Introduction to Parallel & Distributed Computing

Discrete Search & Termination Detection

Lecture 15, Spring 2014

Instructor: 罗国杰

gluo@pku.edu.cn

In the Last Lecture …

- **Parallel DFS**
  - **distribute/request work in a stack**
- **Parallel best-first search**
  - **distribute/request work in a priority queue**

In the Last Lecture …

- **Parallel DFS**
  - **Work-splitting strategies**
    - **Send nodes near the bottom of the stack**
      - suitable if the search space is uniform
    - **Send nodes near the cutoff depth**
      - if a strong heuristic is available (to order successors so that goal nodes move to the left of the state-space tree)
    - **Send half the nodes between the bottom of the stack and the cutoff depth**
      - suitable whenever the search space is uniform or highly irregular

In the Last Lecture …

- **Parallel DFS**
  - **Load-balancing schemes**
    - **Asynchronous round robin**
      - work requests are generated independently
      - 2+ processors may request work from the same donor processor
    - **Global round robin**
      - The *target* is store is a shared variable (shared address space) or at a designated processor (message passing)
      - Work requests are evenly distributed, but there is contention for access to target
    - **Random polling**
      - Work requests are evenly distributed

Distributed Termination Detection

- **Suppose we only want to print one solution**
- **We want all processes to halt as soon as one process finds a solution**
- **This means processes must periodically check for messages**
  - **Every process calls** MPI_Iprobe **every time search reaches a particular level (such as the cutoff depth)**
  - **A process sends a message after it has found a solution**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

MPI_Iprobe

- **Nonblocking test for a message**

```
int MPI_Iprobe(
    int source,
    int tag,
    MPI_Comm comm,
    int *flag,
    MPI_Status *status);
```

- **flag: true if a message with the specified source, tag, and communicator is available**
- **It is not necessary to receive a message immediately after it has been probed for, and the same message may be probed for several times before it is received.**

Simple (Incorrect) Algorithm

- **A process halts after one of the following events has happened:**
  - **It has found a solution and sent a message to all of the other processes**
  - **It has received a message from another process**
  - ~~**It has completely searched its portion of the state space tree**~~

Why Algorithm Fails

- **If a process calls** MPI_Finalize **before another active process attempts to send it a message, we get a run-time error**
- **How this could happen?**
  - **A process finds a solution after another process has finished searching its share of the subtrees**

  OR
  - **A process finds a solution after another process has found a solution**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Distributed Termination Problem

- **Distributed termination problem: Ensuring that**
  - **all processes are inactive AND**
  - **no messages are en route**
- **A number of algorithms have been proposed.**
  - **Token termination detection**
    - **[Dijkstra, Seijen, Gasteren, 1983]**
  - **Tree-based termination detection**
    - **[Rokusawa, Ichiyoshi, Chikayama, Nakashima, 1988]**
  - **…**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Token Termination Detection: A Simplified Scenario

- **Assume that all processors are organized in a logical ring.**
- **A simplified scenario**
  - **Once a processor goes idle, it never receives more work**
  - **Assume, for now that message passing can only happen from P_i to P_j if $j > i$.**
- **Algorithm**
  - **Processor P_0 initiates a token on the ring when it goes idle.**
  - **Each intermediate processor receives this token and forwards it when it becomes idle.**
  - **When the token reaches processor P_0, all processors are done.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Token Termination Detection

- **Assume that all processors are organized in a logical ring.**
  - **Work transfers allowed in the system are from processor P_j to P_i such that $i < j$**
- **Algorithm**
  - **When P_0 goes idle, it colors itself *white* and initiates a *white* token.**
  - **If P_j sends work to P_i and $j > i$ then processor P_j becomes *black*.**
  - **If processor P_i has the token and P_i is idle, it passes the token to P_{i+1}.**
    - **If P_i is *black*, then the color of the token is set to *black* before it is sent to P_{i+1}.**
    - **If P_i is *white*, the token is passed unchanged.**
  - **After P_i passes the token to P_{i+1}, P_i becomes *white*.**
  - **After the token passes P_0, the token becomes *white*.**
  - **Terminates when processor P_0 receives a *white* token and is itself idle.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Token Termination Detection

- **There are modified schemes when more information is passed**
  - **For example, to keep track of #messages in the communication channels**
  - **For example, to transfer the current best value searched**

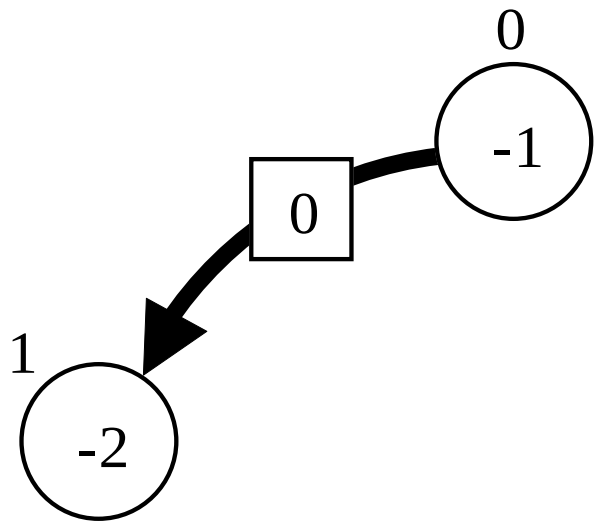
Dijkstra et al.'s Algorithm

- Each process has a color and a message count
  - Initial color is white
  - Initial message count is *0*
- A process that sends a message turns black and increments its message count
- A process that receives a message turns black and decrements its message count
- If all processes are white and sum of all their message counts are *0*, there are no pending messages and we can terminate the processes

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Dijkstra et al.'s Algorithm (cont.)

- **Organize processes into a logical ring**
- **Process *0* passes a token around the ring**
- **Token also has a color (initially white) and count (initially *0*)**



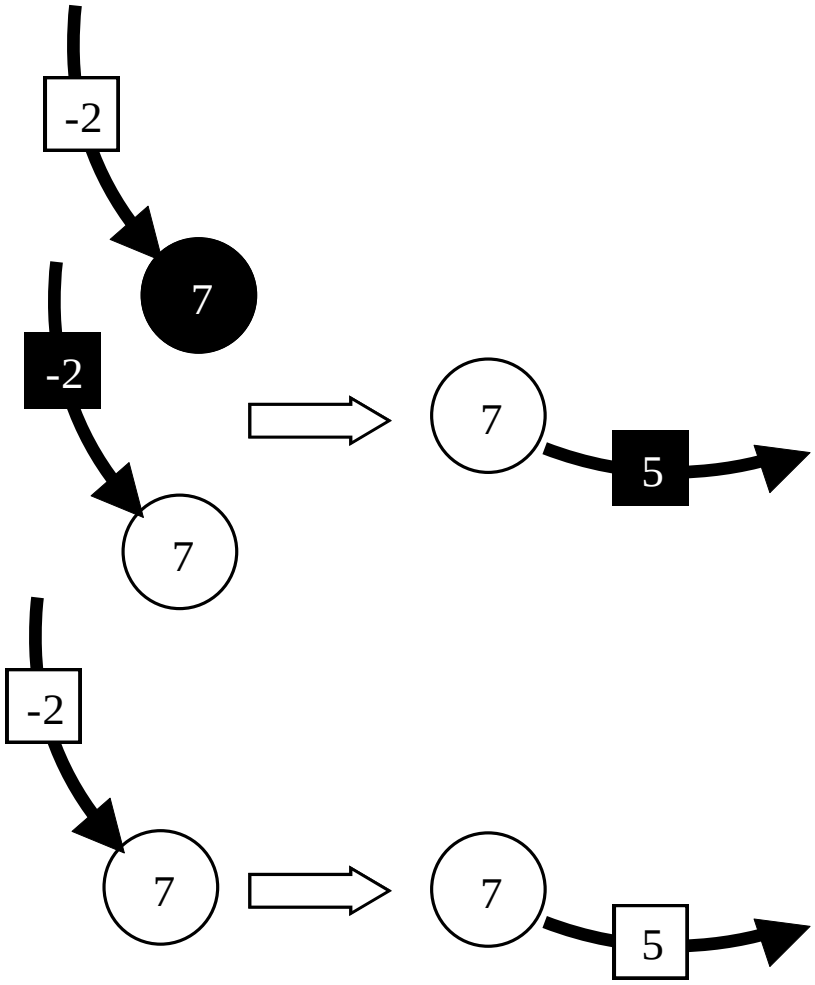
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Dijkstra et al.'s Algorithm (cont.)

- **A process receives the token**
  - **If process is black**
    - **Process changes token color to black**
    - **Process changes its color to white**
  - **Process adds its message count to token's message count**
- **A process sends the token to its successor in the logical ring**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Dijkstra et al.’s Algorithm (cont.)



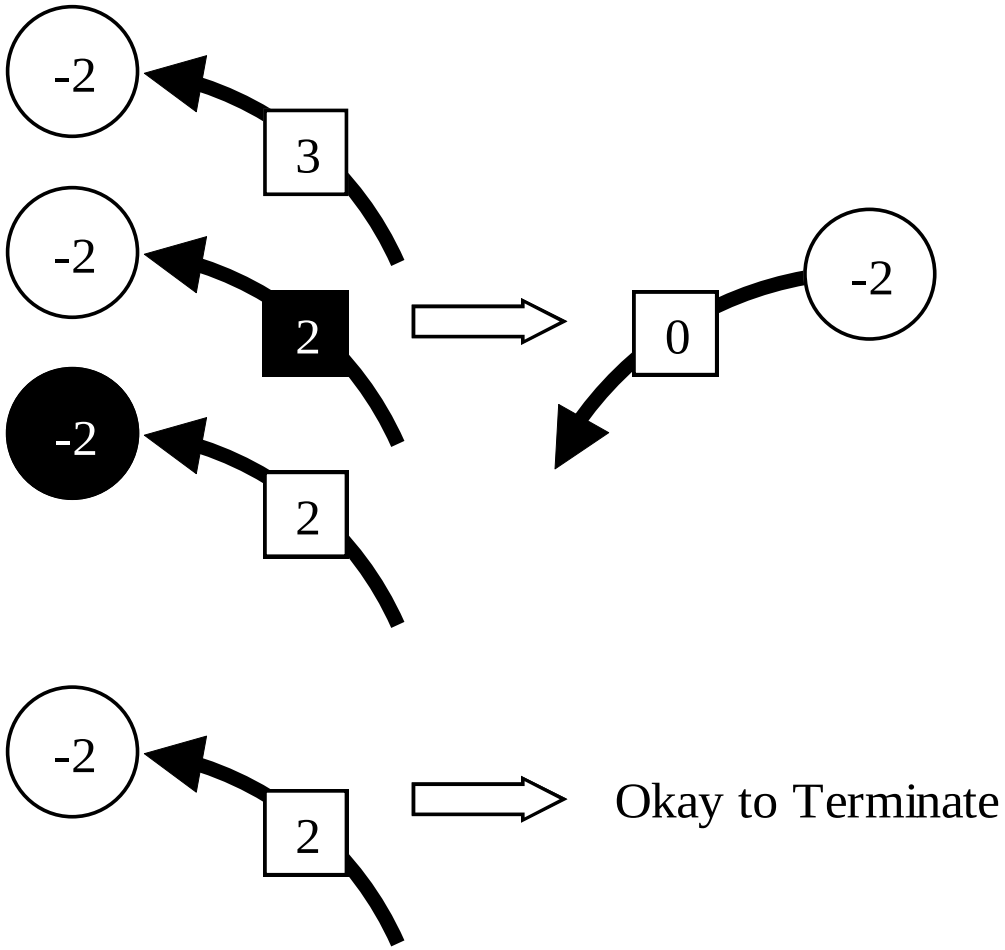
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Dijkstra et al.'s Algorithm (cont.)

- **Process *0* receives the token**
  - **Safe to terminate processes if**
    - **Token is white**
    - **Process *0* is white**
    - **Token count + process *0* message count = *0***
  - **Otherwise, process *0* must probe ring of processes again**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Dijkstra et al.'s Algorithm (cont.)



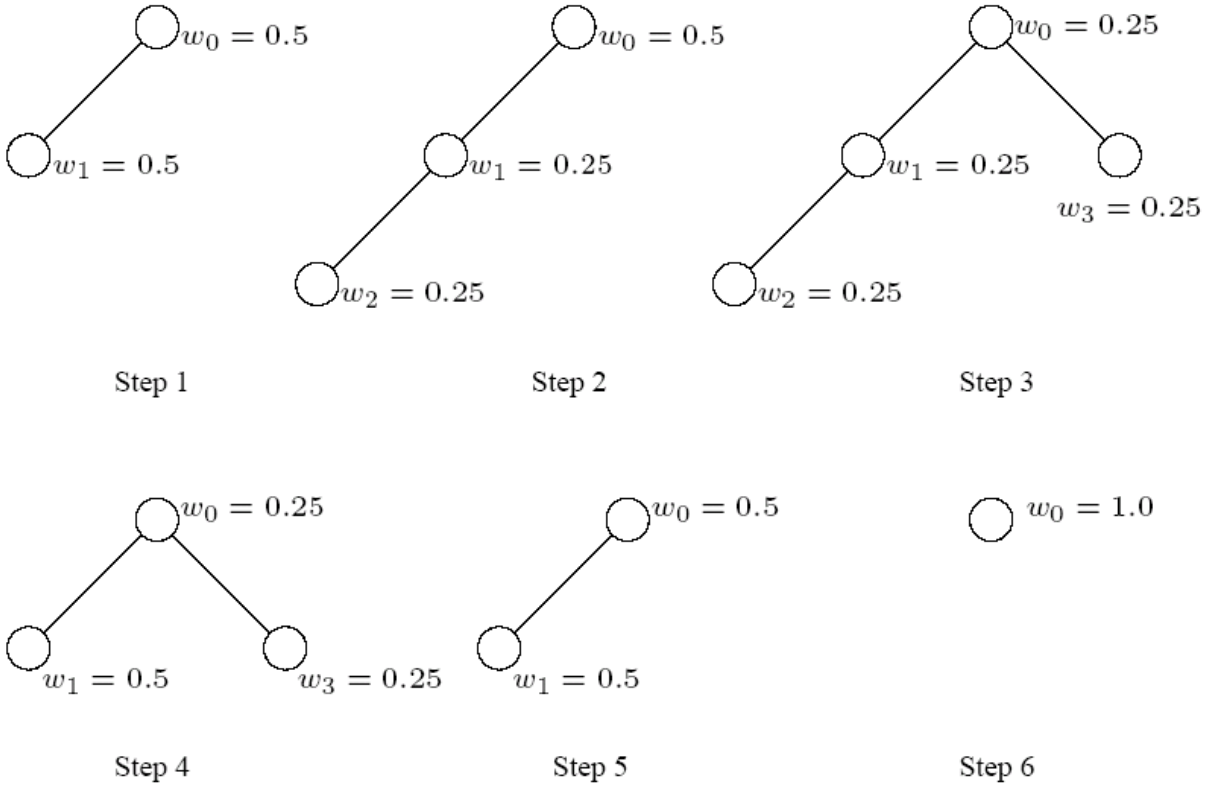
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Tree-Based Termination Detection

- **Algorithm**
  - **Associate weights with individual workpieces. Initially, processor P_0 has all the work and a weight of one.**
  - **Whenever work is partitioned, the weight is split into half and sent with the work.**
  - **When a processor gets done with its work, it sends its parent the weight back.**
  - **Termination is signaled when the weight at processor P_0 becomes one again.**
- **Note that underflow and finite precision are important factors associated with this scheme.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Tree-Based Termination Detection



Steps 1-6 illustrate the weights at various processors after each work transfer

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Introduction to Parallel & Distributed Computing

Distributed Random Number Generation

Lecture 15, Spring 2014

Instructor: Guojie Luo

(gluo@pku.edu.cn)

Outline

- **Monte Carlo method**
- **Parallel (uniform) random number generators**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

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**

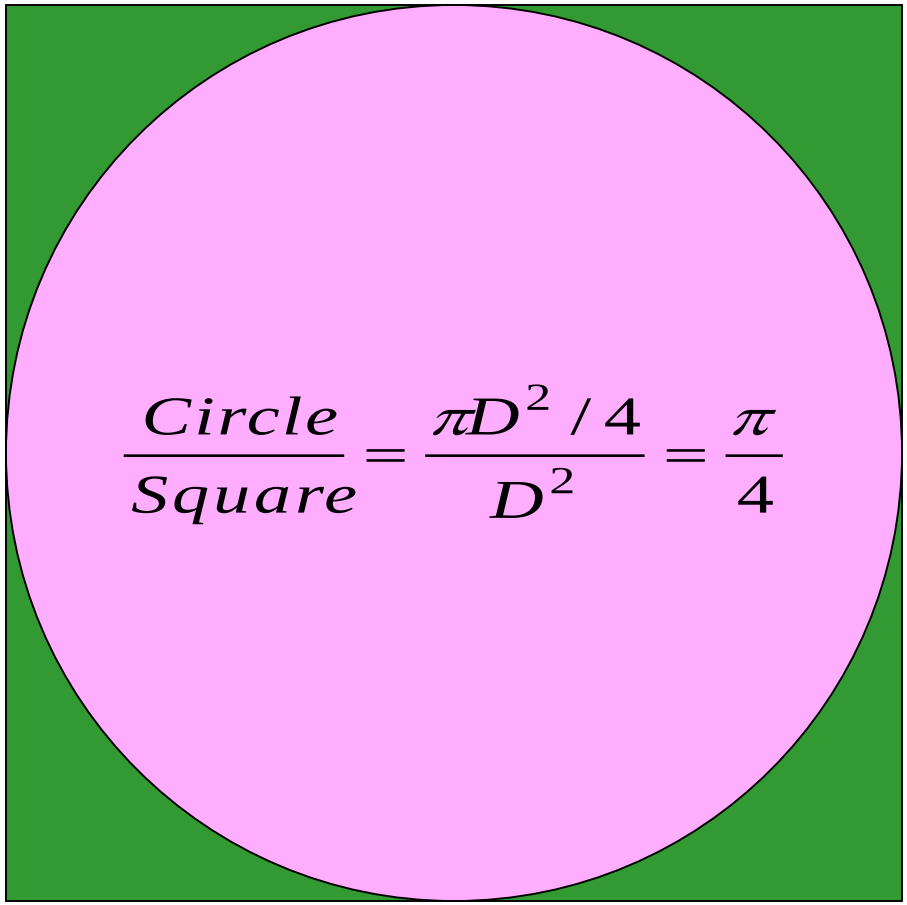
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

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**

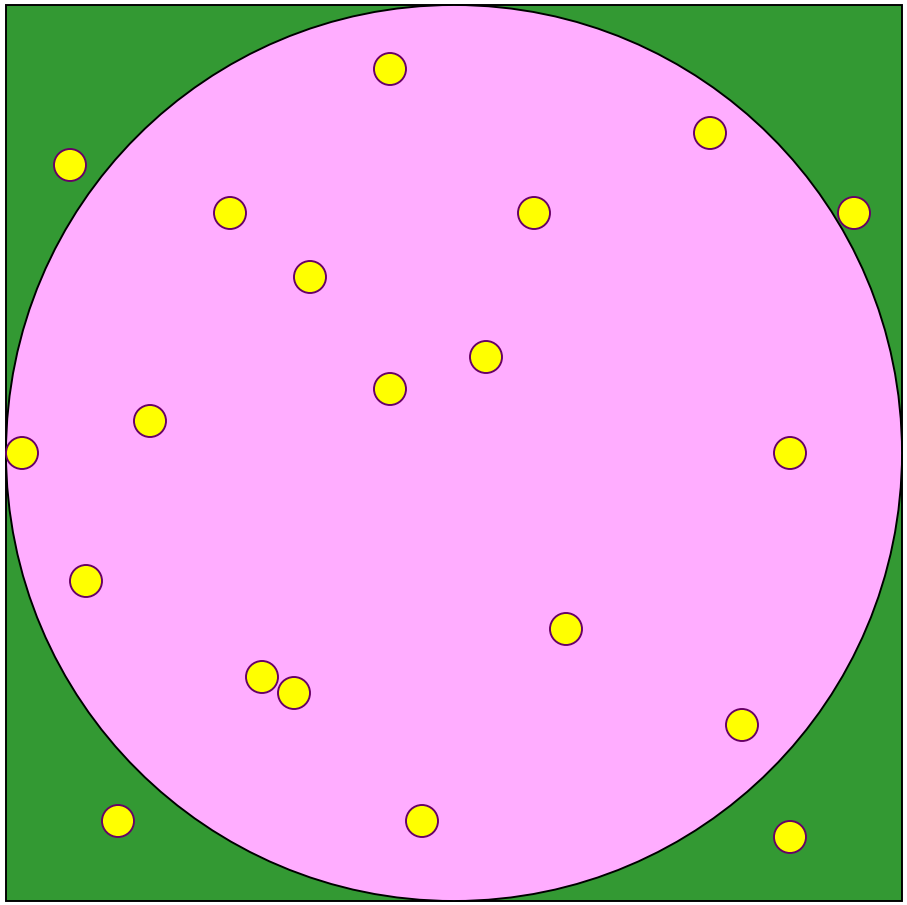
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Example of Monte Carlo Method



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Example of Monte Carlo Method



$$\frac{16}{20} \approx \frac{\pi}{4} \Rightarrow \pi \approx 3.2$$

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Relative Error

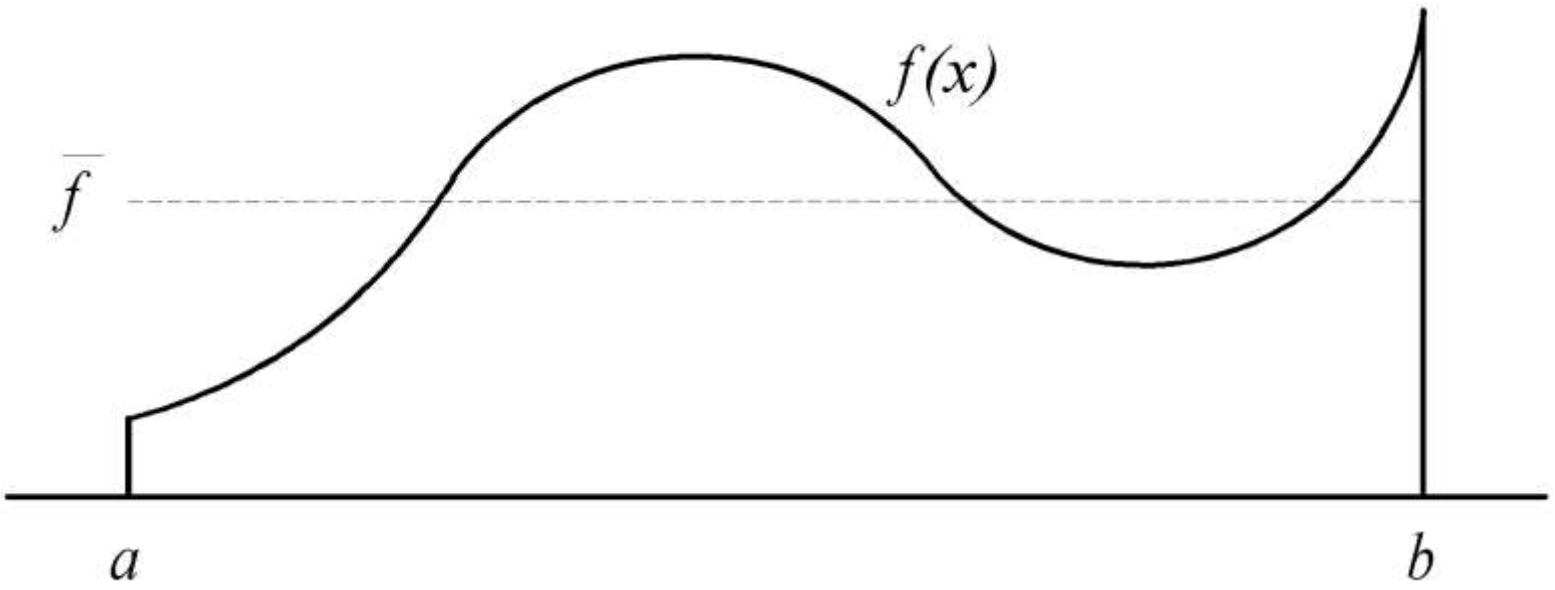
- **Relative error is a way to measure the quality of an estimate**
- **The smaller the error, the better the estimate**
- **Relative error =** $|e-a|/a$
  - a**: actual value**
  - e**: estimated value**

Increasing Sample Size Reduces Error

n	*Estimate*	*Error*	$1/(2n^{1/2})$
10	2.40000	0.23606	0.15811
100	3.36000	0.06952	0.05000
1,000	3.14400	0.00077	0.01581
10,000	3.13920	0.00076	0.00500
100,000	3.14132	0.00009	0.00158
1,000,000	3.14006	0.00049	0.00050
10,000,000	3.14136	0.00007	0.00016
100,000,000	3.14154	0.00002	0.00005
1,000,000,000	3.14155	0.00001	0.00002

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

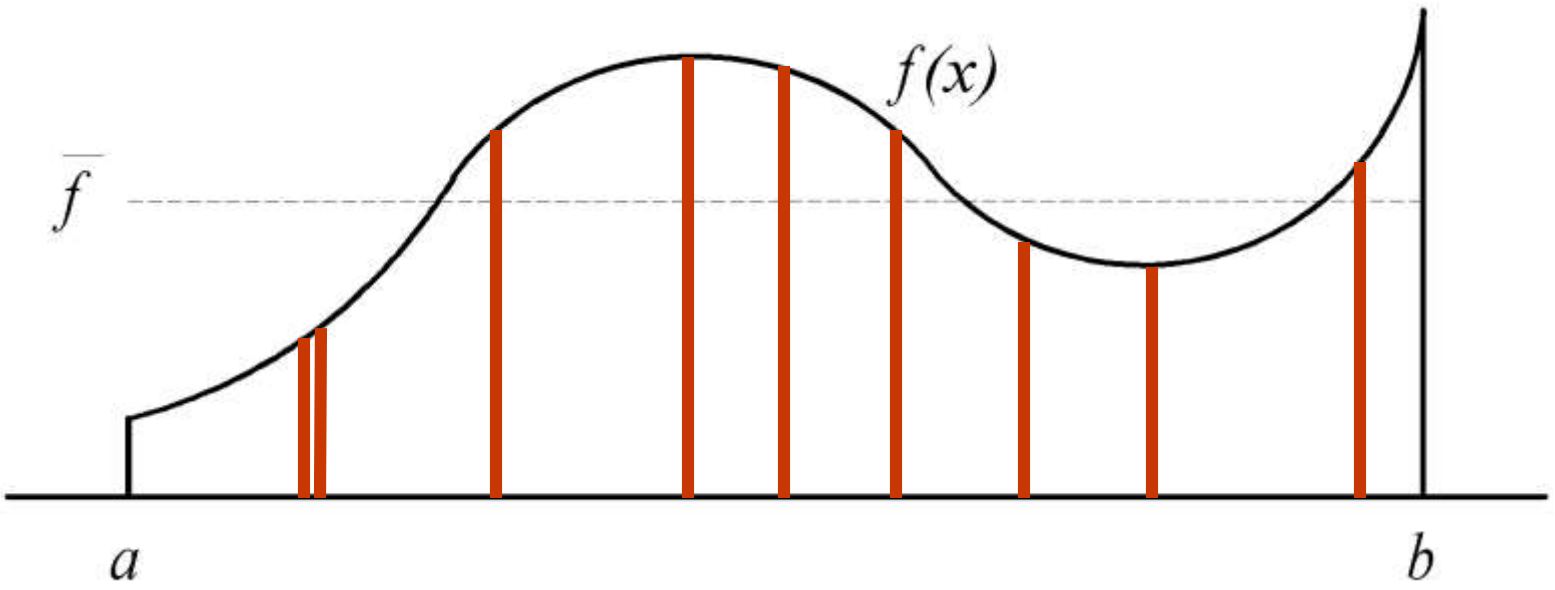
Mean Value Theorem



$$\int_a^b f(x)dx = (b-a)\,\bar{f}$$

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

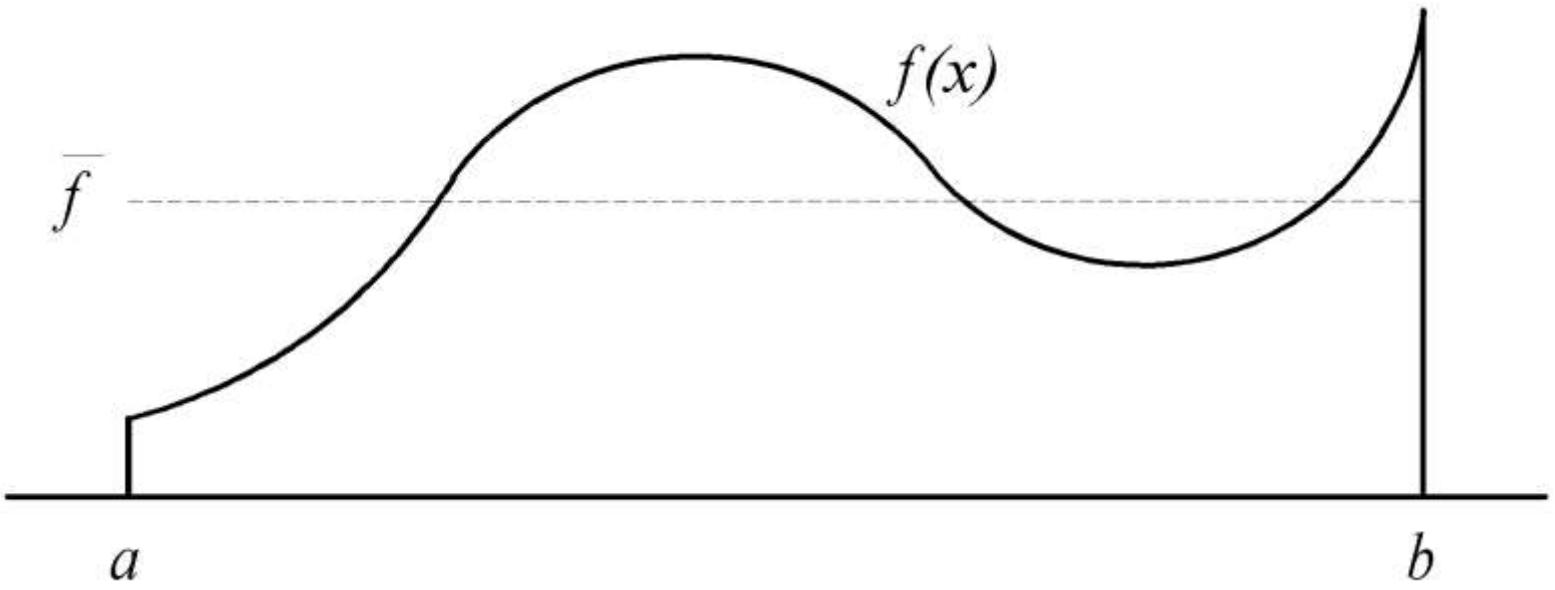
Estimating Mean Value



The expected value of $(1/n)(f(x_0) + \ldots + f(x_{n-1}))$ is $\bar{f}$

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

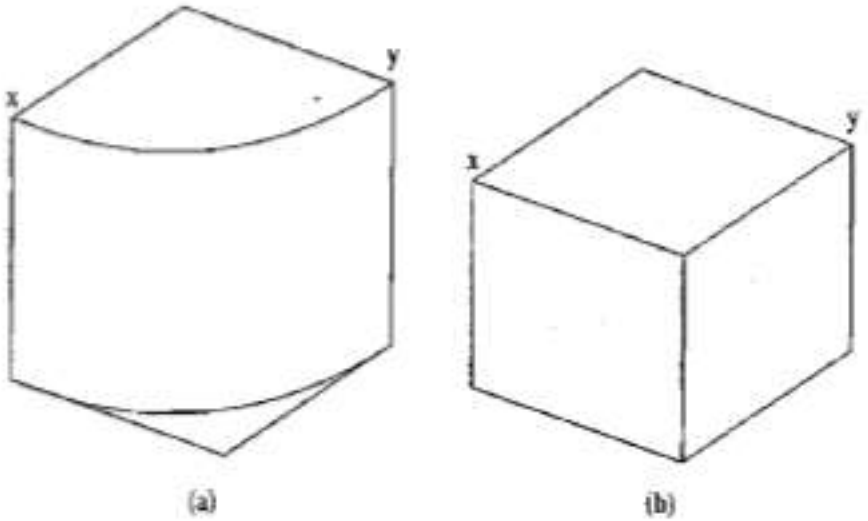
Why Monte Carlo Works



$$\int_a^b f(x)dx = (b-a)\,\bar{f} \approx (b-a)\frac{1}{n}\sum_{i=0}^{n-1} f(x_i)$$

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Interpreting the π Estimation Algorithm



use Monte Carlo to compute volume(a)
volume(a) / volume(b) = π/4

Why Monte Carlo is Effective

- **Error in Monte Carlo estimate decreases by the factor** $1/n^{1/2}$
  - **(#sample: n → 4n) ⇒ (error: ε → ε/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**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Monte Carlo and Parallel Computing

- **Monte Carlo methods often amenable to parallelism**
  - **Find an estimate about p times faster**

    OR
  - **Reduce error of estimate by $p^{1/2}$**
- **Based on the assumption that the random numbers are statistically independent**
- **A principal challenge has been the development of good parallel random number generators**
- **It is widely claimed that half of all super-computer cycles are dedicated to Monte Carlo calculations**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

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 lecture "random" is shorthand for "pseudo-random"**
- **"RNG" means "random number generator"**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

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**

[Coddington, "Random number generators for parallel computers," Technical Report, Syracuse University 1997.]

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

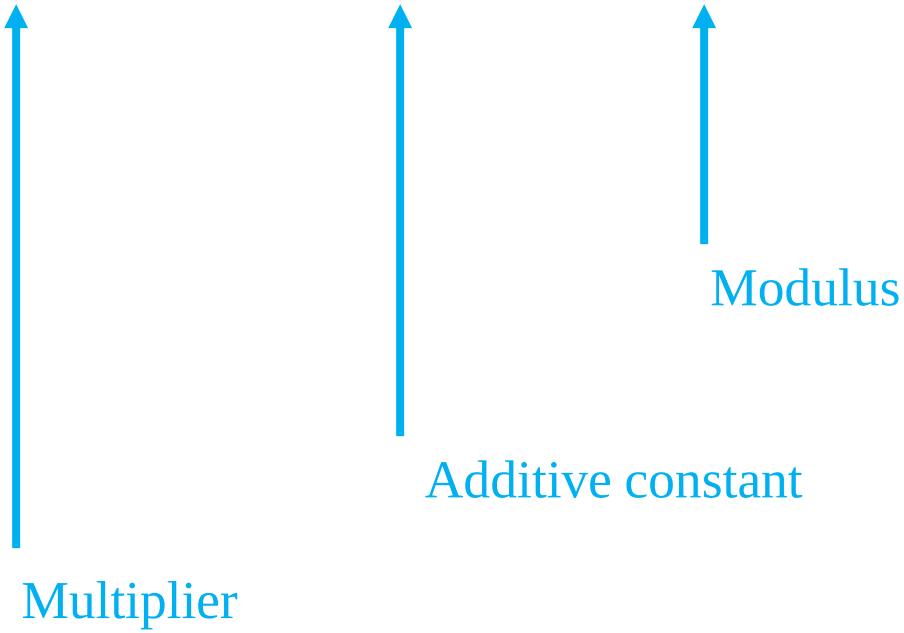
No RNG is Ideal

- **Finite precision arithmetic ⇒ finite number of states ⇒ cycles**
  - **Period = length of cycle**
  - **If period > number of values needed, effectively acyclic**
- **Reproducible ⇒ correlations**
- **Often speed versus quality trade-offs**
  - **Since the time needed to generate a random number is typically a small part of the overall computation time of a program, speed is much less important than quality**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Linear Congruential (线性同余) RNGs

$$X_i = (a \times X_{i-1} + c) \bmod M$$



Sequence depends on choice of seed, X_0

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Period of Linear Congruential RNG

- **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**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

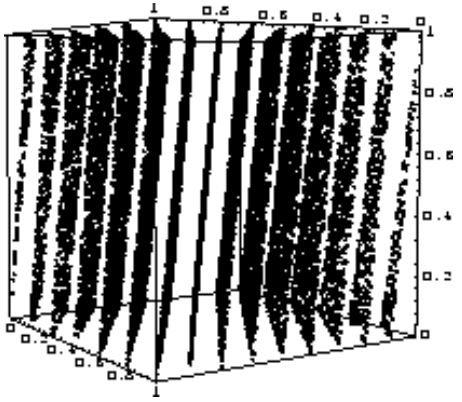
Producing Floating-Point Numbers

- **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**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

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**
  - **Example: the infamous RANDU (IBM)**
    - $M = 2^{31}$, $a = 65539$, $c = 0$, $X_0 = 1$



- **Despite these flaws,**
  - **Linear congruential generators with 48 or more bits of precision and carefully chosen parameters "work very well for all known applications, at least on sequential computers"** [Coddington 1997]

modified from Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Lagged Fibonacci RNGs

$$X_i = X_{i-p} * X_{i-q}$$

- p **and** q **are lags,** $p > q$
- *** is any binary arithmetic operation**
  - **Addition modulo** M
  - **Subtraction modulo** M
  - **Multiplication (of odd integers) modulo** M
  - **Bitwise exclusive or**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Properties of Lagged Fibonacci RNGs

- **Require p seed values (more storage)**
- **Careful selection of seed values, p and q, can result in very long periods and good randomness**
  - **[Coddington 1997] recommends setting (p,q) to at least (1279,1063)**
- **Maximum period attainable with a b-bit M**
  - **Exclusive or:** $2^p - 1$
  - **Addition/subtraction:** $(2^p - 1)2^{b-1}$
  - **Multiplication:** $(2^p - 1)2^{b-3}$

modified from Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Ideal Parallel RNGs

- **In addition to all properties of sequential RNGs,**
- **No correlations among numbers in different sequences**
- **Scalability**
  - **It should be possible to accommodate a large number of processes, each with its own stream(s)**
- **Locality**
  - **A process should be able to spawn a new sequence of random numbers without interprocess communication**

modified from Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Parallel RNG Designs

- **Manager-worker**
- **Leapfrog**
- **Sequence splitting**
- **Independent sequences**

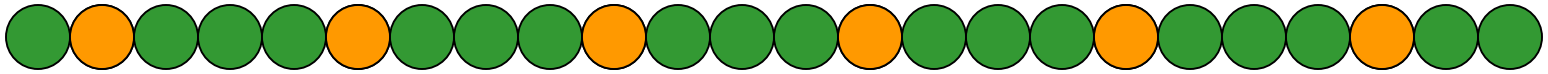
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Manager-Worker Parallel RNG

- **Manager process generates random numbers**
- **Worker processes consume them**
- **Disadvantages**
  - **Not scalable to an arbitrary number of processes**
    - **It may be difficult to balance the speed of the random number producer with the speed of the consumers of these numbers**
  - **Does not exhibit locality**
    - **It is communication-intensive**

modified from Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Leapfrog Method



Process with rank 1 of 4 processes

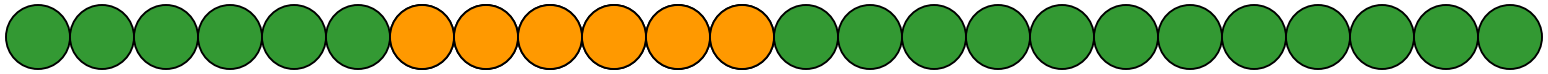
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Properties of Leapfrog Method

- **(Analog to cyclic decomposition)**
- **Easy modify linear congruential RNG to support jumping by *p***
- **Can allow parallel program to generate same tuples as sequential program**
  - **For example, to generate the *2*-tuples (X_{2r}, X_{2r+1})**
- **Does not support dynamic creation of new random number streams**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Sequence Splitting



Process with rank 1 of 4 processes

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Properties of Sequence Splitting

- **(Analog to block decomposition)**
- **Forces each process to move ahead to its starting point**
  - **It may take a long time**
  - **This only needs to be done at the initialization**
- **Can be modified to support dynamic creation of new sequences**
  - **For example, a process creating a new stream could give up half of it section to the new stream**

modified from Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Independent Sequences

- **Run sequential RNG on each process**
- **Start each with different seed(s) or other parameters**
- **Example: linear congruential RNGs**
  - **Works well for up to 100 streams with different additive constants** [Percus & Kalos, 1989]
- **Example: lagged Fibonacci RNGs**
  - **SPRNG library provides around 2^{1008} distinct streams**
- **Supports goals of locality and scalability**

modified from Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.