Chapter 5
Reduction Variables

► Part I. Preliminaries
▼ Part II. Tightly Coupled Multicore
  Chapter 2. Parallel Loops
  Chapter 3. Parallel Loop Schedules
  Chapter 4. Parallel Reduction
  **Chapter 5. Reduction Variables**
  Chapter 6. Load Balancing
  Chapter 7. Overlapping
  Chapter 8. Sequential Dependencies
  Chapter 9. Strong Scaling
  Chapter 10. Weak Scaling
  Chapter 11. Exhaustive Search
  Chapter 12. Heuristic Search
► Part III. Loosely Coupled Cluster
► Part IV. GPU Acceleration
► Part V. Map-Reduce
The \( \pi \) estimating programs in Chapter 4 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 algo-
rium (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 crite-

rion 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 conven-
tion, a large value of the statistic signals poor agreement between the num-
bers 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 thresh-
old, such as 0.05 or 0.01, then the null hypothesis is disproved (at that signifi-
cance), and we conclude that the numbers are in fact not random.

One widely-used statistical test is the chi-square test. Suppose I am deal-
ing with numbers \( x \) in the range \( 0.0 \leq x < 1.0 \), and suppose the null hypothe-
sis 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,

* [http://www.stat.fsu.edu/pub/diehard/](http://www.stat.fsu.edu/pub/diehard/)

† P. L’Ecuyer and R. Simard. TestU01: a C library for empirical testing of random
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

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

(5.1)

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 | 26.778 |

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

[Plot of $p$-value versus $\chi^2$ showing PRNG 1 passing 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 π estimating programs in Chapter 4: 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 5.1). The parallel statistical test program will use the parallel reduction pattern, just like the parallel π 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>

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 π estimating program. This means that the histogram class must be suitable for use as a reduction variable. Specifically, the histogram class must implement interface edu.rit.pj2.Vbl and must provide implementations for the methods declared in that interface.

Listing 5.1 is the code for the histogram, class edu.rit.pj2.example.Histogram, which implements interface Vbl (line 5). The code is mostly self-explanatory. There are three hidden fields: the number of bins, an array of long
Chapter 5.  Reduction Variables

Figure 5.1. Chi-square test done sequentially and in parallel

Listing 5.1. Histogram.java (part 1)

```java
package edu.rit.pj2.example;
import edu.rit.numeric.Statistics;
import edu.rit.pj2.Vbl;
public class Histogram
implements Vbl
{
    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;
}

    // Construct a new histogram that is a deep copy of the given
    // histogram.
    public Histogram
(Histogram hist)
{
    copy (hist);
}
```
bin counters, and the total of all the bin counters (lines 7–9). There are two constructors: a constructor for a given number of bins (line 12) and a copy constructor (line 25). There is a method to copy one histogram into another (line 31). 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 (line 35); afterwards, changes to one histogram will not affect the other. There are methods to accumulate a random number into the proper bin (line 40), to return a given bin counter (line 48), to return the total of all the bin counters (line 55), to compute $\chi^2$ (line 63), and to compute the $p$-value of $\chi^2$ (line 77). The latter is computed by calling a method in class edu.rit.numeric.Statistics. (For an explanation of how the $p$-value is computed, see the “Under the Hood” section below.)

The code so far is a fairly standard class for doing a chi-square test of a series of uniformly distributed random numbers. The remaining methods are the ones declared in interface Vbl, that make the Histogram class suitable for use as a reduction variable. These methods must be implemented as described below, otherwise parallel reduction will not work.

- The clone() method (line 84) must create a new reduction variable object that is a deep copy of this reduction variable object (the one being cloned). Class Histogram’s clone() method does so by invoking the copy constructor, creating a new histogram that is a deep copy of this histogram.
- The set() method (line 89) must change this reduction variable object to be a deep copy of the given reduction variable object. Class Histogram’s set() method does so by invoking the copy() method to copy the given histogram into this histogram.
- The reduce() method (line 96) 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 Histogram’s reduce() method goes through the two histograms’ bin counter arrays, adds the corresponding counters together, and stores the sums back into this array; the method does the same with the total counters.

One detail: To compute the expected bin count on line 65, I cast the total count (total) and the number of bins (B) to type double, and then I did the division. Why convert the variables to type double? Because the variables are type long, and division of long values discards the fractional part of the result. If I divided the variables without casting them, I’d lose the fractional part of the expected bin count, and I’d compute the wrong value of $\chi^2$. Division of double values, however, retains the fractional part of the result. Converting the variables to type double ensures that I compute the correct expected bin count, and thus the correct value of $\chi^2$. 
public void copy (Histogram hist) {
    this.B = hist.B;
    this.count = (long[]) hist.count.clone();
    this.total = hist.total;
}

// Accumulate the given random number into this histogram.
public void accumulate (double x) {
    ++ count[(int)(x*B)];
    ++ total;
}

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

// Returns the total count in all bins of this histogram.
public long total() {
    return total;
}

// Returns the chi-square statistic for this histogram. The null hypothesis is that the accumulated random numbers are uniformly distributed between 0.0 and 1.0.
public double chisqr() {
    double expected = (double)total/(double)B;
    double chisqr = 0.0;
    for (int i = 0; i < B; ++ i) {
        double d = expected - count[i];
        chisqr += d*d;
    }
    return chisqr/expected;
}

// Returns the p-value of the given chi-square statistic for this histogram.
public double pvalue (double chisqr) {
    return Statistics.chiSquarePvalue (B - 1, chisqr);
}

// Create a clone of this shared variable.
public Object clone() {
    return new Histogram (this);
}

Listing 5.1. Histogram.java (part 2)
Now that I have the Histogram reduction variable class, I can write programs to do the actual chi-square test. Class edu.rit.pj2.example.StatTestSeq (Listing 5.2) 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 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 make use of the histogram’s reduction variable methods.

I want the parallel version of the program to compute exactly the same histogram as the sequential version, no matter how many cores (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 5.2 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.

Class edu.rit.pj2.example.StatTestSmp (Listing 5.3) is the multicore parallel chi-square test program. It starts by declaring the histogram variable (line 14); this is the global reduction variable. In the main() method, the loop over the trials has become a parallel loop (line 31). Inside each thread’s parallel loop object are the per-thread thrHistogram variable, the per-thread PRNG, and a variable leap (lines 33–35). The parallel loop’s start() method links the per-thread histogram to the global histogram, so that the automatic parallel reduction will take place (line 38). The start() method ini-
Chapter 5. Reduction Variables

Listing 5.1. Histogram.java (part 3)

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

// Reduce the given shared variable into this shared variable.
public void reduce
  (Vbl vbl)
{
  Histogram hist = (Histogram)vbl;
  if (hist.B != this.B)
    throw new IllegalArgumentException();
  for (int i = 0; i < B; ++ i)
    this.count[i] += hist.count[i];
  this.total += hist.total;
}
```

Listing 5.2. StatTestSeq.java (part 1)

```java
package edu.rit.pj2.example;
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;

  // 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);
  }
```
itializes the per-thread PRNG with the same seed in each thread (line 39), then skips over some random numbers; the quantity skipped is the thread’s rank in the parallel team (line 40). Thus, thread 0 skips over none; thread 1 skips over one; thread 2 skips over two; and so on. The \texttt{start()} method initializes \texttt{leap} to $K - 1$, where $K$ is the number of team threads (line 41).

On each loop iteration, the parallel loop’s \texttt{run()} method generates a random number from the per-thread PRNG, accumulates it into the per-thread histogram, and skips over \texttt{leap} (that is, $K - 1$) random numbers. To skip the PRNG ahead, I call the PRNG’s \texttt{skip()} method (line 46). The \texttt{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 \texttt{Histogram}’s \texttt{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 four-core tardis node with $B =$ ten bins and $N =$ two billion trials:

```
$ java pj2 debug=makespan edu.rit.pj2.example.StatTestSeq \
  142857 10 2000000000

Bin  Count
 0  200007677
 1  199992323
Figure 5.2. Generating random numbers in a round robin fashion
Chapter 5. Reduction Variables

Listing 5.2. StatTestSeq.java (part 2)

```java
package edu.rit.pj2.example;
import edu.rit.pj2.LongLoop;
import edu.rit.pj2.Task;
import edu.rit.util.Random;
public class StatTestSmp
extends Task
{
// Command line arguments.
long seed;
int B;
long N;

// Global histogram of random numbers.
Histogram hist;

// Main program.
public void main
(String[] args)
throws Exception
{
// Validate command line arguments.
if (args.length != 3) usage();
seed = Long.parseLong(args[0]);
}
}
```

Listing 5.3. StatTestSmp.java (part 1)
2 199981249
3 199998299
4 199998146
5 200010690
6 200019197
7 200003302
8 199983897
9 200001399

Chisqr = 5.9335
Pvalue = 0.74655

Job 160 makespan 49913 msec

$ java pj2 debug=makespan edu.rit.pj2.example.StatTestSmp \\
   142857 10 2000000000

Bin  Count
0  200007677
1  199996144
2  199981249
3  199998299
4  199998146
5  200010690
6  200019197
7  200003302
8  199983897
9  200001399

Chisqr = 5.9335
Pvalue = 0.74655

Job 165 makespan 13596 msec

The speedup was 49913 ÷ 13596 = 3.671.

Note that both programs did indeed compute exactly the same histogram. The $p$-value was 0.74655, 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 4.3) for each reduction variable that was specified earlier in a `threadLocal()` method call. All of the team threads’ per-thread variables
B = Integer.parseInt(args[1]);
N = Long.parseLong(args[2]);

// Set up global histogram.
hist = new Histogram(B);

// Do N trials.
parallelFor(0, N-1).exec(new LongLoop()
{
    Histogram thrHist;
    Random prng;
    int leap;
    public void start()
    {
        thrHist = threadLocal(hist);
        prng = new Random(seed);
        prng.skip(rank());
        leap = threads() - 1;
    }
    public void run(long i)
    {
        thrHist.accumulate(prng.nextDouble());
        prng.skip(leap);
    }
});

// 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.StatTestSmp <seed> <B> <N> ");
    System.err.println("<seed> = Random seed");
    System.err.println("<B> = Number of histogram bins");
    System.err.println("<N> = Number of trials");
    throw new IllegalArgumentException();
}

Listing 5.3. StatTestSmp.java (part 2)
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 reduc-
tion. 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.

• While executing the parallel reduction tree, at multiple points the middle-
ware has to combine two intermediate results together by performing the
reduction operation. The middleware does this by calling the 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
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 clone(), set(), and reduce() methods exactly as
specified in interface edu.rit.pj2.Vbl. If you don’t, the Parallel Java 2 middle-
ware will not do the right thing when it performs the reduction, and the pro-
gram will compute the wrong answer.

Here’s how the p-value of the χ² 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 χ² statistic is a random variable that obeys the chi-square dis-
tribution with B − 1 degrees of freedom, where B is the number of bins. The
p-value is given by the formula

\[ p\text{-value} = 1 - P\left(\frac{B - 1}{2}, \frac{\chi^2}{2}\right) \]  \hspace{1cm} (5-2)

where \( P() \) is the “incomplete gamma function.” This 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 Numerical Recipes.*

The Parallel Java 2 Library provides reduction variable classes for histo-
grams: class edu.rit.pj2.vbl.HistogramVbl and its subclasses. The Library’s
classes let you compute histograms for values of type int, long, float, and
double, as well as object types. (The Histogram class in this chapter is actu-
ally a simplified version of class edu.rit.pj2.vbl.DoubleHistogramVbl.) The
Library’s classes let you customize the formula that determines which bin to
increment when a value is accumulated into the histogram. The Library’s

classes also let you customize the formula for the expected count in each bin for the chi-square test. Consider using the Library’s classes when you need histograms in your programs.

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 the given 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 reduction variable classes in package edu.rit.pj2.vbl.