Chapter 38
Map-Reduce Meets GIS

► Part I. Preliminaries
► Part II. Tightly Coupled Multicore
► Part III. Loosely Coupled Cluster
► Part IV. GPU Acceleration
▼ Part V. Big Data
   Chapter 35. Basic Map-Reduce
   Chapter 36. Cluster Map-Reduce
   Chapter 37. Big Data Analysis
   Chapter 38. Map-Reduce Meets GIS
► Appendices
The City of New Orleans has published a data set on the web containing information about each 911 emergency call made there during the years 2011 through 2018.* The data set comprises eight files, one for each year, totaling (at the time I’m writing) about 840 megabytes. The data set has over 3.4 million records, each line of each file being one record. Here is one of the records, in comma separated value (CSV) format:

```
A0002211,94,DISCHARGING FIREARMS,2B,3676541.00000000,527551.00000000,01/01/2011 12:02:55 AM,01/01/2011 12:03:01 AM,01/01/2011 12:07:17 AM,01/01/2011 01:07:31 AM,RTF,REPORT TO FOLLOW,025XX Thalia St,70113,6,"(29.944731889680263, -90.08386573166727)"
```

The first field is a unique identifier. The second and third fields are a code for and a description of the type of emergency call. There is information about when the call took place, the disposition of the call, and the street address of the event. The final field contains the geographic coordinates of the event in the format “(latitude, longitude)” measured in degrees. “(0.0, 0.0)” means the location is not known.

I want to analyze this data set to discover how prevalent a specified 911 call type was in different areas of the city. Rather than an unintelligible list of numbers, I want to see an insightful graphic, like Figure 38.1.

In the graphic, the New Orleans region is divided into a grid of square blocks 0.01 degree (about one kilometer) on a side. The grid lines indicate latitude (vertical axis) and longitude (horizontal axis). The color of each block indicates the number of 911 calls of, in this case, type 94 (discharging firearm) that occurred within that block. Colors green through red correspond to counts of 1 through 601, the maximum in any block, as shown in the legend at the right. White indicates that no calls of type 94 occurred in a block.

In this chapter I’ll develop a parallel map-reduce program whose input is the 911 call data set and whose output is the graphic. This is an example of a geographic information system (GIS). However, this program will not be a full-featured GIS application. Rather, my purpose is to demonstrate how to use the Parallel Java 2 Library to write a map-reduce job with graphical output. To keep things simple, the program will accept input data only in the format shown above, and all aspects of the graphical output will be hard coded.

The program’s command line arguments are

```
$ java pj2 edu.rit.pjmr.example.Nola911 nodes directory \ callType "title" plotfile
```

- **nodes** is a comma-separated list of the cluster backend node names on which to run mapper tasks.
- **directory** is the name of the directory containing the data set. All files in the directory will be analyzed.
- **callType** is the call type code to be analyzed.

Chapter 38. Map-Reduce Meets GIS

Figure 38.1. New Orleans 911 call graphic, type 94

```java
package edu.rit.pjmr.example;
import edu.rit.io.InStream;
import edu.rit.io.OutStream;
import edu.rit.io.Streamable;
import java.io.IOException;
public class Nola911Block
    implements Streamable
{
    // Scale factor. Each side of the square block is 1/scale degrees.
    private static final double scale = 100.0;
    private int lat;
    private int lon;

    // Construct a new uninitialized block.
    private Nola911Block()
    {
    }

    // Construct a new block from the given 911 call record.
    public Nola911Block
        (String record)
    {
        int a = record.indexOf ('"', 0);
        if (a == -1) throw new IllegalArgumentException();
        int b = record.indexOf (',', a + 1);
        if (b == -1) throw new IllegalArgumentException();
        int c = record.indexOf (')', b + 2);
    }

Listing 38.1. Nola911Block.java (part 1)
• **title** is the title for the graphic.

  • **plotfile** is the name of the file in which to store the graphic. You can use the View program in the Parallel Java 2 Library to view the graphic, store the graphic as a PNG image, and so on.

I have to make some design decisions. For the source objects, the map-reduce job will use class TextDirectorySource, like the climate data analysis job in the previous chapter. The mappers’ output (key, value) pairs will consist of a grid block (key) and the number of calls that occurred in that block (value). The value data type will be a reduction variable encapsulating an integer, namely class IntVbl, with a reduction operation of summation, namely subclass IntVbl.Sum, like the concordance and web log analysis jobs in previous chapters. I’ll have to write the grid block data type myself.

Listing 38.1 is the code for class edu.rit.pjmr.example.Nola911Block, the mappers’ output key data type. The `lat` and `lon` fields (lines 13–14) are the latitude and longitude of the block’s lower left corner, rounded down to the nearest multiple of 0.01 degree. The constructor (line 22) takes a record from the data set as its argument, extracts the latitude and longitude from the final field of the record, rounds them down, and stores the rounded values in the `lat` and `lon` fields. Thus, locations that fall within the same 0.01-degree-square block end up with identical `lat` and `lon` values. The class has several accessor methods to retrieve the geographic coordinates (lines 37–61).

Because Nola911Block objects will be keys in a hashed data structure, namely the combiner, the class must override the `equals()` and `hashCode()` methods. The `equals()` method (line 64) returns true if the two objects being compared represent the same grid block, that is, if they have the same latitude and longitude. The `hashCode()` method (line 74) returns equal hash codes for equal grid block objects and unequal hash codes for unequal grid block objects.

Like all key and value objects in a PJMR job, class Nola911Block must be streamable (line 7), must have a no-argument constructor (line 17), and must implement the `writeOut()` and `readIn()` methods (lines 80–94).

Class `edu.rit.pjmr.example.Nola911` (Listing 38.2) has the PJMR job main program (lines 22–59) as well as nested mapper, customizer, and reducer classes. The main program configures one mapper task for each backend node specified on the command line, passing the call type to each mapper, and sourcing data from the specified directory. The program configures one reducer task, with a customizer, passing the graphic’s title and the plot file name to the reducer. To be able to write the output file into the user’s account, the reducer task runs in the job process (line 54).

Processing each input record, the mapper’s `map()` method (lines 86–107) extracts the call type. If this matches the call type being analyzed, the `map()` method extracts the geographic location as a Nola911Block object. If the lo-
if (c == -1) throw new IllegalArgumentException();
lat = parse(record.substring(a + 2, b));
lon = parse(record.substring(b + 2, c));
}

// Returns the latitude of the lower left corner of this block in
// units of 0.01 degree.
public int lat() {
    return lat;
}

// Returns the longitude of the lower left corner of this block
// in units of 0.01 degree.
public int lon() {
    return lon;
}

// Returns the latitude of the lower left corner of this block
// (degrees).
public double latitude() {
    return lat/scale;
}

// Returns the longitude of the lower left corner of this block
// (degrees).
public double longitude() {
    return lon/scale;
}

// Determine if this block is equal to the given object.
public boolean equals(Object obj) {
    return
        (obj instanceof Nola911Block) &&
        (this.lat == ((Nola911Block)obj).lat) &&
        (this.lon == ((Nola911Block)obj).lon);
}

// Returns a hash code for this block.
public int hashCode() {
    return (lat << 16) + lon;
}

// Write this block to the given out stream.
public void writeOut(OutStream out) throws IOException {
    out.writeInt(lat);
    out.writeInt(lon);
}

Listing 38.1. Nola911Block.java (part 2)
cation is not “(0.0, 0.0)”, the mapper adds a (key, value) pair to the combiner, where the key is the Nola911Block object and the value is an IntVbl.Sum object with value 1 (line 74). This increments the count associated with the appropriate grid block in the combiner.

When the mapper tasks finish, the (key, value) pairs in the mappers’ combiners are sent via tuple space to the reducer task and are accumulated into the reducer task’s combiner. Thus, the reducer task ends up with the total number of events of the specified call type that occurred in each grid block.

The reducer task was configured with a customizer (lines 110–137). The customizer class does not define the comesBefore() method, so the pairs in the combiner are not sorted. However, the customizer class does define the start() method. The reducer task calls the customizer’s start() method, once only, in a single thread, before feeding the (key, value) pairs to the reducer. The customizer’s start() method lets you do preprocessing operations. In this job, the start() method scans all the pairs in the combiner to extract certain quantities that will be needed to generate the output graphic, namely the minimum latitude, minimum longitude, maximum latitude, maximum longitude, and maximum count of any block in the grid. (The customizer also has a finish() method, which the reducer task calls at the very end of the job, once only, in a single thread, to do postprocessing operations if any. For further information, refer to the Javadoc for class Customizer.)

Next, the reducer task calls the reducer’s start() method (lines 155–168). It records the graphic’s title and the plot file name from the job’s command line. It obtains parameters for the graphic by querying the customizer. It also defines \( B \), the height and width of a grid block in pixels; \( W \), the width of the whole grid; \( H \), the height of the whole grid; and logMaxCount, which will be used to determine the color of each grid block.

The reducer task calls the reducer’s reduce() method (lines 171–183) repeatedly, passing in a different (key, value) pair from the combiner each time. The reduce() method prints the grid block’s latitude, longitude, and count for informational purposes. However, the reduce() method’s chief purpose is to draw the colored grid blocks in the graphic. To do so, it uses the Parallel Java 2 Library packages edu.rit.draw and edu.rit.draw.item.

Package edu.rit.draw.item has classes for rectangles, ellipses, polygons, lines, arcs, arrows, text, and other kinds of shapes. You can specify a line’s width, color, and style (e.g., solid, dotted, dashed). You can specify a shape’s interior color and outline. To create a “drawing,” you construct an instance of a drawing item class; call methods to set the drawing item’s location and attributes; add the item to the drawing; and do the same for other drawing items. Lastly, you write the drawing to a file, then you use the View program to display the drawing. (The climate data analysis program in the previous chapter did something similar to create its temperature plot, using a different
Chapter 38. Map-Reduce Meets GIS

503

// Read this block from the given in stream.
public void readIn(InStream in)
    throws IOException
{
    lat = in.readInt();
    lon = in.readInt();
}

// Parse the given latitude or longitude string.
private static int parse(String s)
{
    return (int) Math.floor (scale*Double.parseDouble(s));
}

Listing 38.1. Nola911Block.java (part 3)

package edu.rit.pjmr.example;
import edu.rit.draw.Drawing;
import edu.rit.draw.item.*;
import edu.rit.pj2.vbl.IntVbl;
import edu.rit.pjmr.Combiner;
import edu.rit.pjmr.Customizer;
import edu.rit.pjmr.Mapper;
import edu.rit.pjmr.PjmrJob;
import edu.rit.pjmr.Reducer;
import edu.rit.pjmr.TextDirectorySource;
import edu.rit.pjmr.TextId;
import edu.rit.util.Action;
import edu.rit.util.Pair;
import java.awt.Font;
import java.io.File;
import java.io.IOException;
import java.util.Date;
public class Nola911
    extends PjmrJob<TextId,String,Nola911Block,IntVbl>
{
    // PJMR job main program.
    public void main(String[] args)
    {
        // Parse command line arguments.
        if (args.length != 5) usage();
        String[] nodes = args[0].split ("",");
        String directory = args[1];
        String callType = args[2];
        String title = args[3];
        String plotfile = args[4];

        // Determine number of mapper threads.
        int NT = Math.max (threads(), 1);

        // Print provenance.
        System.out.printf
            ("$ java pj2 threads=%d edu.rit.pjmr.example.Nola911", NT);

    }

Listing 38.2. Nola911.java (part 1)
package edu.rit.numeric.plot.) For further information, refer to the Javadoc for packages edu.rit.draw and edu.rit.draw.item.

Returning to the reducer, the reduce() method creates a rectangle drawing object; sets the width and height to B pixels; sets the lower left (“south-west”) corner based on the grid block’s latitude and longitude; sets the interior color based on the grid block’s count; sets the outline to null, meaning don’t draw an outline around the rectangle; and adds the rectangle to the drawing. The interior color is specified in hue-saturation-brightness (HSB) format. The hue is proportional to the logarithm of the grid block’s count, scaled so that a count of 1 yields a hue of 1/3 (green) and a count equal to the maximum count yields a hue of 0 (red). Calculating the hue from the logarithm of the count, rather than the count itself, results in finer resolution for the smaller, more frequent count values and coarser resolution for the larger, less frequent count values.

After running through all the (key, value) pairs, the reducer task calls the reducer’s finish() method (line 194). The finish() method adds the finishing touches to the graphic. It adds a rectangle around the grid (lines 201–202); adds the title above the grid (lines 203–205); adds the grid lines every 0.1 degrees inside the grid, along with their labels (lines 206–221); adds the legend at the right, consisting of a stack of thin rectangles each of a different hue (lines 222–226); and adds labels next to the legend (lines 227–231). Finally, the finish() method writes the drawing to the plot file (lines 232–239). Done!

I downloaded the eight 911 call record data set files from the City of New Orleans’s web site and scattered the records among the tardis cluster nodes, storing each node’s records in a file on the local hard disk. To generate the graphic in Figure 38.1, I ran the Nola911 PJMR job on the tardis cluster. The running time was 3.9 seconds. Here are the first few lines the program printed, beginning with the provenance:

```
$ java pj2 threads=1 edu.rit.pjmr.example.Nola911
dr00,dr01,dr02,dr03,dr04,dr05,dr06,dr07,dr08,dr09
/var/tmp/ark/NOLA911 94 New Orleans Discharging Firearm 911
Calls 2011-2018 94.dwg
Tue Jul 10 14:19:24 EDT 2018

Lat    Lon     Count
29.97  -90.13  61
29.97  -90.12  150
29.97  -90.11  215
29.97  -90.10  105
29.97  -90.09  323
29.97  -90.08  542
29.97  -90.07  599
29.97  -90.06  799
29.97  -90.05  265
30.06  -89.97  6
30.06  -89.96  100
```
for (String arg : args)
    System.out.printf (" %s", arg);
System.out.println();
System.out.printf ("%s%n", new Date());
System.out.printf ("Lat\tLon\tCount%n");
System.out.flush();

// Configure mapper tasks.
for (String node : nodes)
    mapperTask (node)
        .source (new TextDirectorySource (directory))
        .mapper (NT, MyMapper.class, callType);

// Configure reducer task.
reducerTask()
    .runInJobProcess()
    .customizer (MyCustomizer.class)
    .reducer (MyReducer.class, title, plotfile);
startJob();
}

// Print a usage message and exit.
private static void usage()
{
    System.err.println ("Usage: java pj2 [threads=<NT>] " +
                      "edu.rit.pjmr.example.Nola911 <nodes> <directory> " +
                      "<callType> <title> <plotfile>");
    terminate (1);
}

// Mapper class.
private static class MyMapper
    extends Mapper<TextId,String,Nola911Block,IntVbl>
{
    private static final IntVbl ONE = new IntVbl.Sum (1);
    private String callType;

    // Record call type.
    public void start
        (String[] args,
           Combiner<Nola911Block,IntVbl> combiner)
    {
        this.callType = args[0];
    }

    // Process one data record.
    public void map
        (TextId id,
           String data,
           Combiner<Nola911Block,IntVbl> combiner)
    {
        int a = data.indexOf (',', 0);
        if (a == -1) return;
        int b = data.indexOf (',', a + 1);
        if (b == -1) return;
        if (! data.substring (a + 1, b) .equals (callType)) return;

Listing 38.2. Nola911.java (part 2)
**Figure 38.2.** New Orleans 911 call graphic, type 24

**Figure 38.3.** New Orleans 911 call graphic, type 52F
try {
    Nola911Block block = new Nola911Block (data);
    if (block.latitude() == 0.0 || block.longitude() == 0.0)
        return;
    combiner.add (block, ONE);
} catch (Throwable exc) {
}

// Customizer class for reducer task.
private static class MyCustomizer
    extends Customizer<Nola911Block,IntVbl>
    {
        int minLat = Integer.MAX_VALUE;
        int minLon = Integer.MAX_VALUE;
        int maxLat = Integer.MIN_VALUE;
        int maxLon = Integer.MIN_VALUE;
        int maxCount = Integer.MIN_VALUE;

        public void start
            (String[] args,
                 Combiner<Nola911Block,IntVbl> combiner)
            {
                combiner.forEachItemDo
                    (new Action<Pair<Nola911Block,IntVbl>>()
                        {
                            public void run (Pair<Nola911Block,IntVbl> pair)
                                {
                                    minLat = Math.min (minLat, pair.key().lat());
                                    minLon = Math.min (minLon, pair.key().lon());
                                    maxLat = Math.max (maxLat, pair.key().lat());
                                    maxLon = Math.max (maxLon, pair.key().lon());
                                    maxCount = Math.max (maxCount, pair.value().intValue());
                                        }
                                    });
            }

        // Reducer class.
        private static class MyReducer
            extends Reducer<Nola911Block,IntVbl>
            {
                private static final int B = 12;
                String title;
                File plotfile;
                int minLat;
                int minLon;
                int maxLat;
                int maxLon;
                int maxCount;
                double logMaxCount;
                int W;
                int H;

Listing 38.2. Nola911.java (part 3)
I also ran the program to analyze call type 24 (ambulance request) and call type 52F (fire). Figures 38.2 and 38.3 are the resulting graphics.

**Under the Hood**

Now that we’ve seen several examples of PJMR jobs, let’s recap what goes on inside the Parallel Java 2 Library. A PJMR job consists of one or more mapper tasks and one reducer task. Each mapper task and each reducer task runs in a separate process on a backend node of the cluster, or (optionally) in the job process on the frontend node.

A mapper task consists of one or more source objects, a customizer object, and a combiner object. Each source in turn has one or more mapper objects reading records from that source. The mapper task does the following, using the configuration specified in the PJMR job main program:

- For each of the mapper task’s configured sources, create the source object and call its `open()` method.
- For each source’s configured mappers, create the mapper object.
- Create the mapper task’s configured combiner. This is the global combiner. If no combiner was configured, an instance of the Combiner base class is created.
- Create the mapper task’s configured customizer. If no customizer was configured, an instance of the Customizer base class (which does nothing) is created.
- In a single thread, call the customizer’s `start()` method, passing in any configured argument strings plus the global combiner.
- Run the mapper(s) in parallel, each in its own separate thread. Each mapper thread does the following:
  - Create a thread-local copy of the global combiner.
  - Call the mapper’s `start()` method, passing in any configured argument strings plus the thread-local combiner.
  - Repeatedly call the mapper’s source’s `next()` method to obtain the next data record from the source, then call the mapper’s `map()` method, passing in the data record’s ID and contents plus the thread-local combiner. (Note that if there is more than one mapper for the same source, all the mappers call the source’s `next()` method concurrently; the `next()` method is multiple thread safe.)
  - When there are no more data records, call the mapper’s `finish()` method, passing in the thread-local combiner.
- After all the mapper threads have finished, reduce the thread-local combiners into the global combiner.
- In a single thread, call the customizer’s `finish()` method, passing in the global combiner.
// Initialize.
public void start (String[] args)
{
    title = args[0];
    plotfile = new File (args[1]);
    MyCustomizer c = (MyCustomizer) customizer();
    minLat = c.minLat;
    minLon = c.minLon;
    maxLat = c.maxLat;
    maxLon = c.maxLon;
    maxCount = c.maxCount;
    logMaxCount = Math.log (c.maxCount);
    W = B*(maxLon - minLon + 1);
    H = B*(maxLat - minLat + 1);
}

// Process one 911 block.
public void reduce
(Nola911Block key,
IntVbl value)
{
    System.out.printf ("%.2f %.2f %s \n",
        key.latitude(), key.longitude(), value);
    System.out.flush();
    new Rectangle() .width (B) .height (B)
        .sw (B*(key.lon() - minLon), -B*(key.lat() - minLat))
        .fill (new ColorFill()
            .hsb (hueFor (value.intValue()), 1.0f, 1.0f))
        .outline (null) .add();
}

// Returns the hue for the given count.
private float hueFor
(int count)
{
    return (float)
        ((logMaxCount - Math.log(count))/(3*logMaxCount));
}

// Store graphic in plotfile.
public void finish()
{
    System.out.printf ("Max %d \n", maxCount);
    Text.defaultFont
        (new Font (Font.SANS_SERIF, Font.PLAIN, 10));
   OutlinedItem.defaultOutline
        (new SolidOutline() .width (0.5f));
    Rectangle r = new Rectangle() .width (W) .height (H)
        .sw (0, 0) .fill (null) .add();
    new Text() .text (title) .s (r.n().n(B/2))
        .font (new Font (Font.SANS_SERIF, Font.PLAIN, 14))
        .add();
    for (int i = minLon; i <= maxLon; ++ i)
        if ((i % 10) == 0)
            { Line l = new Line() .to (B*(i - minLon), 0) .vby (-H)
                .add();
Listing 38.2. Nola911.java (part 4)
• Call each source’s close() method.
• Send the (key, value) pairs from the global combiner to the reducer task via tuple space.

The reducer task consists of a customizer object, a combiner object, and one or more reducer objects. The reducer task does the following, using the configuration specified in the PJMR job main program:

• For each of the reducer task’s configured reducers, create the reducer object.
• Create the reducer task’s configured combiner. If no combiner was configured, an instance of the Combiner base class is created.
• Create the reducer task’s configured customizer. If no customizer was configured, an instance of the Customizer base class (which does nothing) is created.
• Receive (key, value) pairs from each mapper task via tuple space. Reduce all the received pairs together into the reducer task’s combiner.
• If the customizer’s comesBefore() method was overridden, sort the combiner’s (key, value) pairs into the order determined by the comesBefore() method. Otherwise, leave the combiner’s (key, value) pairs in an unspecified order.
• In a single thread, call the customizer’s start() method, passing in any configured argument strings plus the combiner.
• Do a parallel loop over the (key, value) pairs in the combiner. Each parallel loop iteration processes one (key, value) pair. There is one parallel team thread for each reducer. The reducer task’s schedule and chunk properties determine how the parallel loop iterations are partitioned among the parallel team threads; that is, how the (key, value) pairs are partitioned among the reducers. Each parallel team thread does the following:
  ◦ Call the reducer’s start() method, passing in any configured argument strings.
  ◦ Repeatedly call the reducer’s reduce() method, passing in one of the combiner’s (key, value) pairs.
  ◦ When there are no more pairs, call the reducer’s finish() method.
• After all the parallel team threads have finished, call the customizer’s finish() method in a single thread, passing in the combiner.

The customizer’s start() and finish() methods let you do single-threaded preprocessing and postprocessing operations at the beginning and end of the whole mapper task or reducer task. The mapper’s start() and finish() methods let you do preprocessing and postprocessing operations in each separate mapper thread at the beginning and end of the parallel region.
in the mapper task. The reducer’s start() and finish() methods let you do
preprocessing and postprocessing operations in each separate reducer thread
at the beginning and end of the parallel region in the reducer task.

Points to Remember

- “The purpose of computing is insight, not numbers.”
  —Richard W. Hamming
- Consider whether presenting analysis results in the form of graphics
  might yield better insights. However, do present analysis results as tables
  of numbers also.
- Use the Parallel Java 2 Library packages edu.rit.draw and edu.rit-
  .draw.item to create drawings right in your program.
- Store a drawing object in a file for later viewing. Use the View program
to view the drawing.

Listing 38.2. Nola911.java (part 5)
• A task that needs to read or write files in the user’s account should be specified to run in the job process on the cluster’s frontend node.

• The class for the mapper’s output key data type must be suitable for use as the key in a hashed data structure; it must define the equals() and hashCode() methods appropriately.

• The classes for the mapper’s output key and output value data types must be streamable; they must implement interface edu.rit.io.Streamable, must define a no-argument constructor, and must define the writeOut() and readIn() methods appropriately.