Monte Carlo Simulation Examplar

CSInParallel Project

September 18, 2014

CONTENTS

Text by Devin Bjelland and Libby Shoop, Macalester College

Accompanying instructor videos by Dave Valentine, Slippery Rock University

Monte Carlo simulations are a class of algorithms that are quite easy to convert from their original sequential solutions to corresponding parallel or distributed solutions that run much faster. This module introduces these type of algorithms, providing some examples with C++ code for both the original sequential version and the parallelized OpenMP version.

Hardware and Software Needed

- You will need access to a multicore computer with a C/C++ compiler that enables compilation with OpenMP.
- If you want to try some of the other examples for Message Passing using MPI you will need access to a cluster of computers with an MPI library installed.
- If you want to try some of the other examples for CUDA on GPUs, you will need acces to a computer with a CUDA-capable nVIDIA GPU and you will need the nVIDIA CUDA Toolkit installed.

This document contains several sections with example C++ code to explain Monte Carlo methods and how to parallelize them using the OpenMP library. The last three sections contain exercises that you can try and explain a more advanced topic for ensuring greater accuracy.

CHAPTER

ONE

WHAT ARE MONTE CARLO METHODS?

1.1 Overview

Monte Carlo Methods are a class of numerical methods which use repeated simulations to obtain a random sampling from an underlying unknown probability distribution. Monte Carlo methods rely on repeated independent simulations using random numbers. This makes them very well suited to parallel and distributed solutions, because you can split the work of completing repeated independent calculations among multiple processing units. Monte Carlo methods are often employed when there is no closed form solution or deterministic solution algorithm to the underlying problem. As this sort of problem is quite common, Monte Carlo methods are used in a wide variety of fields–from computational chemistry to finance. Monte Carlo methods were first developed for general applications such as these by Nicholas Metropolis and Stanislaw Ulim in 1949. In 1987, Metropolis wrote an interesting recounting of the beginnings of the use of these methods.

1.2 An Interesting Classic Example

To make this concrete, let's illustrate a classical mathematical problem that can be solved with sampling of random numbers: estimating the value of π using geometric properties of a circle and a square.

Imagine you have a circular target inside a larger square target. You want to find the probability that if you throw a dart it will hit the inner target. To run a 'Monte Carlo Simulation' to solve this problem, you would simply throw a bunch of darts at the target and record the percentage that land in the inner target. More concretely, we simulate the throw of the dart by generating a pair of random numbers between 0 and the length of the side of the square, each one representing the x and y coordinate of the location of the dart. Then we determine if that simulated dart location falls within the circle or not.

We can extend this idea to approximate π quite easily. Suppose the inner target is a circle that is inscribed inside a square outer target. Since the sides of the square are twice the radius of the circle, we have that the ratio, ρ of the area of the circle to the area of the square is

$$\rho = \frac{\pi r^2}{(2r)^2}$$

We can rearrange this formula to solve for π as follows:

$$\pi = \frac{\rho\,(2r)^2}{r^2} = 4\rho$$

If we can determine the ratio of the area of the circle to the total area of the square inscribing it, we can approximate π.

As it happens, we can empirically calculate a value for the ratio of the area of the circle to the area of the square with a Monte Carlo simulation. We pick lots of random points in the square and the ratio is the number of points inside the

circle divided by the total number of points. As we sample more and more points, the ratio of the number of points that fall inside the circle to the total number of points tried is equal to the ratio of the two areas. We multiply this by 4 and we have our estimate of π.

We can simplify this slightly by using a $1/4$ circle of radius 1, as shown in the following illustration.

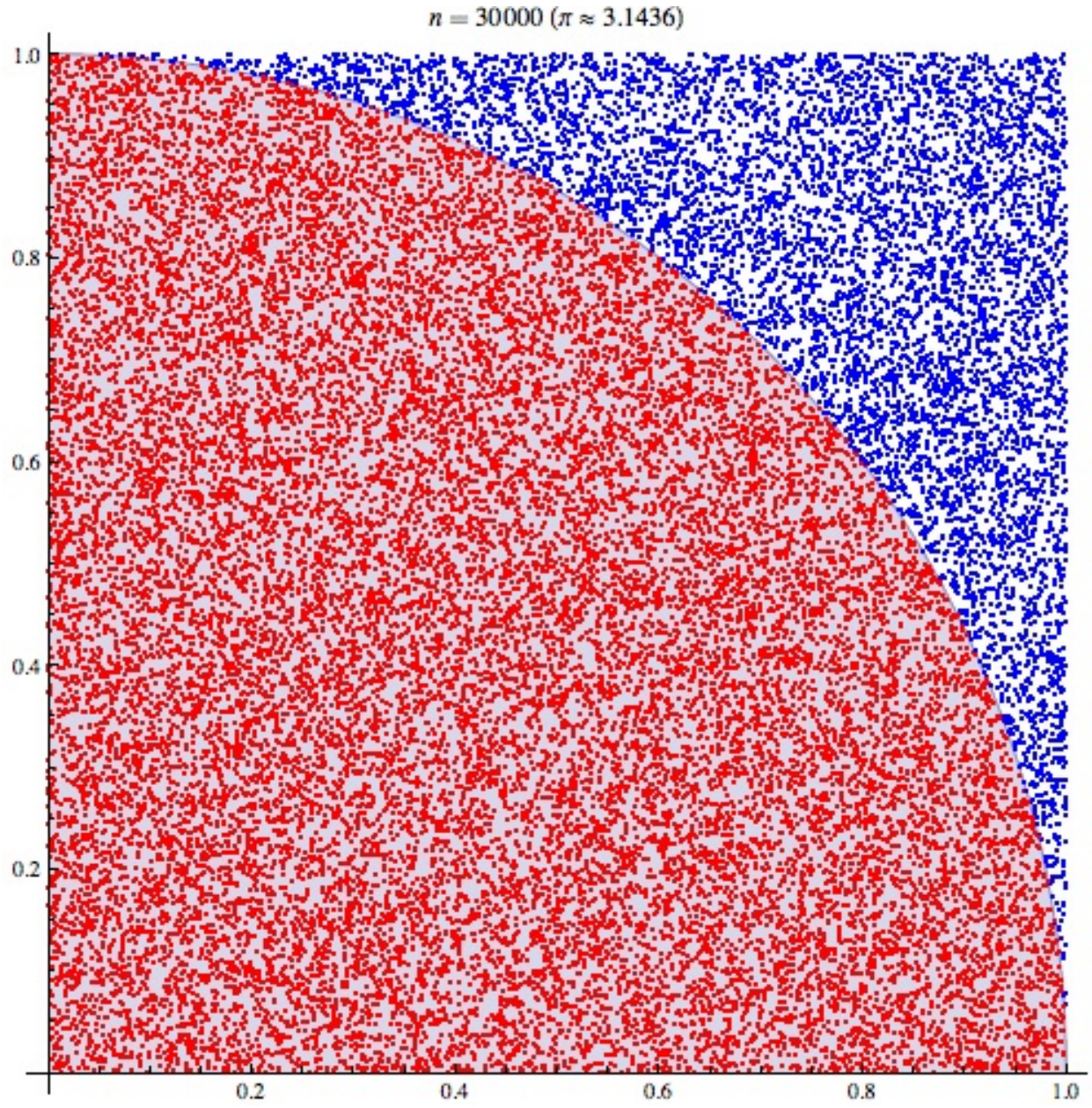


Figure 1.1: Estimating π via random sampling of points. Attribution: By CaitlinJo [CC-BY-3.0 (http://creativecommons.org/licenses/by/3.0)], via Wikimedia Commons.

See Also:

How FogBugz uses Monte Carlo Simulations to estimate the time their software projects will take

1.3 Simulating Card Games for Fun and Profit

The original motivating example that Monte Carlo methods draw their name from is gambling and game playing. In this module, we develop parallel algorithms that approximate the probabilities of various outcomes in card games and the roulette wheel.

CHAPTER

TWO

ABOUT RANDOMNESS

2.1 How Computers Generate Random Numbers

Algorithms that implement Monte Carlo methods require a source of of randomness. If we are writing a serial algorithm, we can simply use the standard random number generator, for example, rand() in C. Computers (unless specifically fitted with a chip which gathers randomness from some other source) cannot produce true random numbers. Rather, the standard library random number generator takes some relatively random 32-bit or 64-bit integer input, called the seed, and transforms it sequentially to produce a stream of psuedo-random numbers. A random number generator is created with the seed as input to a function called *srand()*, and each time a new random number is requested using the *rand()* function, the integer pattern is changed to produce a new number. The sequence of numbers produced by such a pseudorandom number generation algorithm try to approximate a uniform distribution of numbers, each one statistically independent from the others.

Note: It is important to realize that a pseudorandom number generator function such as rand() creates the same sequence of numbers every time for a given input seed, which is typically a very large integer. In C and C++, we often use the function time() create the seed integer, because it returns the number of seconds since January 1, 1970. When running a sequential program multiple times, this seed would be different each time the program was run and the pattern of random numbers generated would be different.

CHAPTER

THREE

TESTING OUT RANDOM NUMBER GENERATORS: FLIP A COIN MANY TIMES

A simple way to see how well a random number generator is working is to simulate flipping a coin over and over again for many trials.

Let's look at some C/C++ code to do this. The listing below shows how we can use srand() to seed our random number generator with a large integer and then make many calls to rand() (or rand_r() on linux/unix) to obtain a series of random integers. If the integer is even, we call it a 'head' coin flip, otherwise it is a 'tail'. This code sets up trials of coin flips with ever increasing numbers of flips. It also calculates the Chi Square statistic using the number of heads and number of tails. A rule of thumb in the case of heads and tails is that if the Chi-Square value is around 3.8 or less, we have a good random distribution of the even and odd values. We want to verify that the random number generator provides such an independent distribution.

See Also:

For more details about chi square calculations and how they measure whether a set of values flows an independent distribution, please see A Chi-square tutorial, which shows an example for coin-flipping.

There are many other examples you can find by searching on the web.

In the main() there is a while loop that conducts the trials of coin flips. Each trial is conducted by obtaining random numbers in the for loop on line 60. You can download the file `coinFlip_seq.cpp` and try this code below yourself. You should note that the longer trials with many coin flips take a somewhat long time (on the order of 20 seconds, depending on your machine).

In the next section, we will look at parallelizing code using threads and OpenMP, then we will explore how we can conduct the coin-flipping simulation in parallel so that it runs considerably faster.

```
1  /*
2    Original code provided by Dave Valentine, Slippery Rock University.
3    Edited by Libby Shoop, Macalester College.
4  */
5  //
6  // Simulate many coin flips with rand() to determine how
7  // random the values are that are returned from each call.
8  //
9
10 #include <stdio.h>         // printf()
11 #include <stdlib.h>        // srand() and rand()
12 #include <time.h>         // time()
13
```

```
14  //const int MAX = 1<<30; //1 gig
15
16  //Standard chi sqaure test
17  double chiSq(int heads, int tails) {
18      double sum = 0;                                          //chi square sum
19      double tot = heads+tails;                  //total flips
20      double expected = 0.5 * tot;          //expected heads (or tails)
21
22      sum = ((heads - expected)*(heads-expected)/expected) + \
23          ((tails - expected)*(tails-expected)/expected);
24      return sum;
25  }
26
27
28  int main() {
29      int numFlips,                          //loop control
30          numHeads, numTails;          //counters
31          clock_t startTime, stopTime; //wallclock timer
32
33  /***** Initialization *****/
34
35      printf("Sequential Simulation of Coin Flip using rand()\n");
36
37      //print our heading text
38      printf("\n\n%15s%15s%15s%15s%15s%15s",
39             "Trials","numHeads","numTails","total",
40             "Chi Squared", "Time (sec)\n");
41
42      //create seed using current time
43      unsigned int seed = (unsigned) time(NULL);
44
45      //create the pseudorandom number generator
46      srand(seed);
47
48  // Try several trials of different numbers of flips, doubling how many each round.
49  //
50  // Use a unsigned int because we will try a great deal of flips for some trials.
51      unsigned int trialFlips = 256;       // start with a small number of flips
52      unsigned int maxFlips = 1073741824;  // end with a very large number of flips
53
54      // below we will double the number of trial flips and come back here
55      // and run another trial, until we have reached > maxFlips.
56      // This will be a total of 23 trials
57      while (trialFlips <= maxFlips) {
58          // reset counters for each trial
59          numHeads = 0;
60          numTails = 0;
61          startTime = clock();                //get start time for this trial
62
63      /***** Flip a coin trialFlips times ****/
64          for (numFlips=0; numFlips<trialFlips; numFlips++) {
65              // if random number is even, call it heads
66              // if (rand()%2 == 0)     // on Windows, use this
67              if (rand_r(&seed)%2 == 0) // on linux, can use this
68                  numHeads++;
69              else
70                  numTails++;
71          }
```

```
72
73          stopTime = clock();    // stop the clock
74
75          /***** Show the results  for this trial  *****/
76          printf("%15d%15d%15d%15d%15.6f%15.6f\n", trialFlips, numHeads, numTails,
77                 (numHeads+numTails), chiSq(numHeads, numTails),
78                 (double)(stopTime-startTime)/CLOCKS_PER_SEC);
79
80          trialFlips *= 2;  // double the number of flips for the next trial
81      }
82
83  /***** Finish Up *****/
84      printf("\n\n\t<<< Normal Termination >>>\n\n");
85      return 0;
86  }
```

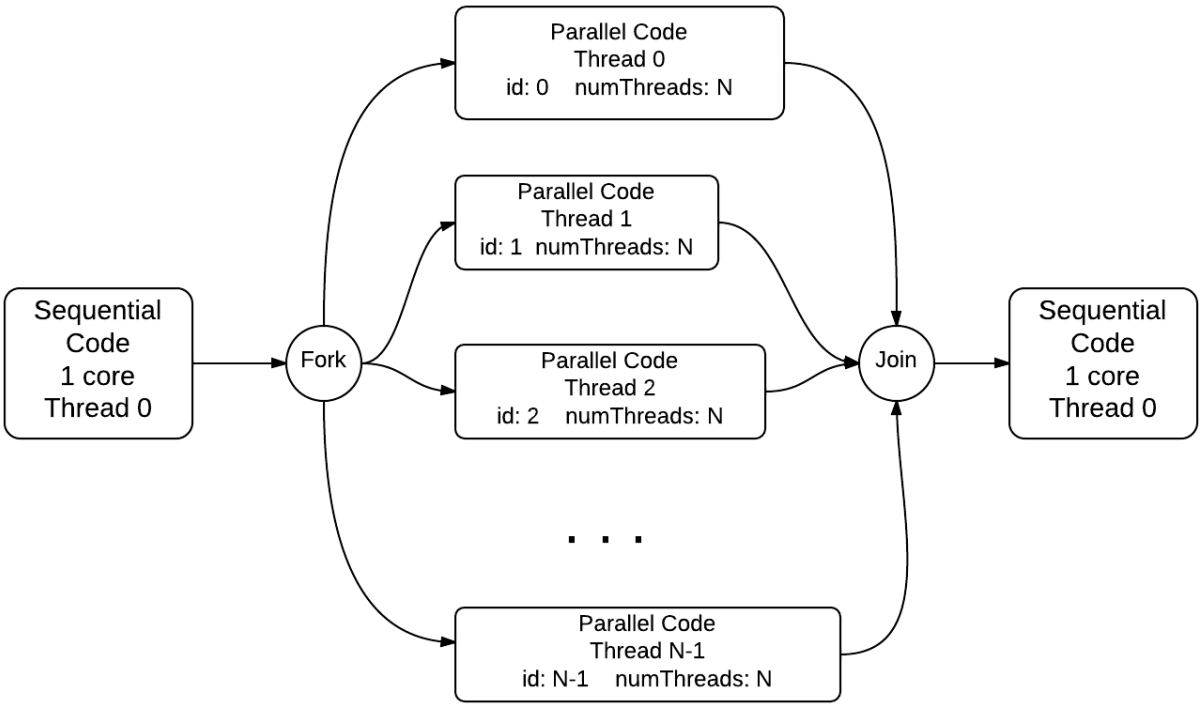
CHAPTER

FOUR

PARALLEL CODE WITH THREADS

We can make code that will run our coin flipping simulation faster, by making use of the cores available in multicore CPUs. We call this type of code parallel code, because we can designate portions of our program to run concurrently in parallel on different cores, computing part of the overall solution. In the case of flipping a coin, we can intuitively sense that it might be simple enough to designate that each core we have available could carry out some portion of the flips independently while other cores were taking care of the rest of the needed flips.

A common mechanism we use to run code on multiple cores simultaneously is by starting **threads** that can excute part of our code independently and in parallel on separate cores, sharing data values in memory if needed. When a program using threads begins execution, it is always running on a single main thread, which we conceptually label as thread 0. Then within the code we can designate that more threads should start executing in parallel along with thread 0. We call a point in the code where multiple threads are executing concurrently a **fork** of those threads. Then when they are done executing, we think of them as **joining** back with the main thread. Conceptually, this looks like this:



4.1 OpenMP: C/C++ aid for providing threads

The basic library for threading in C/C++ on linux/unix is called *pthreads*. There are several other thread libraries for other operating systems. A more convenient way to get started using threads is to use **OpenMP**, which is built into several popular C/C++ compilers as means to compile high-level directives into threaded code using an underlying threads library.

Let's take a look at a very simple example of how this works:

```
1  /*
2   * Illustration of OpenMP thread forking.
3   */
4
5  #include <stdio.h>
6  #include <omp.h>
7
8  int main(int argc, char** argv) {
9      printf("\n");
10
11 #pragma omp parallel
12     {
13         int id = omp_get_thread_num();
14         int numThreads = omp_get_num_threads();
15         printf("Hello from thread %d of %d\n", id, numThreads);
16     }
17
18     printf("\n");
19     return 0;
20 }
```

Line 11 of this code illustrates how we can designate that the main thread 0 should fork and start multiple threads simultaneously. The code within the block following that line and between the curly braces will execute independently on each thread. Lines 13 and 14 illustrate functions that are available as part of the OpenMP library, which was included on line 6. There are several other functions available, most notably one that lets you set the number of threads to use, *omp_set_num_threads*, and one that lets you time your threaded code, *omp_get_wtime*, to see how much faster it performs.

Note: When you try an example like this, you should take special note that the order in which each thread will complete *is not guaranteed*.

---

compiling: To compile a code file like this in linux/unix, you will need to add this option to gcc or g++ in your makefile or on the command line: *-fopenmp*. In and IDE like Visual Studio, you will need to set a preference on your project for the C/C++ language to enable OpenMP.

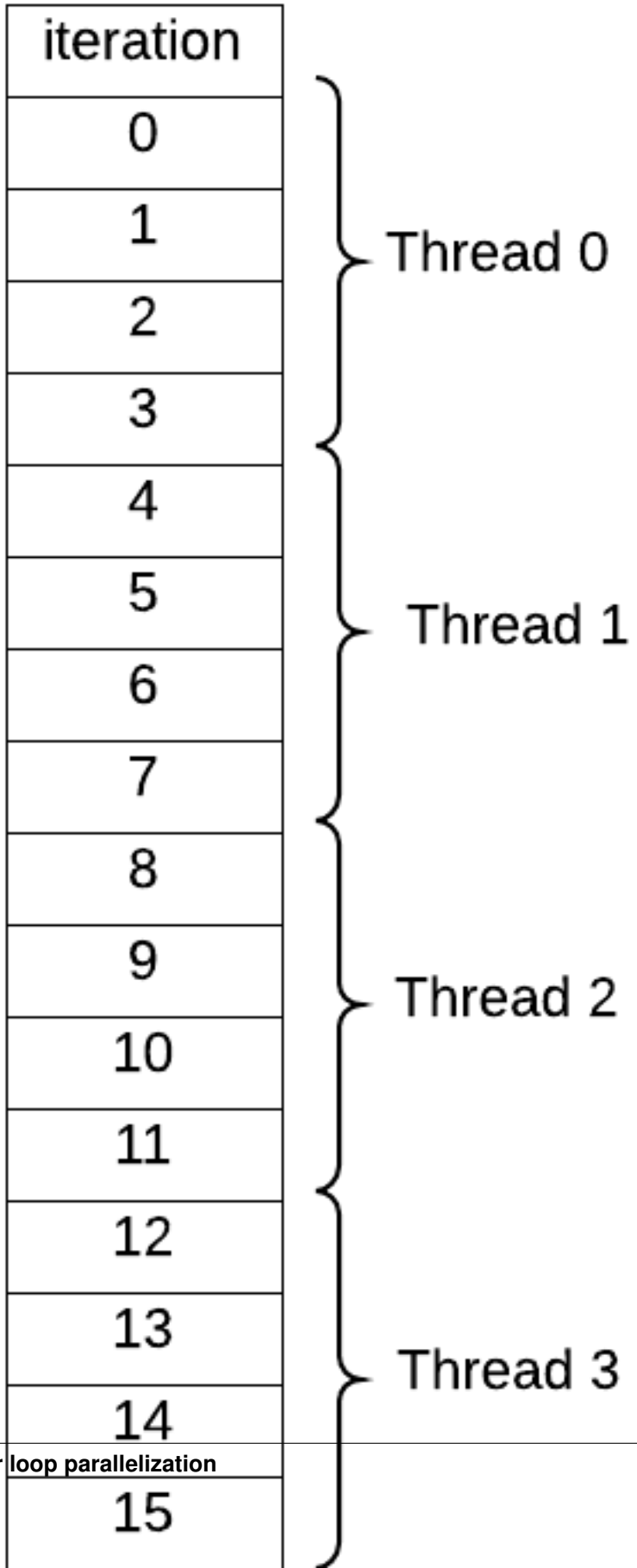
4.2 For loop parallelization

In a great deal of code examples, much of the work being performed can be found within for loops that are performing a large number of iterations, such as the coin-flipping example in the previous section. A well-used pattern in parallel programming is to split the work being done in these loops across multiple forked threads. OpenMP has a pragma for deisgnating this in the code. Here is a simple example:

```
1  /*
2   * Parallel for loop using equal chunks per thread.
3   */
```

```
4
5  #include <stdio.h>    // printf()
6  #include <stdlib.h>   // atoi()
7  #include <omp.h>      // OpenMP
8
9  int main(int argc, char** argv) {
10     const int REPS = 16;
11
12     omp_set_num_threads(4);
13
14     #pragma omp parallel for
15     for (int i = 0; i < REPS; i++) {
16         int id = omp_get_thread_num();
17         printf("Thread %d performed iteration %d\n",
18                  id, i);
19     }
20
21     printf("Main thread 0 done.\n");
22     return 0;
23 }
```

In this example, we set up a very small number of repetitions of the loop, simply to illustrate how forking threads and running the loop iterations works. The OpenMP pragma on line 14 is asking the compiler to set up an equal distribution of work for each thread, which will take place like this for the 4 threads indicated on line 12 and the 16 repetitions of the for loop:


iteration
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Thread 0
Thread 1
Thread 2
Thread 3

When running a simple example like this, you will find that each repetition will not be carried out in order from 0 through 15, as each thread will do its designated repetitions at the same time as the other threads, shceduled by the operating system on the cores available.

4.3 Next step: using OpenMP

In the next section we will see how we can use threads and OpenMP to make coin flipping faster.

CHAPTER

FIVE

COIN-FLIPPING IN PARALLEL

Now that we know a bit about how OpenMP works to provide threads that run code in parallel, let's look at how we can update our coin-flipping example. The places in this code where you see this comment:

```
    /***  OMP ***/
```

indicate where OpenMP was used to enable running the original coin-flipping code example on multiple threads, or where the code needed changes to enable running on multiple threads. Examine these places in the following code:

```
/*
  Original code provided by Dave Valentine, Slippery Rock University.
  Edited by Libby Shoop, Macalester College.
*/
//
// Simulate many coin flips with rand_r() on multiple threads
// to determine how random the values are that are returned
// from each call.
//

#include <stdio.h>        // printf()
#include <stdlib.h>       // srand() and rand()
#include <time.h>         // time()

#include <omp.h>          // OpenMP functions and pragmas


//Standard chi sqaure test
double chiSq(int heads, int tails) {
    double sum = 0;                 //chi square sum
    double tot = heads+tails;       //total flips
    double expected = 0.5 * tot;    //expected heads (or tails)

    sum = ((heads - expected)*(heads-expected)/expected) + \
        ((tails - expected)*(tails-expected)/expected);
    return sum;
}



int main() {
    int numFlips,             //loop control
        numHeads, numTails;   //counters

    /***  OMP ***/
    int nThreads;           // number of threads to use
```

```
37      double ompStartTime, ompStopTime;  // holds wall clock time
38      /***  OMP ***/
39
40
41  /***** Initialization *****/
42
43      printf("Threaded Simulation of Coin Flip using rand_r()\n");
44      /***  OMP ***/
45      nThreads = 4;      // try increasing this if you have more cores
46
47      //print our heading text
48      printf("\n\n%15s%15s%15s%15s%15s%15s%15s",
49             "Trials","numHeads","numTails","total",
50             "Chi Squared", "Time (sec)\n");
51
52
53      //create seed using current time
54      unsigned int seed = (unsigned) time(NULL);
55
56      //create the pseudorandom number generator
57      srand(seed);
58
59
60  // Try several trials of different numbers of flips doubling how many each round.
61  //
62  // Use a unsigned int because we will try a great deal of flips for some trials.
63      unsigned int trialFlips = 256;          // start with a smal number of flips
64      unsigned int maxFlips = 1073741824;     // end with a very large number of flips
65
66      // below we will double the number of trial flips and come back here
67      // and run another trial, until we have reached > maxFlips.
68      // This will be a total of 23 trials
69      while (trialFlips <= maxFlips) {
70
71          numHeads = 0;               //reset counters
72          numTails = 0;
73
74          /***  OMP ***/
75          ompStartTime = omp_get_wtime();   //get start time for this trial
76
77      /***** Flip a coin trialFlips times, on each thread in parallel,
78       *     with each thread getting its 1/4 share of the total flips.
79       *****/
80
81  /***  OMP ***/
82  #pragma omp parallel for num_threads(nThreads) default(none) \
83          private(numFlips, seed) \
84          shared(trialFlips) \
85          reduction(+:numHeads, numTails)
86          for (numFlips=0; numFlips<trialFlips; numFlips++) {
87              // rand() is not thread safe in linux
88              // rand_r() is available in linux and thread safe,
89              // to be run on separate threads concurrently.
90              // On windows in visual studio, use rand(), which is thread safe.
91              if (rand_r(&seed)%2 == 0) // if random number is even, call it heads
92                  numHeads++;
93              else
94                  numTails++;
```

```
95          }
96
97          /***  OMP ***/
98          ompStopTime = omp_get_wtime();  //get time this trial finished
99
100         // Finish this trial by printing out results
101
102         printf("%15d%15d%15d%15d%15.6f%15.6f\n", trialFlips, numHeads, numTails,
103                 (numHeads+numTails), chiSq(numHeads, numTails),
104                 (double)(ompStopTime-ompStartTime));    /***  OMP ***/
105
106         trialFlips *= 2;   // double the number of flips for the next trial
107     }
108
109     /***** Finish Up *****/
110     printf("\n\n\t<<< Normal Termination >>>\n\n");
111     return 0;
112 }
```

5.1 Some notes about this code

1. On line 15 we include the OpenMP library.
2. On lines 75 and 98 we use the OpenMP function to return a wall clock time in seconds. The difference between these provides the total amount of time to run the section of code enclosed by these lines. Note that this OpenMP function called *omp_get_wtime* specifically provides the overall time for the threaded code to run. We need to use this function because the original method using the clock() function does not work properly with threaded code.
3. Lines 82 - 85 indicate the setup for running the for loop of coin flips in equal numbers of iterations per thread. There are several directives needed to be added to the parallel for pragma:
   - *num_threads(nThreads)* designates how many threads to fork for this loop.
   - *default(none)* designates that all variables in the loop will be defined as either private within each thread or shared between the threads by the next three directives.
   - the \ designates that the pragma declaration is continuing onto another line
   - *private(numFlips, seed)* designates that each thread will keep its own private copy of the variables numFlips and seed and update them independently.
   - *shared(trialFlips)* designates that the variable trialFlips is shared by all of the threads (this is safe because no thread will ever update it.)
   - *reduction(+:numHeads, numTails)* is a special indicator for the the two values numHeads and numTails, which need to get updated by all the threads simultaneously. Since this will cause errors when the threads are executing, typically the OpenMP threaded code will have each thread keep a private copy of these variables while they execute their portion of the loop. Then when they join back after they have finished , each thread's private numHeads and numTails sum is added to an overall sum and stored in thread 0's copy of numHeads and numTails.
4. You can download the file `coinFlip_omp.cpp` and try this code yourself. If you have 4 cores available on your computer, you should see the longer trials with many coin flips run almost four times faster than our earlier sequential version that did not use threads.

Figure 5.1: “Sahara Hotel and Casino 2” by Antoine Taveneaux - Own work. Licensed under Creative Commons Attribution-Share Alike 3.0 via Wikimedia Commons

CHAPTER

SIX

ROULETTE SIMULATION

An American Roulette wheel has 38 slots: 18 are red, 18 are black, and 2 are green, which the house always wins. When a person bets on either red or black, the odds of winning are 18/38, or 47.37% of the time.

Our next is example is a simulation of spinning the Roulette wheel. We have a main simulation loop that is similar to the coin-flipping example. The code for determining a win on each spin is more involved than flipping a coin, and the sequential version, `rouletteSimulation_seq.cpp` is decomposed into several methods. Look at this original code file to see how we run the simulations using increasing numbers of random spins of the wheel.

The function that actually runs a single simulation of the Roulette wheel, called spinRed(), is quite simple. It generates a random number to represent the slot that the ball ends up in and gives a payout according to the rules of Roulette.

```
//spin the wheel, betting on RED
//Payout Rules:
//  0..17 you win (it was red)
// 18..35 you lose (it was black)
// 36..37 house wins (green) - you lose half
int spinRed(int bet, unsigned int *seed) {
        int payout;
        int slot = rand_rIntBetween(1,38, seed);
        /* if Windows
        int slot = randIntBetween(1,38);
         */
        if (slot <= 18) //simplify odds: [0..17]==RED
                payout = bet;         //won
        else if (slot <= 36) //spin was 'black'-lose all
                payout = -bet;         //lost
        else //spin was green - lose half
                payout = -(bet/2); //half-back
        return payout;
} // spinRed
```

Note: The sequential version of the simulation takes a fair amount of time. Note how long. Also note how many simulated random spins it takes before the distribution of spins accurately reflects the house odds.

6.1 Parallelism to the Rescue

We add OpenMP parallelism as in the coinFlip example, by running the loop of random spins for each trial on several threads. This code is in this file that you can download: `rouletteSimulation_omp.cpp` The actual simulation function is getNumWins():

```
/********************** getNumWins ************/
int getNumWins(int numSpins, unsigned int seed) {
//always bet 'red' & count wins
        static int wins;//our counter
        int spin;                //loop cntrl var
        int myBet = 10; //amount we bet per spin

        wins = 0;        //clear our counter

/***  OMP ***/
#pragma omp parallel for num_threads(nThreads) default(none) \
    shared(numSpins, myBet) \
        private(spin, seed) \
        reduction(+:wins)
        for (spin=0; spin<numSpins; spin++){
                //spinRed returns +/- number (win/lose)
                if (spinRed(myBet, &seed) > 0) //a winner!
                        wins++;
        }        ////  end forked parallel threads

        return wins;
}  //getNumWins
```

Notes about this code: numSpins and myBet are shared between threads while spin is the loop index and unique to each thread. When using rand_r() as the thread-safe random number generator in linux/unix, the seed should be private to each thread also. Like the previous example, we combine the partial results from each thread with *reduction(+:wins)*.

CHAPTER

SEVEN

DRAWING FOUR CARDS OF THE SAME SUIT

Now let's turn our attention to the card game of Poker. There are methods to calculate the probability of drawing the various types of hands (see the Wikipedia Poker Probability Page for explanation). For our next example, we will examine one such type of hand with the following question:

If you are dealt a random hand of 5 cards, what is the probability that four of the cards each have a different suit?

To answer this question, we simulate shuffling a deck of cards and drawing a hand of cards.

7.1 Code Files

For this code, we have separate versions for Windows, which uses rand(), and linux, which uses rand_r() as the random number generators.

Linux	
sequential version:	`drawFourSuits_seq.cpp`
OpenMP version:	`drawFourSuits_omp.cpp`

Windows	
sequential version:	`drawFourSuits_seq.cpp`
OpenMP version:	`drawFourSuits_omp.cpp`

7.2 Sequential code

We represent the deck of cards as an array of integers. Our function for simulating deck shuffling is not the most efficient, but it tries to capture how a traditional "fan" shuffle actually works. We also have helper functions for initializing a deck, drawing a hand, and checking if the hand has four cards of the same suit. Download the appropriate sequential code file for your environment and study it. Note all the places where random numbers are generated for two aspects of the problem: shuffling the deck and picking cards from the deck to form a hand.

Using these helper functions, it was straightforward to write testOneHand, which initializes a deck, shuffles it, draws a hand, and then checks if all four suits are represented.

```
/***********************************************
************************** testOneHand *********/
bool testOneHand(unsigned int *seed_ptr){
//Create a deck...sort it...pick 4 cards...test 4 suits
        int deck[MAX_CARDS];        //std deck
        int hand[CARDS_IN_HAND];        //card hand

        initDeck(deck, seed_ptr);        //create & shuffle a new deck

        drawHand(deck, hand, seed_ptr);        //go pick cards from deck

        return isFourSuits(hand); //test if 4 suits
}//testOneHand
```

7.3 Open MP Version

Converting our sequential code to use OpenMP is quite simple. We add a pragma compiler directive to the main simulation loop to run the loop simultaneously on multiple CPUs. The directive tells OpenMP to give each thread a different copy of i since each thread needs to keep track of its own loop iterations. numTests is shared because the total number of tests to run is doubled only once per iteration of the out while loop. (If each thread doubled it, we would go up by more than a factor of two.) Finally, the directive *reduction (+:total)* tells OpenMP to combine each of the threads' partial results by summing to find the total number of hands that contained all four suits.

```
/************************ 2.0 Simulation Loop **/
        while (numTests < MAX) {
                total = 0;        //reset counter

#pragma omp parallel for num_threads(nThreads) default(none) \
                private (i, seed) \
                shared (numTests) \
                reduction (+:total)        \
                schedule(dynamic)
                for (i=0; i<numTests; i++) { //make new deck - pick hand - test for 4 suits
                        if (testOneHand(&seed))                //returns TRUE iff 4-suits hand
                                total ++;                //tally hands with 4-suits
                }
                //calc % of 4-suit hands & report results...
                percentage = 100.0*( (double)total)/numTests;
                cout<<setw(12)<<numTests<<setw(14)<<setprecision(3 )<<fixed<<percentage<<endl;
                numTests += numTests;        //double #tests for next round
        } //while
```

Note that the above example is for the linux version of the code, which uses the thread-safe rand_r() function.

CHAPTER

EIGHT

PLINKO FROM THE PRICE IS RIGHT

Another interesting game of chance is called Plinko, which has been used on the television game show "The Price is Right" for many years. This game is quite similar to a game called Quincunx, invented by the mathematician Sir Frances Galton (also called the Galton board).

Please see the following references for details about the game and Galton's pioneering work:

Plinko game described

Detailed Description of the original board

Explanation of Quincunx

Animated Galton Board

8.1 An Exercise for You to Try

After reading these references and any others that you can find, please attempt to build your own simulation of the Plinko game.

A sketch of the board from *The Price is Right* looks like this:

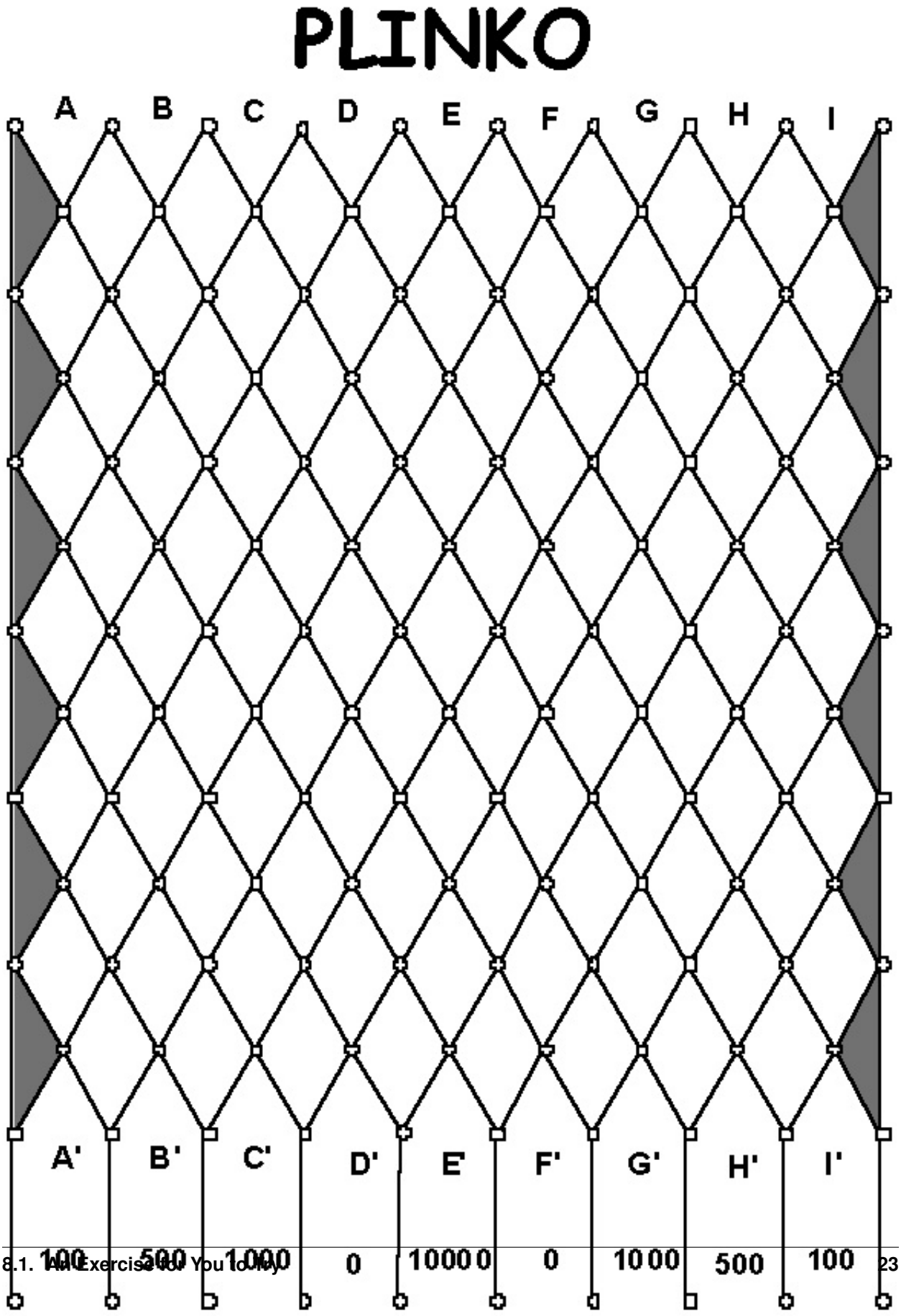

PLINKO
A
B
C
D
E
F
G
H
I
A'
B'
C'
D'
E'
F'
G'
H'
I'
100
500
1000
0
1000
0
0
1000
500
100

Image obtained from http://www.mathdemos.org/mathdemos/plinko/

Similar to previous examples, you should have a series of trial simulations, adding more and more disks to drop for each trial (on the Price is Right, contestants drop from one to five disks). You could start with five disks and double it for each trial. You will want to test what happens for each of the possible slots that they can initially drop the disk, labeled as A through I in the above diagram. The basic loop for simulating the trials would look like this:

```
while (numDisks < MAX) {
        //drop numDisks thru each starting pos [A..I]
        cout<<"\n\nDropping " << numDisks << " disks---\n";
        for(pos='A'; pos<='I'; pos++) {
                dropDisks(pos, numDisks, numBigMoney);
        }//for-pos

        //show totals for this run...
        showResults(numDisks, numBigMoney);

        //increase #disks for next run
        numDisks+=numDisks;
}//while
```

The function called *dropDisks* would do the large task of trying each disk to see where it lands. The loop might look something like this:

```
//The workhorse loop
for (disk=0; disk<numDisks; disk++) {
        double valu = dropOneDisk(pos); //how much did we win?
        totalWon+=valu;                 //keep running total for pos
        if (valu==BIG_MONEY)    //was it the BIG MONEY?
                numBigMoneyHits++;
}//for-disk

numBigMoney[index]=numBigMoneyHits;     //tally bigMoney hits for pos
```

As with the other examples presented earlier, using OpenMP to parallelize this loop should be a fairly straightforward task. The complex task for you is to code the logic of the possible movement of the disks along each row and compute how often the disk landed in each slot.

8.2 Questions you could answer

1. Where should I put my disks at the top to maximize the return at the bottom?
2. **For each starting slot, what is the probability that you will hit the big money slot?** Match your experimental results with the theoretical probabilities.

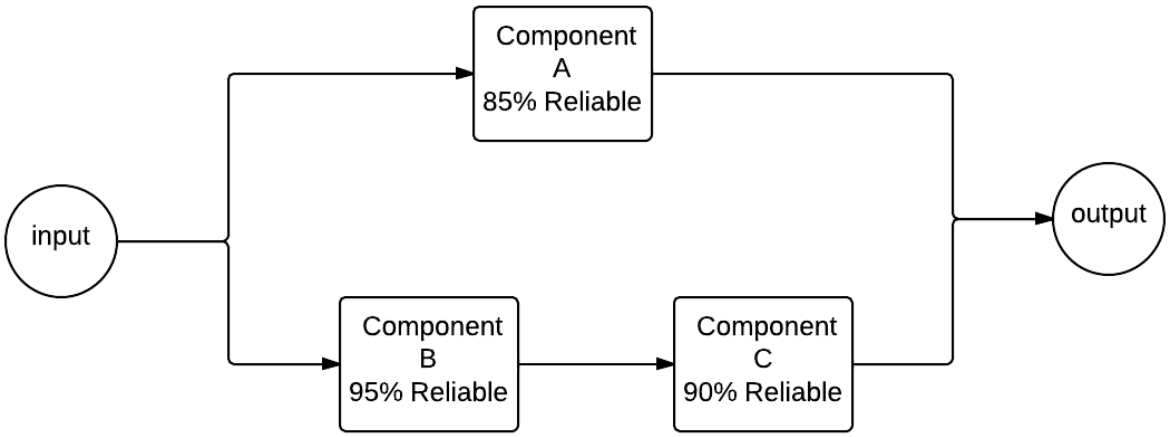
CHAPTER

NINE

EXERCISES

Here are example projects for you to try, now that you have run the previous examples.

1. Code the Monte Carlo method for approximating π mentioned in the Introduction. How many simulated dart throws do you need to run to approximate π to 3 digits of accuracy?
2. Let's examine a twist on the example of drawing all four suits out of 5 drawn cards. Write a program to shuffle a deck of cards and determine on average, how many cards you will need to draw:
   (a) before you have all four suits represented
   (b) before you get all four aces
   (c) before you get a *marriage* (a King and Queen of the same suit)
   (d) before you get all 13 clubs
   (e) before your get a *pinochle* (Queen of Spades and Jack of Diamonds)
3. Given a system made of discrete components with known reliability, what is the reliability of the overall system? For example, suppose we have a system that can be described with the following high-level diagram:



When given an input to the system, that input flows through component A or through components B and C, each of which has a certain reliability of correctness. Probability theory tells us the following:

$$reliability_{BC} = 0.95 * 0.90 = 0.855$$

$$reliability_{A} = 0.85$$

And the overall reliability of the system is:

$$\begin{aligned} reliability_{sys} &= 1.0 - [(1.0 - 0.85) * (1.0 - 0.855)] \\ &= 0.97825 \end{aligned}$$

Create a simulation of this system where half the time the input travels through component A. To simulate its reliability, generate a number between 0 and 1. If the number is 0.85 or below, component A succeeded, and the system works. The other half of the time, the input would travel on the lower half of the diagram. To simulate this, you will generate two numbers between 0 and 1. If the number for component B is less than 0.95 and the number for component C is less than 0.90, then the system also succeeds. Run many trials to see if you converge on the same reliability as predicted by probability theory.

CHAPTER

TEN

ADVANCED TOPIC: SEEDS FOR DIFFERENT THREADS

Adding OpenMP pragmas on the 'workhorse' for loops where most of the computation is being done is often a helpful way to make your code run faster. In the case of Monte Carlo simulations, there is one issue that should be addressed to ensure the best random distribution of numbers from the random number generator functions. We must start each thread with a different seed.

Recall that random number generators start from a 'seed' large integer and create a sequence of integers by permuting the seed and each successive integer in a manner that ensures they are distributed across the range of all integers. The key point is this: *the sequence of numbers from a random generator is always the same when it starts with the same seed.* In code where we fork threads to do the work of generating random numbers, we lose the desired random distribution if each thread begins generating random numbers from the same seed.

The solution to this issue for threaded code, which you can `download as coinFlip_omp_seeds.cpp`, is to ensure that each thread has its own seed from which it begins generating its sequence of integers. Let's revisit the coin flipping example. Instead of generating one seed in main using time(), we can save a seed for each thread in an array and devise a function to create all of the seeds, based on the number of threads to run. We can add this code at the beginning of our original file:

```
/***  OMP ***/
const int nThreads = 4;  // number of threads to use
unsigned int seeds[nThreads];

void seedThreads() {
    int my_thread_id;
    unsigned int seed;
    #pragma omp parallel private (seed, my_thread_id)
    {
        my_thread_id = omp_get_thread_num();

        //create seed on thread using current time
        unsigned int seed = (unsigned) time(NULL);

        //munge the seed using our thread number so that each thread has its
        //own unique seed, therefore ensuring it will generate a different set of numbers
        seeds[my_thread_id] = (seed & 0xFFFFFFF0) | (my_thread_id + 1);

        printf("Thread %d has seed %u\n", my_thread_id, seeds[my_thread_id]);
    }

}
/***  OMP ***/
```

Not how we change the seed value for each thread by using the thread's id to manipulate the original integer obtained from time().

Then later in the main function, we add a call to this function:

```
    /***  OMP ***/
    omp_set_num_threads(nThreads);
    seedThreads();
    /***  OMP ***/
```

For each trial, we still parallelize the workhorse for loop, while also ensuring that each thread running concurrently has its own seed as the starting point for later numbers.

```
#pragma omp parallel num_threads(nThreads) default(none) \
        private(numFlips, tid, seed) \
        shared(trialFlips, seeds) \
        reduction(+:numHeads, numTails)
    {
        tid = omp_get_thread_num();    // my thread id
        seed = seeds[tid];             // it is much faster to keep a private copy of our seed
                srand(seed);                        //seed rand_r or rand

        #pragma omp for
        for (numFlips=0; numFlips<trialFlips; numFlips++) {
//          in Windows, can use rand()
//            if (rand()%2 == 0) // if random number is even, call it heads
            // linux: rand_r() is thread safe, to be run on separate threads concurrently
            if (rand_r(&seed)%2 == 0) // if random number is even, call it heads
                numHeads++;
            else
                numTails++;
        }

    }
```

Study the above code carefully and compare it to our first version below. The *pragma omp* directive above is forking the new set of threads, which do a bit of work to set up their own seeds. Then the *pragma omp for* directive is indicating that those same threads should now split up the work of the for loop, just as in our previous example using the OpenMP pragma. The first OpenMP version we showed you looked like this:

```
#pragma omp parallel for num_threads(nThreads) default(none) \
        private(numFlips, seed) \
        shared(trialFlips) \
        reduction(+:numHeads, numTails)
        for (numFlips=0; numFlips<trialFlips; numFlips++) {
            // rand() is not thread safe in linux
            // rand_r() is available in linux and thread safe,
            // to be run on separate threads concurrently.
            // On windows in visual studio, use rand(), which is thread safe.
            if (rand_r(&seed)%2 == 0) // if random number is even, call it heads
                numHeads++;
            else
                numTails++;
        }
```

Note: A common 'gotcha' that can cause trouble is if you accidentally use the original *pragma omp parallel for* directive near the for loop in the new version. This causes incorrect unintended behavior. Remember to remove the **parallel** keyword in the inner block when nesting bloaks as shown in the new version where we set up seeds first before splitting the loop work.

Note that as before, in linux we need to use the rand_r() function for thread-safe generation of the numbers. However, in Windows, the rand() function is thread-safe.

10.1 Try this yourself

Try creating versions of the Roulette wheel simulation or drawing four suits that ensure that each thread is generating numbers from its own seed.