Monte Carlo Methods &
Parallel Random Number Generation
Outline

♦ Monte Carlo method
♦ Sequential random number generators
♦ Parallel random number generators
♦ Generating non-uniform random numbers
♦ Monte Carlo case studies
Monte Carlo Method

- Solve a problem using statistical sampling
- Name comes from Monaco’s gambling resort city
- First important use in development of atomic bomb during World War II
Applications of Monte Carlo Method

- Evaluating integrals of arbitrary functions of 6+ dimensions
- Predicting future values of stocks
- Solving partial differential equations
- Sharpening satellite images
- Modeling cell populations
- Finding approximate solutions to NP-hard problems
Example of Monte Carlo Method

\[ \frac{\text{Circle}}{\text{Square}} = \frac{\pi D^2 / 4}{D^2} = \frac{\pi}{4} \]
Example of Monte Carlo Method

\[
\frac{16}{20} \approx \frac{\pi}{4} \implies \pi \approx 3.2
\]
Absolute Error

- Absolute error is a way to measure the quality of an estimate.
- The smaller the error, the better the estimate.
- $a$: actual value
- $e$: estimated value
- Absolute error $= \frac{|e-a|}{a}$
Why Monte Carlo is Effective

- Error in Monte Carlo estimate decreases by the factor $1/n^{1/2}$
- Rate of convergence independent of integrand’s dimension
- Deterministic numerical integration methods do not share this property
- Hence Monte Carlo superior when integrand has 6 or more dimensions
Parallelism in Monte Carlo Methods

- Monte Carlo methods often amenable to parallelism
- Find an estimate about $p$ times faster
- OR
- Reduce error of estimate by $p^{1/2}$
Random versus Pseudo-random

♦ Virtually all computers have “random number” generators
♦ Their operation is deterministic
♦ Sequences are predictable
♦ More accurately called “pseudo-random number” generators
♦ In this chapter “random” is shorthand for “pseudo-random”
♦ “RNG” means “random number generator”
Why Prefer Pseudo-Random Numbers

♦ True random numbers are rarely used in computing because:
  – difficult to generate reliably
  – the lack of reproducibility would make the validation of programs that use them extremely difficult

♦ Computers use pseudo-random numbers:
  – finite sequences generated by a deterministic process
  – but indistinguishable, by some set of statistical tests, from a random sequence.
Properties of an Ideal RNG

♦ Uniformly distributed
♦ Uncorrelated
♦ Never cycles
♦ Satisfies any statistical test for randomness
♦ Reproducible
♦ Machine-independent
♦ Changing “seed” value changes sequence
♦ Easily split into independent subsequences
♦ Fast
♦ Limited memory requirements
No RNG Is Ideal

- Finite precision arithmetic $\Rightarrow$ finite number of states $\Rightarrow$ cycles
  - Period = length of cycle
  - If period $>$ number of values needed, effectively acyclic
- Reproducible $\Rightarrow$ correlations
- Often speed versus quality trade-offs
Linear Congruential RNGs

\[ X_i = (a \times X_{i-1} + c) \mod M \]

Multiplier
Additive constant
Modulus
Sequence depends on choice of seed, \( X_0 \)
Linear Congruential RNGs

- As numbers are taken from a finite set (for example, integers between 1 and $2^{31}$), any generator will eventually repeat itself.
- The length of the repeated cycle is called the **period** of the generator.
- A good generator is one with a long period and no discernible correlation between elements of the sequence.
- **Example:** $X_{k+1} = (3 \times X_k + 4) \mod 8$
  
  $E = \{1, 7, 1, 7, \ldots\}$ is bad.
Period of Linear Congruential Sequence

♦ **Theorem:** The linear congruential sequence has period \( m \) if and only if
  
  – \( c \) is relatively prime to \( m \);
  
  – \( b = a - 1 \) is a multiple of \( p \), for every prime \( p \) dividing \( m \);
  
  – \( b \) is a multiple of 4, if \( m \) is a multiple of 4.

♦ **Example:** \( X_{k+1} = (7 \times X_k + 5) \mod 18 \)

\( E = \{1, 12, 17, 16, 9, 14, 13, 6, 11, 10, 3, 8, 7, 0, 5, 4, 15, 2, 1, \ldots \} \)
Period of Linear Congruential Sequence

- Maximum period is $M$
- For 32-bit integers maximum period is $2^{32}$, or about 4 billion
- This is too small for modern computers
- Use a generator with at least 48 bits of precision
Producing Floating-Point Numbers

- $X_i$, $a$, $c$, and $M$ are all integers
- $X_i$'s range in value from 0 to $M-1$
- To produce floating-point numbers in range $[0, 1)$, divide $X_i$ by $M$
Defects of Linear Congruential RNGs

♦ Least significant bits correlated
  – Especially when $M$ is a power of 2
♦ $k$-tuples of random numbers form a lattice
  – Especially pronounced when $k$ is large
Lagged Fibonacci RNGs

\[ X_i = X_{i-p} \ast X_{i-q} \]

- \( p \) and \( q \) are lags, \( p > q \)
- \( \ast \) is any binary arithmetic operation
  - Addition modulo \( M \)
  - Subtraction modulo \( M \)
  - Multiplication modulo \( M \)
  - Bitwise exclusive or
Properties of Lagged Fibonacci RNGs

- Require $p$ seed values
- Careful selection of seed values, $p$, and $q$ can result in very long periods and good randomness
- For example, suppose $M$ has $b$ bits
- Maximum period for additive lagged Fibonacci RNG is $(2^p - 1)2^{b-1}$
Ideal Parallel RNGs

- All properties of sequential RNGs
- No correlations among numbers in different sequences
- Scalability
- Locality
Parallel RNG Designs

- Manager-worker
- Leapfrog
- Sequence splitting
- Independent sequences
Parallel RNG Designs

- Centralized
- Replicated
- Distributed
- Independent Sequences
Centralized Approach

- A sequential generator is encapsulated in a task from which other tasks request random numbers.

- **Advantages:**
  - Avoids the problem of generating multiple independent random sequences

- **Disadvantages:**
  - Unlikely to provide good performance
  - Makes reproducibility hard to achieve: the response to a request depends on when it arrives at the generator, and the result computed by a program can vary from one run to the next.
Manager-Worker Parallel RNG

- Manager process generates random numbers
- Worker processes consume them
- If algorithm is synchronous, may achieve goal of consistency
- Not scalable
- Does not exhibit locality
Replicated Approach

- Multiple instances of the same generator are created (for example, one per task). Each generator uses either the same seed or a unique seed, derived, for example, from a task identifier.

- **Advantages:**
  - Efficiency
  - Ease of implementation

- **Disadvantages:**
  - Not guaranteed to be independent and, can suffer from serious correlation problems.
Distributed Approach

- Responsibility for generating a single sequence is partitioned among many generators, which can then be parcelled out to different tasks.

- **Advantages:**
  - The analysis of the statistical properties of the distributed generator is simplified because the generators are all derived from a single generator
  - Efficiency

- **Disadvantages:**
  - Difficult to implement on parallel computers
Distributed Approaches

- The techniques described here are based on an adaptation of the linear congruential algorithm called the random tree method.
- This facility is particularly valuable in computations that create and destroy tasks dynamically during program execution.

- The Random Tree Method
- The Leapfrog Method
- Modified Leapfrog
The Random Tree Method

- The random tree method employs two linear congruential generators, \( L \) and \( R \), that differ only in the values used for \( a \).
- \( L_{k+1} = a_L L_k \mod m \)
- \( R_{k+1} = a_R R_k \mod m \)
Application of the left generator L to a seed generates one random sequence; application of the right generator R to the same seed generates a different sequence.

By applying the right generator to elements of the left generator's sequence (or vice versa), a tree of random numbers can be generated.

By convention, the right generator R is used to generate random values for use in computation, while the left generator L is applied to values computed by R to obtain the starting points R0, R1, etc., for new right sequences.
Random Tree Method

♦ **Advantages:**
  – Useful to generate new random sequences in a reproducible and noncentralized fashion. This is valuable, in applications in which new tasks and hence new random generators must be created dynamically.

♦ **Disadvantages:**
  – There is no guarantee that different right sequences will not overlap. If two starting points happen to be close to each other, the two right sequences that are generated will be highly correlated.
The Leapfrog Method

♦ A variant of the random tree method
♦ Used to generate sequences that can be guaranteed not to overlap for a certain period.
♦ Useful where a program requires a fixed number of generators. (For example, one generator for each task).
Let $n$ be the number of sequences required. Then we define $a_L$ and $a_R$ as $a$ and $a^n$, respectively,

$L_{k+1} = a_L \cdot L_k \mod m$

$R_{k+1} = a^n \cdot R_k \mod m$

Create $n$ different right generators $R_0 .. R_{n-1}$ by taking the first $n$ elements of $L$ as their starting values.

The name “leapfrog method” refers that the $i$th sequence $R_i$ consists of $L_i$ and every $n$th subsequent element of the sequence generated by $L$.

As the method partitions the elements of $L$, each subsequence has a period of at least $P/n$, where $P$ is the period of $L$.

The $n$ subsequences are disjoint for their first $P/n$ elements.
The generator for the $r$ th subsequence, $R_r$, is defined by $a^n$ and $R_r_0 = L_r$.

First, compute $a^r$ and $a^n$

Then, compute members of the sequence $R_r$ as follows, to obtain $n$ generators, each defined by a triple $(R_r_0, a^n, m)$, for

$0 \leq r < n$.

- $R_r_0 = (a^r L_0) \mod m$
- $R_{r_i+1} = (a^n R_{r_i}) \mod m$
Leapfrog Method

Process with rank 1 of 4 processes
Properties of Leapfrog Method

- Easy to modify linear congruential RNG to support jumping by $p$
- Can allow parallel program to generate same tuples as sequential program
- Does not support dynamic creation of new random number streams
**Modified Leapfrog**

- A variant of the leapfrog method
- Used in situations, where we know the maximum number, \( n \), of random values needed in a subsequence but not the number of subsequences required.
- The role of \( L \) and \( R \) are reversed so that the elements of subsequence \( i \) are the contiguous elements
- \( L_{k+1} = a^n L_k \mod m \)
- \( R_{k+1} = a R_k \mod m \)

- It is not a good idea to choose \( n \) as a power of two, as this can lead to serious long-term correlations.
Figure: Modified leapfrog with n=3. Each subsequence contains three contiguous numbers from the main sequence.
Modified Leapfrog
(Sequence Splitting)

Process with rank 1 of 4 processes
Properties of Sequence Splitting

- Forces each process to move ahead to its starting point
- Does not support goal of reproducibility
- May run into long-range correlation problems
- Can be modified to support dynamic creation of new sequences
Some Tests For Random Number Generators

- The basic idea behind the statistical tests is that the random number streams obtained from a generator should have the properties of a random sample drawn from the uniform distribution.
- Tests are designed so that the expected value of some test statistic is known for uniform distribution.
- The empirically generated random number stream is then subject to the same test, and the statistic obtained is compared against the expected value.
Frequency Test

♦ The focus of the test is the proportion of zeroes and ones for the entire sequence.

♦ Example:
  - (input) 
    E=11001001000011111101101010100010001000101
    1010001100001000110100110001001100011001100010100010111000
  - (input) n = 100, # of 1’s : 42
  - (output) P-value = 0.109599 (coin flip 0-42 1’s)
  - (conclusion) Since P-value >= 0.01, accept the sequence as random.
Runs Test

- A run is an uninterrupted sequence of identical bits.
- The focus of this test is the total number of runs in the sequence.
- A run of length $k$ consists of exactly $k$ identical bits and is bounded before and after with a bit of the opposite value.
- The purpose of the runs test is to determine whether the number of runs of ones and zeros of various lengths is as expected for a random sequence.
- Determines whether the oscillation between such zeros and ones is too fast or too slow.
A fast oscillation occurs when there are a lot of changes, e.g., 010101010 oscillates with every bit.

(input) $E = 1100100100001111101101010100010001000100010$
$11010001100001000110100110001001100011001$
$100010100010111000$

(input) $n = 100$

(output) $P$-value $= 0.500798$

(conclusion) Since $P$-value $\geq 0.01$, accept the sequence as random.
Testing Parallel Random Number Generators

♦ A good parallel random number generator must be a good sequential generator.

♦ Sequential tests check for correlations within a stream, while parallel tests check for correlations between different streams.
Testing Parallel Random Number Generators

- Exponential sums
- Parallel spectral test
- Interleaved tests
- Fourier transform test
- Blocking test
Fourier Transform Test

- Fill a two dimensional array with random numbers. Each row of the array is filled with random numbers from a different stream.
- Calculate the Fourier coefficients and compare with the expected values.
- This test is repeated several times and check if there are particular coefficients that are repeatedly “bad”.
Blocking Test

- Use the fact that the sum of independent variables asymptotically approaches the normal distribution to test for the independence of random number streams.

- Add random numbers from several stream and form a sum.

- Generate several such sums and check if their distribution is normal.
SPRNG

- A “Scalable Library for Pseudorandom Number Generation”,
- Designed to use parametrized pseudorandom number generators to provide random number streams to parallel processes.
- Includes
  - Several, qualitatively distinct, well tested, scalable RNGs
  - Initialization without interprocessor communication
  - Reproducibility by using the parameters to index the streams
- Reproducibility controlled by a single “global” seed
- Minimization of interprocessor correlation with the included generators
- A uniform C, C++, Fortran and MPI interface
- Extensibility
- An integrated test suite.
Independent Sequences

♦ Run sequential RNG on each process
♦ Start each with different seed(s) or other parameters
♦ Example: linear congruential RNGs with different additive constants
♦ Works well with lagged Fibonacci RNGs
♦ Supports goals of locality and scalability
Other Distributions

♦ Analytical transformations
♦ Box-Muller Transformation
♦ Rejection method
Example 1:

- Produce four samples from an exponential distribution with mean 3
- Uniform sample: 0.540, 0.619, 0.452, 0.095
- Take natural log of each value and multiply by -3
- Exponential sample: 1.850, 1.440, 2.317, 7.072
Example 2:

- Simulation advances in time steps of 1 second
- Probability of an event happening is from an exponential distribution with mean 5 seconds
  - \( f(x) = \frac{1}{5}\exp(-x/5) \), \( F(x) = 1 - \exp(-x/5) \)
- What is probability that event will happen in next second?
  - \( F(1) = 1 - \exp(-1/5) = 0.18127 \)
- Use uniform random number to test for occurrence of event
Box-Muller Transformation

- Cannot invert cumulative distribution function to produce formula yielding random numbers from normal (gaussian) distribution
- Box-Muller transformation produces a pair of standard deviates $g_1$ and $g_2$ from a pair of normal deviates $u_1$ and $u_2$
Box-Muller Transformation

repeat
    \( v_1 \leftarrow 2u_1 - 1 \)
    \( v_2 \leftarrow 2u_2 - 1 \)
    \( r \leftarrow v_1^2 + v_2^2 \)
until \( r > 0 \) and \( r < 1 \)
\( f \leftarrow \sqrt{-2 \ln r / r} \)
\( g_1 \leftarrow f v_1 \)
\( g_2 \leftarrow f v_2 \)
Example

- Produce four samples from a normal distribution with mean 0 and standard deviation 1

<table>
<thead>
<tr>
<th>$u_1$</th>
<th>$u_2$</th>
<th>$v_1$</th>
<th>$v_2$</th>
<th>$r$</th>
<th>$f$</th>
<th>$g_1$</th>
<th>$g_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.234</td>
<td>0.784</td>
<td>-0.532</td>
<td>0.568</td>
<td>0.605</td>
<td>1.290</td>
<td>-0.686</td>
<td>0.732</td>
</tr>
<tr>
<td>0.824</td>
<td>0.039</td>
<td>0.648</td>
<td>-0.921</td>
<td>1.269</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.430</td>
<td>0.176</td>
<td>-0.140</td>
<td>-0.648</td>
<td>0.439</td>
<td>1.935</td>
<td>-0.271</td>
<td>-1.254</td>
</tr>
</tbody>
</table>
Different Mean, Std. Dev.

\[ g_1 \leftarrow s f v_1 + m \]

\[ g_2 \leftarrow s f v_2 + m \]
Example

- Generate random variables from this probability density function

\[ f(x) = \begin{cases} 
\sin x, & \text{if } 0 \leq x \leq \frac{\pi}{4} \\
\frac{-4x + \pi + 8}{8\sqrt{2}}, & \text{if } \frac{\pi}{4} < x \leq 2 + \frac{\pi}{4} \\
0, & \text{otherwise}
\end{cases} \]
Example (cont.)

\[ h(x) = \begin{cases} 
\frac{1}{2 + \pi/4}, & \text{if } 0 \leq x \leq 2 + \pi/4 \\
0, & \text{otherwise}
\end{cases} \]

\[ \delta = \frac{2 + \pi/4}{\sqrt{2}/2} \]

\[ \delta h(x) = \begin{cases} 
\sqrt{2}/2, & \text{if } 0 \leq x \leq 2 + \pi/4 \\
0, & \text{otherwise}
\end{cases} \]

So \( \delta h(x) \geq f(x) \) for all \( x \).
Example (cont.)

<table>
<thead>
<tr>
<th>$x_i$</th>
<th>$u_i$</th>
<th>$u_i \delta h(x_i)$</th>
<th>$f(x_i)$</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.860</td>
<td>0.975</td>
<td>0.689</td>
<td>0.681</td>
<td>Reject</td>
</tr>
<tr>
<td>1.518</td>
<td>0.357</td>
<td>0.252</td>
<td>0.448</td>
<td>Accept</td>
</tr>
<tr>
<td>0.357</td>
<td>0.920</td>
<td>0.650</td>
<td>0.349</td>
<td>Reject</td>
</tr>
<tr>
<td>1.306</td>
<td>0.272</td>
<td>0.192</td>
<td>0.523</td>
<td>Accept</td>
</tr>
</tbody>
</table>

Two samples from $f(x)$ are 1.518 and 1.306