Chapter 18
Massively Parallel

► Part I. Preliminaries
► Part II. Tightly Coupled Multicore
▼ Part III. Loosely Coupled Cluster
  Chapter 18. Massively Parallel
  Chapter 19. Hybrid Parallel
  Chapter 20. Tuple Space
  Chapter 21. Cluster Parallel Loops
  Chapter 22. Cluster Parallel Reduction
  Chapter 23. Cluster Load Balancing
  Chapter 24. File Output on a Cluster
  Chapter 25. Interacting Tasks
  Chapter 26. Cluster Heuristic Search
  Chapter 27. Cluster Work Queues
  Chapter 28. On-Demand Tasks
► Part IV. GPU Acceleration
► Part V. Big Data
► Appendices
Turning away from multicore parallel programming, we come to cluster parallel programming. A cluster parallel computer can have more—perhaps many more—cores than a single multicore node, hence can offer a higher degree of scalability than a single node. That’s good. But in a cluster, the memory is distributed and cannot be shared by all the cores, requiring the program to do interprocess communication to move data from node to node and increasing the program’s overhead. That’s bad. But because there isn’t a limit on the amount of memory in the cluster, as there is with a single node—if you need more memory, just add more nodes to the cluster—a memory-hungry parallel program can scale up to much larger problem sizes on a cluster. That’s good. On balance, extreme-scale parallel computing has to be done on a cluster. All of the world’s fastest supercomputers on the Top500 List are cluster parallel computers.

An easy way to utilize the parallelism of a cluster is to run multiple independent instances of a sequential program. As an example, consider the bitcoin mining program from Chapter 7. The sequential program in that chapter mined (computed a golden nonce for) just one bitcoin. But suppose I want to mine many bitcoins, K of them. All I have to do is execute K processes, each process running one thread on one core of the cluster, each process running the sequential bitcoin mining program on a different coin ID (Figure 18.1). No thread needs to access the memory of another thread, or communicate with another thread. Each thread mines its own bitcoin and prints the result.

A parallel program like this, where each thread runs without interacting with any other thread, is called a massively parallel program. The lack of interaction minimizes the overhead and leads to ideal or near-ideal weak scaling. If I had a 1,000-core cluster, I could mine 1,000 bitcoins simultaneously, and the program would finish in the same time as one of the bitcoins. Such a program is also called embarrassingly parallel—there’s so much parallelism, it’s embarrassing!

I could mine a bunch of bitcoins in a massively parallel fashion by logging into the cluster and typing in a bunch of commands to run multiple instances of the sequential program. However, it would be far better to automate this and let the parallel program itself fire up all the necessary processes and threads. That’s where the cluster parallel programming features of the Parallel Java 2 Library come in.

Listing 18.1 is the MineCoinClu program, a massively parallel cluster version of the bitcoin mining program. Let’s take it one line at a time.

Whereas a multicore parallel program consists of a single task, a cluster parallel program is expressed as a job consisting of one or more tasks. Accordingly, class MineCoinClu extends class edu.rit.pj2.Job (line 8) rather than class edu.rit.pj2.Task. The job defines the tasks that will run, but the job itself does not do any computations. The tasks do all the work.
**Figure 18.1.** Massively parallel program running on a cluster
To run a cluster parallel program, run the pj2 launcher program, giving it the job class name and any command line arguments. MineCoinClu’s first argument is $N$, the number of most significant zero bits needed in the digest; the remaining arguments are one or more coin IDs to be mined. For example:

```
$ java pj2 edu.rit.pj2.example.MineCoinClu \
   16 0123456789abcdef 3141592653589793
```

pj2 creates an instance of the job class, then calls the job’s `main()` method, passing in an array of the command line argument strings. The code in the `main()` method sets up the job’s tasks but does not actually execute the tasks. Once the `main()` method returns, pj2 proceeds to create and execute the tasks, running each task on some node in the cluster.

Let’s examine how the job’s tasks are set up. Line 20 loops over the job’s second and following command line arguments, setting up one task for each coin ID. A task is specified by calling the `rule()` method (line 21) to define a rule. The rule is an instance of class `edu.rit.pj2.Rule`. The rule object specifies several things:

- The rule specifies when the task is to be started. By default, the task is started when the job commences executing. (Other possibilities are to start the task later during the job’s execution, or to start the task at the end of the job after all the other tasks have finished; we will see examples of such rules later.)
- The rule specifies the task that is to be executed. This is done by calling the rule’s `task()` method, passing in the task’s class (line 21). In this case, the task will be an instance of class MineCoinTask, which appears further down in the listing (line 38).

The `task()` method returns a task spec, an instance of class `edu.rit.pj2.TaskSpec`. The task spec specifies several things:

- The task spec specifies the task’s command line arguments if any. This is done by calling the task spec’s `args()` method (line 21). In this case the task’s two command line arguments are the coin ID (which is the job’s command line argument `args[i]`) and $N$ (which is the job’s `args[0]`).
- The task’s default parallel loop properties—threads, schedule, and chunk—can be specified. (The MineCoinClu program doesn’t need to do this.)
- Other attributes of the task can also be specified; we will look at some of these later.

Class Rule and class TaskSpec are separate classes because a rule can also specify a task group consisting of multiple tasks—one rule with many task specs. MineCoinClu does not need this capability, but some of the cluster parallel programs in later chapters will.
package edu.rit.pj2.example;
import edu.rit.crypto.SHA256;
import edu.rit.pj2.Job;
import edu.rit.pj2.Task;
import edu.rit.util.Hex;
import edu.rit.util.Packing;

public class MineCoinClu
extends Job
{

    // Job main program.
    public void main
    (String[] args)
    {
        // Parse command line arguments.
        if (args.length < 2) usage();
        int N = Integer.parseInt (args[0]);
        if (1 > N || N > 63) usage();

        // Set up one task for each coin ID.
        for (int i = 1; i < args.length; ++ i)
            rule() .task (MineCoinTask.class) .args (args[i], args[0]);
    }

    // Print a usage message and exit.
    private static void usage()
    {
        System.err.println ("Usage: java pj2 " +
            "edu.rit.pj2.example.MineCoinClu <N> <coinid> " +
            "[<coinid> ...]");
        System.err.println ("<N> = Number of leading zero bits " +
            "(1 .. 63)");
        System.err.println ("<coinid> = Coin ID (hexadecimal)");
        terminate (1);
    }

    // Class MineCoinTask provides the Task that computes one coin
    // ID's nonce in the MineCoinClu program.
    private static class MineCoinTask
extends Task
    {
        // Command line arguments.
        byte[] coinId;
        int N;

        // Mask for leading zeroes.
        long mask;

        // For computing hash digests.
        byte[] coinIdPlusNonce;
        SHA256 sha256;
        byte[] digest;

        // Timestamps.
        long t1, t2;

        // Task main program.
        public void main
        (String[] args)
The MineCoinTask class comes next. It is a nested class inside the outer MineCoinClu job class. (The task class doesn’t have to be a nested class; I wrote it that way for convenience.) Does the code in this class look familiar? It should; this class is virtually identical to the sequential program that mines one BitCoin, namely class MineCoinSeq from Chapter 7. The only thing I added was code to measure and print the task’s running time (lines 62 and 100). I want to know how long it took each task to find its own golden nonce.

Here are two separate runs of the MineCoinClu program. Each runs one coin mining task on one core of the tardis cluster.

```bash
$ java pj2 edu.rit.pj2.example.MineCoinClu 28 0123456789abcdef
Job 1396 launched Fri May 11 08:56:36 EDT 2018
Job 1396 started Fri May 11 08:56:37 EDT 2018
Coin ID = 0123456789abcdef
Nonce  = 0000000000c0ff47
Digest = 00000009cc107197f63d1bfb134d8a40f2f71ae911b56d54e57bc4c1e3329ca4
25015 msec
Job 1396 finished Fri May 11 08:57:02 EDT 2018 time 25434 msec
$ java pj2 edu.rit.pj2.example.MineCoinClu 28 3141592653589793
Job 1397 launched Fri May 11 08:58:53 EDT 2018
Job 1397 started Fri May 11 08:58:53 EDT 2018
Coin ID = 3141592653589793
Nonce  = 000000000020216d
Digest = 0000000746265312a0b2c8b834c69cf30c9823e44fb49c6d41260da97e87eb8f
4206 msec
Job 1397 finished Fri May 11 08:58:58 EDT 2018 time 4635 msec
```

When running a cluster job, the pj2 program automatically prints timestamps when the job launches, starts, and finishes. You can turn these and other debugging printouts on and off; refer to the pj2 Javadoc for more information.

Here are the same two coin IDs mined by a single run of the MineCoinClu program in parallel on two cores of the tardis cluster.

```bash
$ java pj2 edu.rit.pj2.example.MineCoinClu \     28 0123456789abcdef 3141592653589793
Job 1398 launched Fri May 11 08:59:56 EDT 2018
Job 1398 started Fri May 11 08:59:57 EDT 2018
Coin ID = 3141592653589793
Nonce  = 000000000020216d
Digest = 0000000746265312a0b2c8b834c69cf30c9823e44fb49c6d41260da97e87eb8f
4303 msec
Coin ID = 0123456789abcdef
Nonce  = 0000000000c0ff47
Digest = 00000009cc107197f63d1bfb134d8a40f2f71ae911b56d54e57bc4c1e3329ca4
25192 msec
Job 1398 finished Fri May 11 09:00:22 EDT 2018 time 25586 msec
```
throws Exception
{
    // Start timing.
    t1 = System.currentTimeMillis();
    // Parse command line arguments.
    coinId = Hex.toByteArray (args[0]);
    N = Integer.parseInt (args[1]);
    // Set up mask for leading zeroes.
    mask = ((1L << (64 - N)) - 1L);
    // Set up for computing hash digests.
    coinIdPlusNonce = new byte [coinId.length + 8];
    System.arraycopy (coinId, 0, coinIdPlusNonce, 0,
                    coinId.length);
    sha256 = new SHA256();
    digest = new byte [sha256.digestSize()];
    // Try all nonces until the digest has N leading zero bits.
    for (long nonce = 0L; nonce <= 0x7FFFFFFFFFFFFFFFL;
     ++ nonce)
    {
        // Test nonce.
        Packing.unpackLongBigEndian
            (nonce, coinIdPlusNonce, coinId.length);
        sha256.hash (coinIdPlusNonce);
        sha256.digest (digest);
        sha256.hash (digest);
        sha256.digest (digest);
        if ((Packing.packLongBigEndian (digest, 0) & mask) ==
            0L)
        {
            // Stop timing and print result.
            t2 = System.currentTimeMillis();
            System.out.printf ("Coin ID = %s%n",
                                Hex.toString (coinId));
            System.out.printf ("Nonce = %s%n",
                                Hex.toString (nonce));
            System.out.printf ("Digest = %s%n",
                                Hex.toString (digest));
            System.out.printf ("%d msec%n", t2 - t1);
            break;
        }
    }
    // The task requires one core.
    protected static in coresRequired()
    {
        return 1;
    }
}

Listing 18.1. MineCoinClu.java (part 2)
Note that the whole job finishes as soon as the longest-running task finishes, as we’d expect when the tasks are performed in parallel in separate cores.

Got more bitcoins? No problem!

```bash
$ java pj2 edu.rit.pj2.example.MineCoinClu \
  28 0123456789abcdef 3141592653589793 face2345abed6789 \
  0f1e2d3c4b5a6879

Job 1399 launched Fri May 11 09:01:31 EDT 2018
Job 1399 started Fri May 11 09:01:31 EDT 2018
Coin ID = 3141592653589793
Nonce   = 0000000000000020216d
Digest  = 0000000746265312a0b2c834c69cf30c9823e44fb49c6d41260
da97e87eb8f
4190 msec
Coin ID = 0123456789abcdef
Nonce   = 000000000000c0ff47
Digest  = 00000009cc107197f63d1bfb134d8a40f2f71ae911b56d54e57bc
4c1e3329ca4
25014 msec
Coin ID = 0f1e2d3c4b5a6879
Nonce   = 00000000001fe1c82
Digest  = 00000009d68870e4edd493f9aad0acea7d858605d3e086c282e7e8
4f4c821cb92
65789 msec
Coin ID = face2345abed6789
Nonce   = 00000000195365d1
Digest  = 000000091061e29a6e915cd9c4ddef6962c9de0fc253c6cca82bc
8e3152a8085
851057 msec
Job 1399 finished Fri May 11 09:15:42 EDT 2018 time 851487 msec
```

Under the Hood

When you run a Parallel Java 2 task on a multicore node, only one process is involved—the pj2 process—and the task’s parallel team threads all run within that process. But when you run a Parallel Java 2 job on a cluster, a whole bunch of processes running on multiple nodes have to get involved under the hood. These processes constitute the Parallel Java 2 middleware (Figure 18.2). The processes communicate with each other over the cluster’s backend network using TCP sockets.

As already mentioned, you kick off the job by logging into the cluster’s frontend node and running the pj2 program, creating the job process. After instantiating the Job subclass and calling its `main()` method to define the job’s rules, the job contacts the Tracker, a so-called “daemon” process that is always present on the frontend node. The job tells the Tracker that a new job has launched. Either immediately, or at a later time when resources are available, the Tracker tells the job to start. The job then requests the Tracker to run the job’s task or tasks. Each request includes the Task subclass to be exe-
cuted, the resources the task requires (such as the number of cores needed), and other information. The task requests go into a queue in the Tracker.

The Tracker is keeping track of which cores on which cluster nodes are busy or idle. As idle cores become available, the Tracker assigns each task to one of the cluster’s backend nodes. A Launcher daemon process is running on each backend node. For each task, the Tracker tells the Launcher to create a backend process on the Launcher’s node. The backend process runs a special Backend program that is part of the middleware. The backend contacts the job and obtains the Task subclass to be executed, the task’s command line arguments, and other information. The backend instantiates the Task subclass and calls the task’s main() method, passing in the command line arguments. The main() method then carries out the task’s computation. All this happens simultaneously for each of the job’s tasks. Each task ends up executing in its own process on one of the cluster’s backend nodes.

All the aforementioned processes remain in communication with each other as shown in Figure 18.2. Each backend periodically exchanges heart-beat messages with the job. If a heartbeat message fails to arrive when expected, the job (or the backend) detects that the backend (or the job) has failed, so the job (or the backend) terminates itself. Likewise, each Launcher
and each job exchanges heartbeat messages with the Tracker. If a Launcher fails, the Tracker no longer schedules tasks on that Launcher’s node. If a job fails, the tracker removes the job and its tasks from the Tracker’s queue. If the Tracker fails, the Launchers and the jobs terminate themselves.

Earlier we saw that a task can print stuff on `System.out`. (A task can also print on `System.err`.) These printouts appear on the console where the user typed the `pj2` command; that is, the console of the frontend process running on the frontend node. But the tasks are not running in the frontend process, or even on the frontend node! How can a task running in a backend process on a backend node cause printouts to appear on the frontend node’s console? The Parallel Java 2 middleware takes care of this automatically. As we have already seen, all the characters a task prints are stored in an internal buffer. In a single-node environment, when the task terminates (or when it calls `flush()`, the buffer’s contents are printed on the console. In a cluster environment, when the task terminates (or when it calls `flush()`), the middleware sends the buffer’s contents from the backend process to the frontend process, and then the frontend process prints the characters on its console (the job’s console). Furthermore, each task’s print buffer is printed as a unit, without being intermingled with any other task’s print buffer.

When a backend’s task finishes, the backend informs the job, and the backend process terminates. When all of the job’s tasks have finished, the job informs the Tracker, and the job process terminates. The Tracker then removes the job and its tasks from the Tracker’s queue.

All of the above is going on for multiple users logged into the cluster and running multiple jobs simultaneously. The Tracker keeps track of all the tasks in all the jobs and ensures that each task will run only when the needed resources are available. In this way, the tasks have full use of the cores to which they are assigned, ensuring that every job gets the maximum possible performance out of the cluster.

The Tracker has a web interface. Using any web browser, you can go to the Tracker’s URL and display the status of each node, core, job, and task in the cluster. The web interface provides status only; you can’t change the order of jobs in the queue, terminate jobs, and so on.

Although I’ve described the Parallel Java 2 middleware at some length, you don’t really need to be aware of the middleware when writing a cluster parallel program. Just define a Job subclass and Task subclasses as necessary. Then run the `pj2` program. The middleware does all its work behind the scenes.

One final detail: Depending on how the cluster is configured, you might need to use the `jar` option when you run the `pj2` program. This lets you provide a Java archive (JAR) containing the compiled class files for your program. The JAR gets sent to each backend process along with the other information about the task to be run. The middleware configures the backend
process’s Java class loader to look for classes in the JAR as well as in the usual Java class path in the file system. If the backend processes can’t find the program’s class files on the class path, and you omit the \texttt{jar} option, the backend processes won’t be able to find the class files needed to instantiate the tasks, and the job will fail. See the \texttt{pj2} Javadoc for further information.

**Points to Remember**

- In a massively parallel program, each computation is sequential, and all the computations run in parallel independently with no interaction.
- Write a Parallel Java 2 cluster program as a subclass of class \texttt{Job}.
- Put the job’s computational code in subclasses of class \texttt{Task}.
- Put code to define the job’s rules in the job’s \texttt{main()} method. The rules specify the tasks to be executed.
- Use the “\texttt{java pj2}” command to run your Parallel Java 2 cluster program. Use the \texttt{jar} option if necessary.