Chapter 36
Cluster Map-Reduce

► 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 parallel map-reduce workflow I introduced in the previous chapter featured multiple raw data sources, each with its own source, mapper, and combiner objects, followed by an overall combiner object and a reducer object. To speed up the application, the portions of the workflow associated with each data source were executed in parallel by separate threads.

I ran the program on a PC to analyze four text files containing novels by Charles Dickens. But this meant that the four raw data sources all resided on the same disk drive. Consequently, the disk drive interface became a bottleneck that to some extent nullified the program’s parallelism. Even though the program ran in multiple threads on multiple cores, all the threads had to take turns reading their raw data sources sequentially through the one disk drive interface.
To get full parallelism during the mapping stage, each raw data source must reside on a separate disk drive with its own disk drive interface. Then the raw data sources no longer all have to go through the same bottleneck. Each parallel thread can read its own raw data source simultaneously with all the other threads.

It’s possible to attach multiple disk drives to a node, just as a node can have multiple cores. But there is a limit to the number of disk drives a single node can handle, just as there is a limit to the number of cores on one node. And there is a limit to the amount of data that can be stored on one disk drive, just as there is a limit to the amount of main memory on one node.

Scaling up to big data analysis jobs requires shifting from a single-node computer to a cluster parallel computer. A cluster is not limited by the amount of disk storage a single node can handle. If you need more storage, add more nodes (with their disk drives) to the cluster. Of course, adding more nodes also means adding more cores to the cluster, thus also increasing the parallelism available for computing with the data on the new disk drives.

Figure 36.1 shows a PJMR map-reduce job running in the environment for which PJMR was designed, namely a cluster. I’ll point out three features of the PJMR architecture.

First, the map-reduce job consists of multiple mapper tasks containing source, mapper, and combiner objects, as well as a reducer task containing combiner and reducer objects. Each mapper task runs on its own separate node in the cluster, in parallel with the other mapper tasks. Each mapper task’s raw data source resides on the node’s disk drive. This way, each source object can read its own disk drive without interference from other source objects, and the source object can download raw data at up to the full bandwidth of the disk interface. Consequently, the effective download bandwidth for the whole job is one disk’s bandwidth multiplied by the number of nodes.
This eliminates the bottleneck that would occur if all the raw data resided on the same disk drive.

Second, sometimes the map-reduce job’s running time can be decreased by configuring each mapper task with *multiple mapper objects*, running in parallel in multiple threads (cores) on the node, all getting records from the one source object. Why might this improve the performance? Because typically, the source object reads raw data from the disk drive in large chunks, then breaks the chunks into separate records. Multiple mapper objects running in multiple threads can then process these multiple records simultaneously. Also, if one thread has to stall waiting for the source to read the next chunk off the disk drive, another thread can still be occupied processing a previous record.

Third, PJMR follows the *cluster parallel reduction pattern*. Each mapper object has its own per-thread combiner. These are reduced together within each mapper task, yielding a per-task combiner. The \((K,V)\) pairs in the per-task combiners are sent via tuple space to the reducer task, where they are absorbed into the overall combiner. This means that the key data type \(K\) and the value data type \(V\) must be streamable or serializable, so that they can be sent across the cluster’s backend network encapsulated inside tuples.

Keep in mind that these architectural features are hidden inside PJMR. Even when running a PJMR map-reduce job on a cluster parallel computer, the only code you have to write is still just the mapper object, reducer object, and job main program, plus possibly the source object and the reduction variable. PJMR takes care of the rest.

To illustrate a cluster parallel PJMR map-reduce job, I’ll analyze some web server log data. Courtesy of our system administrator, I obtained ten days’ worth of logs from the RIT Computer Science web server. Each daily log is a text file. Each line of text is a log entry containing the IP address that made a web request, the date and time of the request, the URL that was requested, and other information. Here is a small portion of one log file:

```
157.55.39.63 - - [10/Jul/2015:00:00:03 -0400] "GET /~vcss243/La
66.249.73.178 - - [10/Jul/2015:00:00:07 -0400] "GET /images/Bay
207.46.13.80 - - [10/Jul/2015:00:00:12 -0400] "GET /~vcss233/pu
180.76.15.163 - - [10/Jul/2015:00:00:12 -0400] "GET /~mxt4877/C
180.76.15.158 - - [10/Jul/2015:00:00:17 -0400] "GET /~ats/ds-20
129.21.36.110 - - [10/Jul/2015:00:00:19 -0400] "GET /~displays/
157.55.39.78 - - [10/Jul/2015:00:00:21 -0400] "GET /usr/local/p
157.55.39.39 - - [10/Jul/2015:00:00:31 -0400] "GET /usr/local/p
80.240.138.58 - - [10/Jul/2015:00:00:32 -0400] "GET /~ark/pj2/d
180.76.15.148 - - [10/Jul/2015:00:00:35 -0400] "GET /~ats/java-
```

I partitioned the log files into ten chunks of roughly equal size and stored one chunk on the disk drive of each of the tardis cluster’s ten backend nodes, which are named \(dr00\) through \(dr09\). The log files occupy about 400 megabytes total, or 40 megabytes on each node—a medium size data set.
A PJMR job usually assumes the data to be analyzed is already located on the cluster’s disk drives. You have to do the data partitioning yourself. (The Parallel Java 2 Library includes some programs to help with data partitioning, in package edu.rit.pjmr.util; refer to the Javadoc for further information.) However, you only need to partition the data once. Thereafter, you can run as many map-reduce jobs on the data as you want, doing various kinds of analysis.

For the first web log analysis program, I want to make a list of all the computers (IP addresses) that made web requests to the RIT CS web server, as well as the number of web requests each computer made. I want the list to be sorted so that the computers that made the most web requests come first;
if multiple computers made the same number of web requests, I want the smaller IP addresses to come first. Listing 36.1 gives the program for the PJMR job, class edu.rit.pjmr.example.WebLog01. The program is run with this command:

```
$ java pj2 [threads=NT] edu.rit.pjmr.example.WebLog01 nodes \file [pattern]
```

The program’s command line arguments are as follows:

- `NT` is the number of mapper threads in each mapper task (default: 1).
- `nodes` is a comma-separated list of one or more backend node names in the cluster. The program will run a mapper task on each node.
- `file` is the name of the web log file to be analyzed. The program assumes that each node has a file with this name, containing a portion of the web log data.
- `pattern` is an optional argument. If present, the pattern must be a regular expression as specified in Java platform class java.util.regex.Pattern, and the program will analyze only the web log entries that match the pattern. For example, the pattern could be a date, to limit the analysis to that date. If the `pattern` argument is omitted, the program will analyze all the web log entries.

The main program class (line 12) extends class PjmrJob (line 13). Because I will be using a TextFileSource as I did in the previous chapter, the source’s record ID and record contents data types are TextId and String. The program will be mapping IP addresses to number of occurrences, similar to the concordance program in the previous chapter. Thus, the mapper’s `(K,V)` data types are class edu.rit.pjmr.example.IPAddress for the key `K` and class LongVbl for the value `V`. (Class IPAddress is a separate class that stores an IP address in the form of an integer and that has a method to decide whether one IP address is smaller than another.)

After parsing the command line arguments, the program prints its provenance (lines 34–41). The provenance includes the command line used to run the program and the date and time when the program was run. This information is useful when examining the program’s output at a later time. This information is also useful if the same analysis needs to be performed again.

For each backend node name specified (line 44), the program configures a mapper task to run on that node (line 45). The mapper task’s source object is a TextFileSource reading the given web log file (line 46). The mapper task is configured with `NT` mapper objects, each an instance of class MyMapper; the `pattern` argument is passed to each mapper object (line 47).

The program also configures a reducer task (line 50), which can run on any available node. The reducer task includes a customizer object that is an instance of class MyCustomizer (line 51) and a reducer object that is an in-
package edu.rit.pjmr.example;
import edu.rit.pj2.vbl.LongVbl;
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.TextFileSource;
import edu.rit.pjmr.TextId;
import java.util.Date;
import java.util.regex.Pattern;
public class WebLog01
extends PjmrJob<TextId,String,IPAddress,LongVbl>
{
   // PJMR job main program.
   public void main (String[] args)
   {
      // Parse command line arguments.
      if (args.length < 2 || args.length > 3) usage();
      String[] nodes = args[0].split (",");
      String file = args[1];
      String pattern = null;
      if (args.length >= 3)
      {
         pattern = args[2];
         Pattern.compile (pattern); // Verify that pattern compiles
      }

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

      // Print provenance.
      System.out.printf
      ("$ java pj2 threads=%d edu.rit.pjmr.example.WebLog01",
       NT);
      for (String arg : args)
      System.out.printf ("%s", arg);
      System.out.println();
      System.out.printf ("%s%n", new Date());
      System.out.flush();

      // Configure mapper tasks.
      for (String node : nodes)
      mapperTask (node)
         .source (new TextFileSource (file))
         .mapper (NT, MyMapper.class, pattern);

      // Configure reducer task.
      reducerTask()
         .customizer (MyCustomizer.class)
         .reducer (MyReducer.class);

      startJob();
   }

Listing 36.1. WebLog01.java (part 1)
stance of class MyReducer (line 52). The customizer object lets you include extra functionality in the reducer task, as we will see later. (You can configure a mapper task with a customizer object too; I don’t need that in this program.)

Lastly, the program starts the map-reduce job (line 54).

The mapper class MyMapper (line 68) introduces another feature of PJMR. The Mapper base class actually defines three methods: start(), map(), and finish()—not unlike the start(), run(), and finish() methods of a parallel loop. The mapper task operates as follows. Each thread in the mapper task gets its own separate mapper object. The thread calls the mapper object’s start() method, once only, at the start of the map-reduce job. The start() method’s arguments are an array of argument strings passed to the mapper task from the job main program, and a reference to the per-thread combiner. The start() method can do initializations, if necessary, before commencing the processing of records. If the start() method is omitted, the default is to do nothing. Next, the thread repeatedly gets a record from the mapper task’s source object and passes the record’s ID and contents, along with the per-thread combiner reference, to the mapper object’s map() method. As we have already seen, the map() method extracts relevant information from the record and adds zero or more (K,V) pairs to the combiner. After the source object signals that there are no more records, the thread calls the mapper object’s finish() method, once only, which can do finalizations if necessary. If the finish() method is omitted, the default is to do nothing.

In the WebLog01 program, the MyMapper class has a start() method (lines 74–80). The args argument is an array of strings specified back in the job main program, when the mapper object was configured (line 47):

```java
.mapper (NT, MyMapper.class, pattern);
```

The argument strings, if any, appear after the mapper class name. There can be any number of argument strings, including none. Here, there is one argument string, namely the pattern (regular expression) that a web log entry must match in order to be included in the analysis, or null if the user did not specify a pattern. If a pattern was specified, the start() method compiles the pattern for later use and stores the resulting Pattern object in the pattern field (line 72); otherwise, the pattern field remains null.

Note how information specified by the user—in this case, the pattern—flows from the pj2 command line to the job main program and from there to all the mapper objects, via the argument strings. This is how user input can affect the analysis the map-reduce job performs.

The MyMapper class’s map() method (line 82) first checks whether the contents argument, which is the contents of one record (line) from the web log file, matches the pattern the user specified, if any (line 87). If the line matches, or if no pattern was specified, the line is to be included in the analy-
Listing 36.1. WebLog01.java (part 2)
sis. The map() method then extracts the initial portion of the line up to the first whitespace character; this is the IP address string. The map() method constructs an IPAddress object from that string and adds the IPAddress object as the key, along with a LongVbl.Sum object initialized to 1 as the value (line 71), to the combiner. Thus, the number of occurrences associated with the IP address in the combiner is increased by 1. (This is similar to what the concordance program in the previous chapter did.) The WebLog01 program doesn’t care about the rest of the web log file line, so the map() method just ignores the rest.

The MyCustomizer class (line 100) introduces yet another feature of PJMR. You can configure a mapper task or a reducer task, or both, with a customizer object. Here, I configured a customizer object only in the reducer task (line 51). The Customizer base class defines three methods: start(), finish(), and comesBefore(). Here, I only need the comesBefore() method. (See the Javadoc for class Customizer for descriptions of the other methods.) If a reducer task includes a customizer object, and if the customizer object’s comesBefore() method is defined, the reducer task will sort the (K,V) pairs in the overall combiner before sending the pairs to the reducer object. The comesBefore() method’s arguments are the key and value from one (K,V) pair and the key and value from another (K,V) pair. The comesBefore() method must return true if the first pair comes before the second pair in the desired sorted order. If the reducer task does not include a customizer object, or if the reducer task includes a customizer object but the comesBefore() method is not defined, the reducer task will not sort the (K,V) pairs. (This is what happened in the concordance program in the previous chapter.)

In the WebLog01 program, a (K,V) pair consists of an IP address (key) and a number of occurrences (value). The reducer task’s customizer’s comesBefore() method (line 103) compares the pairs such that pairs with larger numbers of occurrences come first. For equal numbers of occurrences, pairs with smaller IP addresses come first. This puts IP addresses that made the most requests at the top of the program’s output.

Finally, the MyReducer class (line 117) generates the WebLog01 program’s output. The reduce() method’s arguments are the key and the value of one (K,V) pair from the overall combiner. Remember, these are fed to the reduce() method in sorted order. The reduce() method simply prints the number of occurrences (the value) followed by the IP address (the key).

I ran the WebLog01 program over the ten days’ worth of web logs on the tardis cluster. Because I had partitioned the web log files among all ten backend nodes dr00 through dr09, I specified all the node names in the command, causing the program to run a mapper task on every node. I also specified threads=2, causing the program to run two mapper objects in separate threads in each mapper task. This reduced the program’s running time by
private static class MyReducer
    extends Reducer<IPAddress, LongVbl>
    {
        public void reduce
            (IPAddress key, // IP address
             LongVbl value) // Number of requests
            {
                System.out.printf("%s\t%s\n", value, key);
                System.out.flush();
            } } 

Listing 36.1. WebLog01.java (part 3)

private static class MyMapper
    extends Mapper<TextId, String, String, LongVbl>
    {
        private static final LongVbl ONE = new LongVbl.Sum(1L);
        private static final Pattern getPattern =
            Pattern.compile("GET /([^t\nxf\r/]+)");
        private Pattern ipaddrPattern;
        private Pattern pattern;

        public void start
            (String[] args, // File name and line number
             Combiner<String, LongVbl> combiner)
            {
                ipaddrPattern = Pattern.compile("^" + args[0] + "\\s");
                if (args[1] != null)
                    pattern = Pattern.compile(args[1]);
            } 

        public void map
            (TextId id, // File name and line number
             String contents, // Line from file
             Combiner<String, LongVbl> combiner)
            {
                if ((pattern == null || pattern.matcher(contents).find())
                    && ipaddrPattern.matcher(contents).find())
                    {
                        Matcher m = getPattern.matcher(contents);
                        if (m.find())
                            combiner.add(m.group(1), ONE);
                    } } 

Listing 36.2. WebLog05.java, class MyMapper
about 23 percent, from 11.5 seconds down to 8.9 seconds. Here are the first few lines the program printed (beginning with the provenance):

```
$ java pj2 threads=2 edu.rit.pjmr.example.WebLog01
dr00,dr01,dr02,dr03,dr04,dr05,dr06,dr07,dr08,dr09
/var/tmp/ark/weblog/weblog.txt
Mon Aug 03 08:57:39 EDT 2015
907360  108.45.93.78
38119   129.21.36.111
28307   129.21.36.110
22890   68.180.229.52
19579   144.76.63.35
13929   32.208.223.22
9715    157.55.39.63
7474    66.249.67.46
7414    66.249.67.39
```

Wait a minute. During those ten days, one IP address—108.45.93.78—made over nine hundred thousand web requests. A DNS lookup shows that this IP address is associated with a host name that I believe represents a Verizon home customer. What URLs was that person, or web crawler as the case may be, requesting?

To find out, I need to do a different analysis. Given a particular computer (IP address), I want a list of the URLs—actually, just the top-level directories of the URLs—that the computer requested, as well as the number of web requests for each URL. I want the list to be sorted so that the most frequently requested URLs come first; if multiple URLs were requested the same number of times, I want the URLs to appear in alphabetical order.

I wrote another PJMR map-reduce program, class edu.rit.pjmr.example.WebLog05. Most of the WebLog05 program is quite similar to the WebLog01 program, so I have listed only the significantly different piece, the mapper class (Listing 36.2). On the WebLog05 command line, the user specifies an IP address. The web log lines that begin with that IP address (and that match an optional regular expression pattern if any) are analyzed. For each such line, the mapper extracts the URL that follows the string "GET /". If the URL refers to a document in a subdirectory, only the top-level directory (up to the next "/" character) is extracted. The program counts and prints the number of occurrences of each unique URL.

I ran the WebLog05 program over the web logs on the tardis cluster, specifying IP address 108.45.93.78. The program run took 2.4 seconds—less time than the previous run, because the program didn’t have to analyze every web log entry. Here are the first few lines the program printed:

```
$ java pj2 threads=2 edu.rit.pjmr.example.WebLog05
dr00,dr01,dr02,dr03,dr04,dr05,dr06,dr07,dr08,dr09
/var/tmp/ark/weblog/weblog.txt 108.45.93.78
Mon Aug 03 09:37:30 EDT 2015
404807  ~ark
Chapter 36. Cluster Map-Reduce

This person appears to be requesting a dump of the entire RIT CS web site, for purposes unknown. The above URLs refer to the personal web sites of RIT CS faculty. The top web site belongs to user ~ark, who happens to be me. (I have a pretty big web site. It includes web pages for all the courses I’ve taught over several decades, as well as Javadoc web pages for several large software libraries I’ve written, including the Parallel Java 2 Library.)

To narrow down when this person made all these web requests, I ran the WebLog05 program again, this time specifying a specific date as the matching pattern. Here is what one of the program runs (running time: 2.5 seconds) printed, showing what this person requested on July 12, 2015:

$ java pj2 threads=2 edu.rit.pjmr.example.WebLog05
/var/tmp/ark/weblog/weblog.txt 108.45.93.78 12/Jul/2015
Mon Aug 03 09:40:27 EDT 2015
132682  usr
26454  ~ark
9133  ~hp
7083  ~ats
5574  ~wrc
5867  ~rp
2019  ~af
1291  ~nc
1278  ~anhinga

And here is what this person requested on July 13, 2015:

$ java pj2 threads=2 edu.rit.pjmr.example.WebLog05
/var/tmp/ark/weblog/weblog.txt 108.45.93.78 13/Jul/2015
Mon Aug 03 09:40:50 EDT 2015
378351  ~ark
138257  usr
100413  ~ats
56741  %7Eats
2520  ~wrc
1369  ~hp
982  ~rp
698  ~cs
648  ~rwd

The chief takeaway message here is that map-reduce is a versatile pattern for analyzing large, unstructured data sets. You don’t have to set up a data-
base, with schemas, tables, and SQL; map-reduce works just fine to “query” plain text files. By writing various map-reduce programs, you can do as many different kinds of analysis as you want; and the map-reduce programs are neither lengthy nor complicated. You can run the analysis programs as many times as you want over the same data set, specifying different parameters each time. You can speed up the analysis programs by partitioning the data among the disk drives of a cluster parallel computer’s backend nodes and running the map-reduce jobs in multiple parallel tasks on the cluster.

**Under the Hood**

PJMR is designed so that the first stage of reduction happens in the combiner objects inside the mapper tasks. The resulting \((K,V)\) pairs are sent from each mapper task to the reducer task via tuple space. The second stage of reduction happens in the combiner object inside the reducer task.

This design has several consequences for PJMR map-reduce programs. As I mentioned earlier, the key data type \(K\) and the value data type \(V\) must be streamable or serializable, so they can be packaged into tuples and sent across the network.

Each per-thread combiner object in each mapper task, each per-task combiner in each mapper task, and the overall combiner in the reducer task are implemented as hash table data structures stored in main memory. Consequently, the \((K,V)\) pairs stored in each task’s combinners must fit in the main memory of the node where each task is running. The whole point of the map-reduce pattern is to reduce the raw data down to just the information essential to the analysis. PJMR assumes that this essential information is much smaller than the raw data and is therefore able to fit into the nodes’ main memories.

Specifically, the combiner objects are stored in the JVM’s heap. Depending on the size of your data, you might have to increase the JVM’s maximum heap size above the default. To do this, include the `jvmflags` option on the `pj2` command line. For example, the command

```
$ java pj2 jvmflags=-Xmx2000m ...
```

causes each backend process’s JVM to run with the flag `-Xmx2000m`, which sets the JVM’s maximum heap size to 2,000 megabytes. However, it’s a bad idea to set the maximum heap size larger than the node’s physical main memory size. If your map-reduce job needs more memory to store intermediate results than is available on your nodes, then you should be using a library like Hadoop, not PJMR. (Hadoop stores intermediate results on disk, not in main memory.)

Besides being stored in the nodes’ main memories, the mapper tasks’ combinners’ \((K,V)\) pairs have to be sent via tuple space to the reducer task. The amount of network traffic is proportional to the numbers and sizes of the
keys and values in all the combiners after the mapping stage has finished. Again, PJMR assumes that these \((K,V)\) pairs occupy much less space than the raw data, so that the map-reduce program spends most of its time running the mappers and comparatively little time sending and receiving tuples. If this is not the case, use Hadoop.

**Points to Remember**

- Large scale data analysis tasks need to run on a cluster.
- Partition the data set to be analyzed across the cluster backend nodes’ disk drives. Don’t store the whole data set on one disk drive.
- The program’s running time can potentially be reduced by configuring each mapper task with more than one mapper object. Each mapper object runs in a separate thread (core) on a backend node.
- Define the mapper class’s `start()`, `map()`, and `finish()` methods as necessary.
- Pass user-specified arguments from the job main program to the mapper objects as necessary. The arguments appear as an array of strings passed to the mapper object’s `start()` method.
- Define the reducer class’s `start()`, `reduce()`, and `finish()` methods as necessary.
- Pass user-specified arguments from the job main program to the reducer object as necessary. The arguments appear as an array of strings passed to the reducer object’s `start()` method.
- Configure a mapper task or a reducer task with a customizer object as necessary.
- To sort the final \((K,V)\) pairs into a desired order, configure the reducer task with a customizer object whose `comesBefore()` method defines the desired ordering.
- By writing various map-reduce programs, you can do as many different kinds of analysis of a data set as you want.
- You can run the analysis programs as many times as you want over the same data set, specifying different parameters each time.