Chapter 19
Hybrid 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
The massively parallel bitcoin mining program in Chapter 18 doesn’t necessarily take full advantage of the cluster’s parallel processing capabilities. Suppose I run the program on the *tardis* cluster, which has 10 nodes with 12 cores per node, 120 cores total. Because the program mines each bitcoin sequentially on a single core, I have to mine 120 or more bitcoins to take full advantage of the cluster. If I mine fewer than 120 bitcoins, some of the cores will be idle. That’s not good. I want to put those idle cores to use.

I can achieve better utilization of the cores if I run the *multithreaded* bitcoin mining program on each node, rather than the sequential program. That way, I would have 12 cores working on each bitcoin, rather than just one core. This is a *hybrid* parallel program (Figure 19.1). I run one process on each node, each process mining a different bitcoin. Inside each process I run multiple threads, one thread on each core, each thread testing a different series of nonces on the same bitcoin. The program has two levels of parallelism. It is massively parallel (with no interaction) between the nodes, and it is multithreaded parallel within each node. Consequently, I would expect to see a speedup relative to the original cluster program.

Listing 19.1 gives the hybrid parallel MineCoinClu2 program. The outer Job subclass is the same as in the previous MineCoinClu program. The job’s main() method specifies a rule for each coin ID on the command line. Each rule runs an instance of the nested MineCoinTask defined later. The task’s command line argument strings are $N$, the number of most significant zero bits in the digest, and the coin ID.

Next comes the nested MineCoinTask, a subclass of class Task, that contains the code for one of the bitcoin mining tasks. Here is where this program differs from the previous program. This program’s task subclass is almost identical to the multithreaded parallel MineCoinSmp task from Chapter 7, rather than the single-threaded MineCoinSeq task. This program’s task has the same parallel for loop as MineCoinSmp, with the same leapfrog schedule that divides the work of testing the possible nonces among the threads of the parallel thread team, each thread running on its own core of the node.

The MineCoinTask subclass does not override the coresRequired() method. So by default an instance of the task will use all the cores on the node where the task is running. The Tracker takes the tasks’ required characteristics into account when scheduling tasks on the cluster. The Tracker will not execute a MineCoinTask instance until there is a node all of whose cores are idle.

I ran the hybrid parallel MineCoinClu2 program on the *tardis* cluster, giving it four coin IDs to mine. Here’s what it printed:

```
$ java pj2 edu.rit.pj2.example.MineCoinClu2 \
 28 0123456789abcdef 3141592653589793 face2345abed6789 \
 0f1e2d3c4b5a6879
Job 1403 launched Thu May 17 08:51:11 EDT 2018
```
Figure 19.1. Hybrid parallel program running on a cluster
Job 1403 started Thu May 17 08:51:11 EDT 2018
Coin ID = 3141592653589793
Nonce   = 000000000020216d
Digest  = 0000000746265312a0b2c8b834c69cf30c9823e44fb49c6d41260
da97e87eb8f
415 msec
Coin ID = 0123456789abcdef
Nonce   = 0000000000c0ff47
Digest  = 00000009cc107197f63d1bfb134d8a40f2f71ae911b56d54e57bc4c1e3329ca4
2261 msec
Coin ID = 0f1e2d3c4b5a6879
Nonce   = 0000000001fe1c82
Digest  = 0000000d68870e4edd493f9aad0acea7d858605d3e086c282e7e84f4c821cb92
5725 msec
Coin ID = face2345abed6789
Nonce   = 0000000001bc439ac
Digest  = 0000000d68870e4edd493f9aad0acea7d858605d3e086c282e7e84f4c821cb92
80010 msec
Job 1403 finished Thu May 17 08:52:31 EDT 2018 time 80408 msec

For comparison, here’s what the original MineCoinClu program printed, running on the tardis cluster with the same input:

$ 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   = 000000000020216d
Digest  = 0000000746265312a0b2c8b834c69cf30c9823e44fb49c6d41260
da97e87eb8f
4190 msec
Coin ID = 0123456789abcdef
Nonce   = 0000000000c0ff47
Digest  = 00000009cc107197f63d1bfb134d8a40f2f71ae911b56d54e57bc4c1e3329ca4
25014 msec
Coin ID = 0f1e2d3c4b5a6879
Nonce   = 0000000001fe1c82
Digest  = 0000000d68870e4edd493f9aad0acea7d858605d3e086c282e7e84f4c821cb92
65789 msec
Coin ID = face2345abed6789
Nonce   = 0000000001bc439ac
Digest  = 0000000d68870e4edd493f9aad0acea7d858605d3e086c282e7e84f4c821cb92
851057 msec
Job 1399 finished Fri May 11 09:15:42 EDT 2018 time 851487 msec
package edu.rit.pj2.example;
import edu.rit.crypto.SHA256;
import edu.rit.pj2.Job;
import edu.rit.pj2.LongLoop;
import edu.rit.pj2.Task;
import edu.rit.util.Hex;
import edu.rit.util.Packing;
public class MineCoinClu2
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.MineCoinClu2 <N> <coinid> " +
    "[<coinid> ...]");
    System.err.println("<N> = Number of leading zero bits " +
    "(1 .. 63)\n    System.err.println("<coinid> = Coin ID (hexadecimal)\n    terminate (1);
}

// Class MineCoinClu2.MineCoinTask provides the Task that
// computes one coin ID's nonce in the MineCoinClu2 program.
private static class MineCoinTask
extends Task
{
// Command line arguments.
byte[] coinId;
int N;

// Mask for leading zeroes.
long mask;

// Timestamps.
long t1, t2;

// Task main program.
public void main
(String[] args)
throws Exception
{
// Start timing.
t1 = System.currentTimeMillis();

Listing 19.1. MineCoinClu2.java (part 1)
Comparing the two printouts, we see that the hybrid parallel MineCoinClu2 program found exactly the same golden nonce as the original cluster parallel MineCoinClu program for three of the four coin IDs. For the fourth coin ID, the MineCoinClu2 program found a different golden nonce. Why? In the MineCoinClu program, only one thread was searching for a golden nonce for the fourth coin ID, and the one thread had to examine all the candidate nonces starting from 0. In the MineCoinClu2 program, there were 12 threads searching for a golden nonce in parallel. It so happened that one of the 12 threads, following the parallel loop’s leapfrog schedule, skipped ahead in the sequence of nonces and hit upon a different golden nonce sooner than the one thread in the MineCoinClu program did. (The MineCoinClu2 program was not so lucky with the other three coin IDs.)

The printouts also show that the hybrid parallel program finished more quickly. To be precise, the speedups were 10.096, 11.063, 11.492, and 10.637, respectively for the four coin IDs. This shows that each task was indeed utilizing all the cores of the node where the task was running.

**Under the Hood**

As mentioned previously, the Parallel Java 2 cluster middleware includes a Tracker daemon running on the cluster’s frontend node. The Tracker makes all the scheduling decisions for jobs and tasks running on the cluster. When a job executes one of its rules, the job sends a message to the Tracker, telling it to launch the rule’s task. The message includes the required characteristics of the node on which the task is to run. These characteristics are specified by overriding protected static methods in the task subclass. You can specify any or all of the following:

- **Override coresRequired()** to specify the number of CPU cores the task requires, either a specific number of cores, or ALL_CORES. If not overridden, the default is to use all the cores on the node where the task runs.
- **Override gpusRequired()** to specify the number of GPU accelerators the task requires, either a specific number of GPUs, or ALL_GPUS. If not overridden, the default is to require no GPUs. (We will study GPU accelerated parallel programming later in the book.)
- **Override nodeNameRequired()** to specify the name of the node on which the task must run, either a specific node name, or ANY_NODE_NAME. If not overridden, the default is to let the task run on any node regardless of the node name. (Later in the book, we will see cluster parallel programs whose tasks have to run on specific node names.)

You can also specify these characteristics in the rule that defines the task, by calling the appropriate methods on the TaskSpec. Refer to the Parallel Java 2 documentation for further information.
Chapter 19. Hybrid Parallel

Listing 19.1. MineCoinClu2.java (part 2)
The Tracker puts all launched tasks into a queue. There is one queue for each node in the cluster; tasks that require a specific node name go in the queue for that node. There is one additional queue for tasks that do not require a specific node name. The Tracker’s scheduling policy is first to start tasks from the node-specific queues on those nodes, then to start tasks from the non-node-specific queue on any available nodes, until the queue is empty or until the first task in the queue requires more resources (CPU cores, GPU accelerators) than are available. Whenever a task finishes and its resources go idle, the Tracker starts as many pending tasks as possible. To guarantee fair access to the cluster’s resources, the Tracker starts tasks from the queues in strict first-in-first-out (FIFO) order.

I have found the Tracker’s strict FIFO scheduling policy to be adequate for my teaching and research. I have not had the need, for example, to put priorities on tasks, so that higher-priority tasks can go ahead of lower-priority tasks in the queues. If your needs are different, you can write a modified version of the Tracker with a different scheduling policy; that’s why I’ve released the Parallel Java 2 Library under the GNU GPL free software license.

Points to Remember

- A hybrid parallel cluster program exhibits multiple levels of parallelism. It runs in multiple processes, one process per node of the cluster; each process runs multiple threads, one thread per core of the node.
- A task in a job can use the multithreaded parallel programming constructs, such as parallel for loops, to run on multiple cores in a node.
- If necessary, specify a task’s required characteristics—number of CPU cores, number of GPU accelerators, node name—by overriding the appropriate methods in the task subclass.