't use because they make things "too easy".
- However, "thread pools" and "futures" can be pretty useful ...

Slide 4

Java "Thread Pools" and "Futures"

- One limitation of the original design of threads was that the `run` method couldn't return anything. Also, explicitly creating and starting threads can be annoying and doesn't provide an easy way to set up "worker" threads to which work can be assigned.
- "Thread pools" (class `ExecutorService` provides various ways to set up a pool of threads and assign them work.
- `Future` class and `Callable` provide a way for tasks to return values. (Notice that Java `Future`s seem to be fairly different from the ones in Scala, which are much more asynchronous.)
- (More versions of "hello world" and numerical integration.)

Slide 5

Controlling Threads in Java

- Preferred method of controlling one thread from another uses “interrupted” status. (Early version of Java provided other methods, e.g., `stop` — now deprecated.)
- Set status with `interrupt` (instance method).
- Check status with `isInterrupted` (instance method) or `interrupted` (static method), or by catching `InterruptedException` thrown by `wait`, `sleep`, `join`, etc.

  (Right — all of these methods potentially throw `InterruptedException`, which is a “checked exception”, and in Java you’re supposed to do something about those.)
- (Example another time.)

Slide 6

A Few Words about Measuring Performance in Java

- C programs should exhibit fairly predictable performance — i.e., related in some semi-obvious way to something in the code.
- Java programs, however . . . By default byte code is interpreted, but most JVMs will do “just in time” compilation to native code — but it’s difficult to predict under what circumstances. Also, programmers have basically no control over when garbage collection runs, which also can affect performance. So timing experiments may be less useful than in other languages.

Slide 7

Homework 1 Revisited — Sequential Programs

- 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? from machine to machine? (Yes.)

Slide 8

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 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 9

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, MPI, and Java?)
  - 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” . . .

Slide 10

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 11

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 12

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 13

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.

Slide 14

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

Slide 15

- 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.

RNGs and Homework 1

Slide 16

- 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 threads? (Is the explanation the same for the different programming environments?)
- Does it explain why performance of OpenMP program 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?

Slide 17

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 18

Approaches to Parallelizing RNGs — Central Server, Continued

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

Slide 19

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 20

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.

Slide 21

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?

Slide 22

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.

Slide 23

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 also seems promising. Is there a situation in which that wouldn't work? (Can you guarantee that sequences don't overlap "too much"?)

Slide 24

Slide 25

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 26

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.)

Slide 27

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.)

Slide 28

Minute Essay

- What kind of experiments might be useful in figuring out whether a random sequence is "good" for the Monte Carlo pi problem? (This question is really somewhat beyond the scope of this course, but interesting to think about!)