Chapter 9
Reduction Variables

► Part I. Preliminaries
▼ Part II. Tightly Coupled Multicore
  Chapter 6. Parallel Loops
  Chapter 7. Parallel Loop Schedules
  Chapter 8. Parallel Reduction
  Chapter 9. Reduction Variables
  Chapter 10. Load Balancing
  Chapter 11. Overlapping
  Chapter 12. Sequential Dependencies
  Chapter 13. Strong Scaling
  Chapter 14. Weak Scaling
  Chapter 15. Exhaustive Search
  Chapter 16. Heuristic Search
  Chapter 17. Parallel Work Queues
► Part III. Loosely Coupled Cluster
► Part IV. GPU Acceleration
► Part V. Map-Reduce
► Appendices
The $\pi$ estimating programs in Chapter 8 used class edu.rit.util.Random to generate random $(x, y)$ points. Each call of the `nextDouble()` method produced a new random number in the range 0.0 (inclusive) through 1.0 (exclusive). But this is paradoxical. How can a completely deterministic algorithm (in the `nextDouble()` method) produce random numbers?

The answer is that, for all practical purposes, the sequence of numbers produced by class Random is indistinguishable from the sequence of random numbers produced by a “true” random source, like flipping coins or rolling dice. To emphasize that the numbers did not come from a true random source, we call them pseudorandom numbers, and we call the algorithm that produced them a pseudorandom number generator (PRNG).

But how can we tell if the output of a PRNG is indistinguishable from the output of a true random number generator? The generally accepted criterion is to use statistical tests of randomness. If the PRNG passes many such tests, then for all practical purposes we can treat the PRNG’s output as being random. Several random number generator statistical test suites exist, such as Diehard* and TestU01.†

A statistical test of a sequence of allegedly random numbers goes like this: Formulate a null hypothesis, which is a statement about the probability distribution we expect the random numbers to obey. For example, the null hypothesis might be that the numbers are uniformly distributed between 0.0 and 1.0. Next, compute a statistic from the sequence of numbers. By convention, a large value of the statistic signals poor agreement between the numbers and the null hypothesis; a small statistic signals good agreement; and a statistic of 0.0 signals perfect agreement. Finally, compute the p-value of the statistic. The p-value is the probability that a value greater than or equal to the calculated statistic would be observed, even if the null hypothesis were true; that is, even if the numbers were in fact random. The smaller the p-value, the less likely that the null hypothesis is true; that is, the less likely that the numbers are random. If the p-value falls below a significance threshold, such as 0.05 or 0.01, then the null hypothesis is disproved (at that significance), and we conclude that the numbers are in fact not random.

One widely-used statistical test is the chi-square test. Suppose I am dealing with numbers $x$ in the range $0.0 \leq x < 1.0$, and suppose the null hypothesis is that the numbers are uniformly distributed; this is what I expect from class Random’s `nextDouble()` method. I divide the interval from 0.0 to 1.0 into some number of equally-sized bins; say, ten bins. Bin 0 corresponds to the subinterval $0.0 \leq x < 0.1$, bin 0 to subinterval $0.1 \leq x < 0.2$, … bin 9 to subinterval $0.9 \leq x < 1.0$. Each bin has an associated counter. Now I do a bunch of trials, say $N$ of them. For each trial, I generate a random number,

and I increment the proper bin’s counter depending on the subinterval in which the random number falls.

When I’m done with the trials, I compute the chi-square statistic, $\chi^2$, from the bin counters. If the null hypothesis is true, that the numbers are uniformly distributed, then ideally the count in each bin should be the same, namely $N$ divided by the number of bins. The formula for $\chi^2$ in this case is

$$\chi^2 = \sum_{0}^{B-1} \frac{(n_i - N/B)^2}{N/B}$$

(9.1)

where $B$ is the number of bins, $N$ is the number of trials, and $n_i$ is the count in bin $i$. Finally, I compute the $p$-value of $\chi^2$. (I’ll discuss how later.)

If the count in every bin is exactly equal to the expected value $N/B$, then $\chi^2$ is 0.0 and the $p$-value is 1.0. As the bin counts start to deviate from the expected value, either up or down, $\chi^2$ increases and the $p$-value decreases. If the bin counts get too far from the expected value, $\chi^2$ gets too large, the $p$-value gets too small and falls below the significance threshold, and the statistical test fails.

Here are examples of a six-bin chi-square test with 6000 trials on two different PRNGs. The bin counts and the $\chi^2$ statistics are

<table>
<thead>
<tr>
<th>Bin</th>
<th>PRNG 1 Count</th>
<th>PRNG 2 Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>976</td>
<td>955</td>
</tr>
<tr>
<td>1</td>
<td>971</td>
<td>950</td>
</tr>
<tr>
<td>2</td>
<td>989</td>
<td>968</td>
</tr>
<tr>
<td>3</td>
<td>989</td>
<td>968</td>
</tr>
<tr>
<td>4</td>
<td>1039</td>
<td>1018</td>
</tr>
<tr>
<td>5</td>
<td>1036</td>
<td>1141</td>
</tr>
</tbody>
</table>

$\chi^2 = 4.4760 \quad 26.778$

Here is a plot of the $p$-value versus $\chi^2$ for a six-bin chi-square test:

PRNG 1 passes the test at a significance of 0.01; for $\chi^2 = 4.4760$, the $p$-value is above the significance threshold. PRNG 2 fails the test; for $\chi^2 = 26.778$, the $p$-value is below the significance threshold.
I want to write computer programs that carry out the chi-square test on the PRNG in class edu.rit.util.Random, both a sequential version and a parallel version. The programs’ designs are similar to those of the \( \pi \) estimating programs in Chapter 8: generate a bunch of pseudorandom numbers, count things, compute the answer from the counts. But this time, instead of a single counter, I’m dealing with multiple counters, one for each bin. Following the principles of object oriented design, I’ll encapsulate the bin counters inside an object; this object is called a histogram.

The sequential statistical test program will generate \( N \) random numbers, accumulate them into a histogram, and compute \( \chi^2 \) (Figure 9.1). The parallel statistical test program will use the parallel reduction pattern, just like the parallel \( \pi \) estimating program. The parallel program will partition the \( N \) trials among the \( K \) parallel team threads. Each thread will do \( N/K \) trials and will accumulate its random numbers into its own per-thread histogram.

Before computing \( \chi^2 \), the per-thread histograms have to be reduced together into one overall histogram. Here is an example of a six-bin chi-square test with 6000 trials, where the trials were done in parallel by four threads (1500 trials in each thread). The per-thread histograms, and the overall histogram after the reduction, are

<table>
<thead>
<tr>
<th>Bin</th>
<th>Thread 0 Histogram</th>
<th>Thread 1 Histogram</th>
<th>Thread 2 Histogram</th>
<th>Thread 3 Histogram</th>
<th>Overall Histogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>244</td>
<td>249</td>
<td>227</td>
<td>256</td>
<td>976</td>
</tr>
<tr>
<td>1</td>
<td>239</td>
<td>241</td>
<td>248</td>
<td>243</td>
<td>971</td>
</tr>
<tr>
<td>2</td>
<td>248</td>
<td>259</td>
<td>239</td>
<td>243</td>
<td>989</td>
</tr>
<tr>
<td>3</td>
<td>261</td>
<td>241</td>
<td>260</td>
<td>227</td>
<td>989</td>
</tr>
<tr>
<td>4</td>
<td>245</td>
<td>260</td>
<td>259</td>
<td>275</td>
<td>1039</td>
</tr>
<tr>
<td>5</td>
<td>263</td>
<td>250</td>
<td>267</td>
<td>256</td>
<td>1036</td>
</tr>
</tbody>
</table>

\[ \chi^2 = 4.4760 \]

Each bin count in the overall histogram is the sum of the corresponding bin counts in the per-thread histograms. Thus, the reduction operation that combines two histograms together is to add each bin in one histogram to its counterpart in the other histogram.

I want the parallel loop classes in the Parallel Java 2 Library to do the histogram reduction automatically, just as they did with the LongVbl reduction variable in the parallel \( \pi \) estimating program. This means that the histogram class must be suitable for use as a reduction variable. Specifically, the histogram reduction variable class must implement interface edu.rit.pj2.Vbl and must provide implementations for the methods declared in that interface.

The Parallel Java 2 Library provides two classes, one for a histogram, the other for a histogram reduction variable. The classes are separate so that a program that needs a histogram, but not a histogram reduction variable, can just use the former class.
Chapter 9. Reduction Variables

Figure 9.1. Chi-square test done sequentially and in parallel

```
package edu.rit.numeric;
public class Histogram
    implements Cloneable
{
    private int B;        // Number of bins
    private long[] count; // Count in each bin
    private long total;   // Total count in all bins

    // Construct a new histogram with the given number of bins.
    public Histogram
    (int B)
    {
        if (B < 2)
            throw new IllegalArgumentException (String.format
                ("Histogram(): B = %d illegal", B));
        this.B = B;
        this.count = new long [B];
        this.total = 0;
    }

    // Make this histogram be a deep copy of the given histogram.
    public Histogram copy
    (Histogram hist)
    {
        this.B = hist.B;
        this.count = hist.count == null ? null :
            (long[]) hist.count.clone();
        this.total = hist.total;
        return this;
    }
```

Listing 9.1. Histogram.java (part 1)
Listing 9.1 is the code for the histogram, class edu.rit.numeric.Histogram. (To save space, I omitted some of the methods that are not used by the statistical test program.) The code is mostly self-explanatory. There are three hidden fields: the number of bins, an array of long bin counters, and the total of all the bin counters (lines 5–7). There is a constructor for a given number of bins (line 10). There is a method to copy one histogram into another (line 22). This method makes a deep copy by setting this histogram’s bin counter array to a new array whose contents are copied from the other histogram’s bin counter array (lines 26–27); afterwards, changes to one histogram will not affect the other. There are methods to clone a histogram (line 33), to get the number of histogram bins (line 48), to accumulate a double value into the proper bin (line 54), to do the actual bin update (line 61), to return a given bin counter (line 69), to compute $\chi^2$ (line 90), and to compute the $p$-value of $\chi^2$ (line 104) by calling a method in class edu.rit.numeric.Statistics. There is also a method to add each bin counter in one histogram to the corresponding bin counter in another histogram (line 111); this method will be used by the histogram reduction variable class.

Now that I have the Histogram class, I can write a program to do the actual chi-square test. Class edu.rit.pj2.example.StatTestSeq (Listing 9.2) is the sequential version. The command line arguments are the seed for the PRNG, the number of histogram bins $B$, and the number of trials $N$. The code is mostly straightforward. It sets up a PRNG object and a histogram object; performs $N$ trials in a loop, where each trial generates the next random number from the PRNG and accumulates the random number into the histogram; and finally prints the $\chi^2$ statistic and the $p$-value. The sequential version does not do a reduction and so does not use the histogram reduction variable class.

The only thing not quite straightforward in the StatTestSeq program is the way the histogram variable is initialized (lines 34–40). I designed class Histogram with flexibility in mind. One aspect of a histogram that could change from program to program is the manner in which data values are mapped to histogram bins. For data values of type double, the mapping must be specified by overriding the accumulate() method. The StatTestSeq program does this by assigning an instance of a subclass of class Histogram to the hist variable. The subclass is specified using Java’s anonymous inner class syntax, like the body of a parallel for loop. When the subclass’s accumulate() method is called, the argument is the output of the PRNG’s nextDouble() method, namely a value in the range 0.0 (inclusive) through 1.0 (exclusive). This is multiplied by the number of histogram bins (size()) and truncated to an integer, yielding a value in the range 0 (inclusive) through $B$ (exclusive), where $B$ is the number of bins; that is, an integer bin index in the range 0 through $B-1$. Calling the protected increment() method then increments the count in that bin.
// Create a clone of this histogram.
public Object clone()
{
    try {
    Histogram hist = (Histogram) super.clone();
    hist.copy (this);
    return hist;
    }
    catch (CloneNotSupportedException exc)
    { throw new RuntimeException ("Shouldn't happen", exc); }
}

// Returns the number of bins in this histogram.
public int size()
{
return B;
}

// Accumulate the given value of type double into this histogram.
public void accumulate
(double x)
{ throw new UnsupportedOperationExce

// Increment the given bin in this histogram.
protected void increment
(int i)
{ ++ count[i];
++ total;
}

// Returns the count in the given bin of this histogram.
public long count
(int i)
{ return count[i];
}

// Determine the expected count in the given bin for a chi-square
// test.
public double expectedCount
(int i)
{ return expectedProb(i)*total;
}

// Determine the expected probability of incrementing the given
// bin for a chi-square test.
public double expectedProb
(int i)
{ return 1.0/B;
}

Listing 9.1. Histogram.java (part 2)
Turning to the parallel statistical test program, I need a reduction variable class for histograms. This is class edu.rit.pj2.vbl.HistogramVbl in the Parallel Java 2 Library (Listing 9.3). (Again, I have omitted some of the methods to save space.) The class implements interface Vbl (line 5). The class “wraps” an instance of class Histogram, which is specified as the constructor argument (line 12) and is stored in the public hist field (line 8). The remaining methods are the ones declared in interface Vbl, that make the HistogramVbl class suitable for use as a reduction variable. These methods must be implemented as described below, otherwise parallel reduction will not work.

- The reduction variable’s clone() method (line 19) must create a new reduction variable object that is a deep copy of this reduction variable object (the one being cloned). Class HistogramVbl’s clone() method does so by invoking the superclass’s clone() method (line 21). This creates a shallow copy of the reduction variable. To make the new reduction variable be a deep copy of the original reduction variable, the histogram in the hist field is itself cloned, creating a deep copy of the original histogram (see Listing 9.1 lines 32–44); the deep copy then replaces the original histogram (lines 22–23). This follows the standard pattern for cloning an object in Java.
- The reduction variable’s set() method (line 28) must change this reduction variable object to be a deep copy of the given reduction variable object. Class HistogramVbl’s set() method does so by invoking the histogram’s copy() method (see Listing 9.1 lines 22–30) to copy the given reduction variable’s histogram into this reduction variable’s histogram.
- The reduction variable’s reduce() method (line 35) performs the reduction operation. It must apply the reduction operation to this reduction variable object and the given reduction variable object, and store the result back into this reduction variable object. Class HistogramVbl’s reduce() method accomplishes this by calling the add() method on this reduction variable’s histogram, passing in the other reduction variable’s histogram. The add() method then does the work (see Listing 9.1 lines 111–121).

Class HistogramVbl also extends class edu.rit.pj2.Tuple (line 5). A tuple is an object used to communicate information between processes in a cluster parallel program. We will encounter tuples in Chapter 20. I decided to make class HistogramVbl extend class Tuple so that (in some future program) I could send histogram reduction variable objects between processes. However, if a reduction variable class is not going to be used for interprocess communication, the class does not need to extend class Tuple; it only needs to implement interface Vbl.

I want the parallel version of the chi-square program to compute exactly the same histogram as the sequential version, no matter how many cores
public double chisqr() {
    double chisqr = 0.0;
    for (int i = 0; i < B; ++i) {
        double expected = expectedCount (i);
        double d = expected - count[i];
        chisqr += d*d/expected;
    }
    return chisqr;
}

public double pvalue (double chisqr) {
    return Statistics.chiSquarePvalue (B - 1, chisqr);
}

public void add (Histogram hist) {
    if (hist.B != this.B)
        throw new IllegalArgumentException
            ("Histogram.add(): Histograms are different sizes");
    for (int i = 0; i < B; ++i)
        this.count[i] += hist.count[i];
    this.total += hist.total;
}

Listing 9.1. Histogram.java (part 3)

package edu.rit.pj2.example;
import edu.rit.numeric.Histogram;
import edu.rit.pj2.Task;
import edu.rit.util.Random;
public class StatTestSeq
    extends Task {
    // Command line arguments.
    long seed;
    int B;
    long N;
    // Pseudorandom number generator.
    Random prng;
    // Histogram of random numbers.
    Histogram hist;

Listing 9.2. StatTestSeq.java (part 1)
(threads) the program runs on. Strictly speaking, the program doesn’t need to do this, but I want to illustrate how to code it.

As in the parallel \( \pi \) estimating program, each thread in the parallel chi-square program will have its own per-thread PRNG. Unlike the parallel \( \pi \) estimating program which initialized each per-thread PRNG with a different seed, in the parallel chi-square program I will initialize each per-thread PRNG with the same seed. Doing so would normally cause each per-thread PRNG to generate the same sequence of random numbers. But this time, after initializing the PRNG, parallel team thread 1 will skip over one random number, namely the random number generated by thread 0; thread 2 will skip over two random numbers, namely those generated by threads 0 and 1; thread 3 will skip over three random numbers, namely those generated by threads 0, 1, and 2; and so on. Later, after generating each random number, each thread will skip over \( K - 1 \) random numbers, namely those generated by the other \( K - 1 \) threads in the parallel team of \( K \) threads. In this way, the threads in the parallel program will generate the same random numbers as the single thread in the sequential program, except that the random numbers will be generated in a round robin fashion among all the threads.

Figure 9.2 shows how the parallel program generates the same sequence of random numbers, no matter how many threads are running. The lines stand for the per-thread PRNGs, the white circles stand for the initial seed values (the same in all the threads), and the black circles stand for the random numbers generated by each thread.
// Main program.
public void main
(String[] args)
throws Exception
{
    // Validate command line arguments.
    if (args.length != 3) usage();
    seed = Long.parseLong (args[0]);
    B = Integer.parseInt (args[1]);
    N = Long.parseLong (args[2]);

    // Set up PRNG.
    prng = new Random (seed);

    // Set up histogram.
    hist = new Histogram (B)
    {
        public void accumulate (double x)
        {
            increment ((int)(x*size()));
        }
    }

    // Do N trials.
    for (long i = 0; i < N; ++ i)
        hist.accumulate (prng.nextDouble());

    // Print results.
    System.out.printf("Bin\tCount\n");
    for (int i = 0; i < B; ++ i)
        System.out.printf("%d\t%d\n", i, hist.count (i));
    double chisqr = hist.chisqr();
    System.out.printf("Chisqr = %.5g\n", chisqr);
    System.out.printf("Pvalue = %.5g\n", hist.pvalue (chisqr));
}

// Print a usage message and exit.
private static void usage()
{
    System.err.println("Usage: java pj2 " +
        "edu.rit.pj2.example.StatTestSeq <seed> <B> <N>\n");
    System.err.println("<seed> = Random seed\n");
    System.err.println("<B> = Number of histogram bins\n");
    System.err.println("<N> = Number of trials\n");
    terminate (1);
}

// Specify that this task requires one core.
protected static int coresRequired()
{
    return 1;
}

Listing 9.2. StatTestSeq.java (part 2)
Class edu.rit.pj2.example.StatTestSmp (Listing 9.4) is the multicore parallel chi-square test program. It starts by declaring the `histvbl` variable (line 16); this is the global reduction variable. In the `main()` method, the loop over the trials has become a parallel loop (line 39). Inside each thread's parallel loop object are the per-thread `thrHistvbl` variable, the per-thread PRNG, and a variable `leap` (lines 41–43). The parallel loop's `start()` method links the per-thread histogram reduction variable to the global histogram reduction variable, so that the automatic parallel reduction will take place (line 46). The `start()` method initializes the per-thread PRNG with the same seed in each thread (line 47), then skips over some random numbers; the quantity skipped is the thread's rank in the parallel team (line 48). Thus, thread 0 skips over none; thread 1 skips over one; thread 2 skips over two; and so on. The `start()` method initializes `leap` to the number of team threads, minus one (line 49).

On each loop iteration, the parallel loop's `run()` method generates a random number from the per-thread PRNG, accumulates it into the per-thread histogram, and skips over `leap` (that is, \( K - 1 \)) random numbers. To skip the PRNG ahead, I call the PRNG's `skip()` method (line 54). The `skip()` method is very fast; it advances the PRNG without bothering to generate the intervening random numbers. (The standard Java PRNG class has no such capability, which is one reason why I prefer to use my own class.)

Once the parallel loop iterations have finished, the per-thread histograms are automatically reduced into the global histogram under the hood. The reductions are performed using class `HistogramVbl`'s `reduce()` method. The program then uses the global histogram to compute and print the \( \chi^2 \) statistic and the \( p \)-value.

Here are runs of the sequential program and the parallel program on a 12-core tardis node with \( B = \) ten bins and \( N = 12 \) billion trials:

```bash
$ java pj2 debug=makespan edu.rit.pj2.example.StatTestSeq
   142857 10 12000000000

Bin    Count
0   1199992974
1   1199975267
2   1200003509
3   1200028728
4   1199950485
5   1200008745
6   1200033714
7   1199983654
8   1199998844
9   1200024080

Chisqr = 5.0099
Pvalue = 0.83344
Job 1 makespan 115063 msec
```

$ java pj2 debug=makespan edu.rit.pj2.example.StatTestSeq 
   142857 10 12000000000

Bin    Count
0   1199992974
1   1199975267
2   1200003509
3   1200028728
4   1199950485
5   1200008745
6   1200033714
7   1199983654
8   1199998844
9   1200024080

Chisqr = 5.0099
Pvalue = 0.83344
Job 1 makespan 115063 msec
package edu.rit.pj2.vbl;
import edu.rit.pj2.Vbl;
import edu.rit.numeric.Histogram;
public class HistogramVbl
extends Tuple implements Vbl
{
    // The histogram itself.
    public Histogram hist;

    // Construct a new histogram reduction variable wrapping the
    // given histogram.
    public HistogramVbl
    (Histogram hist)
    {
        this.hist = hist;
    }

    // Create a clone of this shared variable.
    public Object clone()
    {
        HistogramVbl vbl = (HistogramVbl) super.clone();
        if (this.hist != null)
            vbl.hist = (Histogram) this.hist.clone();
        return vbl;
    }

    // Set this shared variable to the given shared variable.
    public void set
    (Vbl vbl)
    {
        this.hist.copy (((HistogramVbl)vbl).hist);
    }

    // Reduce the given shared variable into this shared variable.
    public void reduce
    (Vbl vbl)
    {
        this.hist.add (((HistogramVbl)vbl).hist);
    }
}

Listing 9.3. HistogramVbl.java

package edu.rit.pj2.example;
import edu.rit.numeric.Histogram;
import edu.rit.pj2.LongLoop;
import edu.rit.pj2.Task;
import edu.rit.pj2.vbl.HistogramVbl;
import edu.rit.util.Random;
public class StatTestSmp
extends Task
{
    // Command line arguments.
    long seed;
    int B;
    long N;
}

Listing 9.4. StatTestSmp.java (part 1)
$ java pj2 debug=makespan edu.rit.pj2.example.StatTestSmp \
 142857 10 12000000000

<table>
<thead>
<tr>
<th>Bin</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1199992974</td>
</tr>
<tr>
<td>1</td>
<td>1199975267</td>
</tr>
<tr>
<td>2</td>
<td>1200003509</td>
</tr>
<tr>
<td>3</td>
<td>1200028728</td>
</tr>
<tr>
<td>4</td>
<td>1199950485</td>
</tr>
<tr>
<td>5</td>
<td>1200008745</td>
</tr>
<tr>
<td>6</td>
<td>1200033714</td>
</tr>
<tr>
<td>7</td>
<td>1199983654</td>
</tr>
<tr>
<td>8</td>
<td>1199998844</td>
</tr>
<tr>
<td>9</td>
<td>1200024080</td>
</tr>
</tbody>
</table>

Chisqr = 5.0099
Pvalue = 0.83344
Job 1 makespan 12492 msec

The speedup was $115063 \div 12492 = 9.211$.

Note that both programs did indeed compute exactly the same histogram. The $p$-value was 0.83344, so the PRNG passed the chi-square test at a significance of 0.01. (If it had failed the test, I’d be worried!)

**Under the Hood**

When dealing with a reduction variable, the Parallel Java 2 middleware utilizes the reduction variable class’s methods that are declared in interface edu.rit.pj2.Vbl. Here’s how the middleware uses each method:

- When a parallel team thread calls a parallel loop’s `start()` method, and the `start()` method calls the `threadLocal()` method passing in a global reduction variable, the `threadLocal()` method calls the global reduction variable’s `clone()` method to create a new reduction variable that is a deep copy of the global reduction variable. A reference to this new reduction variable is returned, and this new reduction variable becomes the per-thread variable.

- When a parallel loop finishes, the middleware performs a reduction tree (Figure 8.3) for each reduction variable that was specified earlier in a `threadLocal()` method call. All of the team threads’ per-thread reduction variables feed into the top of the reduction tree. As execution proceeds down the reduction tree, the intermediate results are stored back into the team threads’ per-thread variables. When the reduction tree has finished, team thread 0’s per-thread variable ends up containing the result of the reduction. This result now has to be stored in the global reduction variable for use outside the parallel loop. The middleware does this by calling the global reduction variable’s `set()` method, passing in team thread 0’s per-thread variable. This sets the global reduction variable to be a deep copy of the final reduced result.
Chapter 9. Reduction Variables

Listing 9.4. StatTestSmp.java (part 2)
• While executing the parallel reduction tree, at multiple points the middleware has to combine two intermediate results together by performing the reduction operation. The middleware does this by calling the \texttt{reduce()} method. This combines two per-thread variables together in the desired manner and stores the result back in one of the per-thread variables. The middleware itself handles all the necessary thread synchronization, so \texttt{reduce()} does not need to be a synchronized method.

As stated previously, when defining your own reduction variable class, it’s crucial to implement the \texttt{clone()}, \texttt{set()}, and \texttt{reduce()} methods exactly as specified in interface \texttt{edu.rit.pj2.Vbl}. If you don’t, the Parallel Java 2 middleware will not do the right thing when it performs the reduction, and the program will compute the wrong answer.

Here’s how the \textit{p}-value of the $\chi^2$ statistic is calculated. In a chi-square test, if the null hypothesis is true, and if the count in each histogram bin is large, then the $\chi^2$ statistic is a random variable that obeys the \textit{chi-square distribution} with $B-1$ \textit{degrees of freedom}, where $B$ is the number of bins. The \textit{p}-value is given by the formula

$$p\text{-value} = 1 - P\left(\frac{B-1}{2}, \frac{X^2}{2}\right)$$ (9-2)

where $P(\cdot, \cdot)$ is the “incomplete gamma function.” The incomplete gamma function is included in most numerical software libraries as well as the Parallel Java 2 Library. For further information about the chi-square test, refer to a statistics textbook or to a numerical software textbook like \textit{Numerical Recipes}.*

The Parallel Java 2 Library’s class \texttt{edu.rit.numeric.Histogram} lets you compute histograms for data values of type \texttt{int}, \texttt{long}, \texttt{float}, and \texttt{double}, as well as object types. As illustrated earlier, you have to override the \texttt{accumulate()} method to map a data value to the proper histogram bin and increment the corresponding bin counter.

Class \texttt{Histogram} also lets you customize the formula for the expected probability of each bin; that is, the probability that a given bin will be incremented, given a randomly chosen data value. The default expected probability distribution is a uniform distribution, where each bin has an equal chance $1/B$ of being incremented; but you can override this if the expected probability distribution is nonuniform.

Class \texttt{edu.rit.pj2.vbl.HistogramVbl} wraps an instance of class \texttt{Histogram} or a subclass thereof, to provide a reduction variable for any kind of histogram.

Class edu.rit.util.Random generates a sequence of pseudorandom numbers, like any PRNG class. Unlike most PRNG classes, class Random can efficiently skip ahead in the sequence without actually generating the intervening pseudorandom numbers. How? Class Random maintains a hidden 64-bit counter. The seed supplied as the constructor argument is used to initialize the counter. To generate the next pseudorandom number in the sequence, class Random increments the counter, feeds the new counter value through a hash function, and returns the hash function’s result. The hash function is designed so that its sequence of outputs looks random—that is, its sequence of outputs passes the Diehard and TestU01 statistical test suites—when its inputs are a sequence of consecutive counter values. (Class Random does not use a cryptographic hash function like SHA-256; its hash function is much simpler.) To skip ahead $n$ positions in the pseudorandom sequence, one merely increases the counter by $n$. As the statistical test program in this chapter illustrates, this skipping capability is often useful in parallel programs that work with random numbers.

### Points to Remember

- Make your own class a reduction variable class by implementing interface edu.rit.pj2.Vbl.
- Define the clone() method to create a new reduction variable that is a deep copy of this reduction variable.
- Define the set() method to make this reduction variable be a deep copy of the given reduction variable.
- Define the reduce() method to combine this reduction variable with the given reduction variable using the desired reduction operation and store the result back in this reduction variable.
- To generate the same sequence of random numbers in a parallel program no matter how many threads are running, initialize all the threads’ per-thread PRNGs with the same seed, and use the skip() method to skip past the other threads’ random numbers.
- The chi-square test is often used to test whether a sequence of numbers obeys a certain probability distribution.
• In your programs, consider using the Parallel Java 2 Library’s histogram class and histogram reduction variable class.