Chapter 16
Cluster Parallel Loops

- Part I. Preliminaries
- Part II. Tightly Coupled Multicore
- Part III. Loosely Coupled Cluster
  - Chapter 13. Massively Parallel
  - Chapter 14. Hybrid Parallel
  - Chapter 15. Tuple Space
  - Chapter 16. Cluster Parallel Loops
  - Chapter 17. Cluster Parallel Reduction
  - Chapter 18. Cluster Load Balancing
  - Chapter 19. File Output on a Cluster
  - Chapter 20. Interacting Tasks
  - Chapter 21. Cluster Heuristic Search
- Part IV. GPU Acceleration
- Part V. Big Data
The massively parallel bitcoin mining program in Chapter 14 still doesn’t take full advantage of the cluster’s parallel processing capabilities. Each bitcoin mining task uses all the cores on just one node. So on the 10-node tardis cluster, I have to mine 10 or more bitcoins to fully utilize all the cores. Furthermore, the most parallelism I can achieve for each bitcoin is the number of cores on a node, four cores in the case of tardis. What if I want to use every core in the cluster to mine a single bitcoin in parallel?

The multithreaded bitcoin mining program in Chapter 3 utilizes all the cores on one node by converting the plain sequential for loop over the nonces into a parallel for loop. Each parallel team thread, running on its own core in parallel with the other team threads, computes a subset of the nonces. I used a leapfrog schedule to partition the nonces among the threads. Running with, say, a team of four threads, thread 0 computes nonces 0, 4, 8, 12, . . . ; thread 1 computes nonces 1, 5, 9, 13, . . . ; and so on. When a thread finds the golden nonce, that thread tells the parallel for loop to stop, and then every thread exits the loop. All the machinery that creates the threads, that decides which threads would compute which nonces, that tells the threads to exit the loop, and so on is hidden inside the parallel for loop object. Because all the threads are in the same process, they all can access the loop control information, which is stored in hidden shared variables.

I can follow the same strategy to design a cluster parallel bitcoin mining program: I run a thread on each core of the cluster. Following the hybrid parallel programming pattern, the threads reside in tasks, one task on each node of the cluster. Then I have to partition the loop iterations among all the threads in all the tasks. Because the threads are not all in the same process, I can’t use a hidden shared variable to do the partitioning.

This brings us to the master-worker pattern (Figure 16.1) for parallel loops in a cluster parallel program. There is a master task and a number of worker tasks, each worker having a unique rank (0, 1, 2, and so on). The master task is in charge of the loop schedule; it partitions the loop indexes into chunks in some manner, such as one chunk per worker task (a fixed schedule), multiple equal-sized chunks (a dynamic schedule), or any of the other schedules. Each worker sends a message to the master requesting a chunk of work for that worker’s rank; the master sends a message back to the worker with the next available chunk for that rank. Each worker computes the designated chunk of work and puts the results in the appropriate place (the workers’ results are not shown in Figure 16.1). Whenever a worker finishes a chunk, the worker requests and obtains the next available chunk from the master. This continues until the master informs the worker that there are no more chunks, whereupon the worker terminates.

In a Parallel Java 2 program, which uses tuple space for inter-task communication, the master-worker pattern is modified slightly (Figure 16.2). The job’s main program partitions the loop iterations into chunks all at once and
puts a tuple containing each chunk into tuple space. Then, once the worker tasks start, each worker repeatedly takes a chunk tuple out of tuple space, computes the designated chunk, and puts the result somewhere. The worker does a conditional take so that, when all the chunks have been used up, the worker will detect that there is no more work.

I also need to stop all the threads in all the tasks whenever one thread finds the golden nonce. Because the threads are not all in the same process, I can’t use a hidden shared variable for this. Instead, I have to use tuple space. The thread that finds the golden nonce will inform the other threads by putting a special “stop tuple” into tuple space, where the other threads can see it.

Class Job and class Task provide methods to support the master-worker parallel loop pattern in a cluster parallel program, as well as the multi-
threaded parallel loop pattern in a multicore parallel program. Recall that in a multicore parallel program, you set up a parallel loop like this in the task class:

```java
public class MyTask extends Task {
    public void main (String[] args)
    {
        parallelFor (0, 999999) .exec (new Loop()
        {
            public void run (int i)
            {
                Loop body code for iteration i
            }
        });
    }
}
```

In a cluster parallel program, you set up a master-worker parallel loop like this; part is in the job class, the rest is in the task class:

```java
public class MyJob extends Job {
    public void main (String[] args)
    {
        masterFor (0, 999999, MyTask.class);
    }
}
public class MyTask extends Task {
    public void main (String[] args)
    {
        workerFor() .exec (new Loop()
        {
            public void run (int i)
            {
                Loop body code for iteration i
            }
        });
    }
}
```

In the job main program, the `masterFor()` method specifies the inclusive lower and upper bounds for the loop as well as the worker task class. Under the hood, this sets up a start rule to run one or more instances of the worker task class. The number of worker tasks is specified by the job's `workers` property; you can specify this property on the `pj2` command line or by calling the job's `workers()` method; if not specified, the default is one worker task. The `masterFor()` method also partitions the index range among the worker tasks and writes the appropriate chunk tuples into tuple space. The index partitioning is specified by the job's `masterSchedule` and `masterChunk`
package edu.rit.pj2.example;
import edu.rit.crypto.SHA256;
import edu.rit.io.InStream;
import edu.rit.io.OutStream;
import edu.rit.pj2.EmptyTuple;
import edu.rit.pj2.Job;
import edu.rit.pj2.LongLoop;
import edu.rit.pj2.Section;
import edu.rit.pj2.Task;
import edu.rit.util.Hex;
import edu.rit.util.Packing;

public class MineCoinClu3
    extends Job
{
    // Job main program.
    public void main(String[] args)
    {
        // Parse command line arguments.
        if (args.length != 2) usage();
        String coinId = args[0];
        int N = Integer.parseInt(args[1]);
        if (1 > N || N > 63) usage();

        // Set up master-worker cluster parallel for loop.
        masterSchedule(leapfrog);
        masterFor(0L, 0x7FFFFFFFFFFFFFFFL, WorkerTask.class)
            .args(coinId, ""+N);
    }

    // Print a usage message and exit.
    private static void usage()
    {
        System.err.println("Usage: java pj2 [workers=<K>] " +
            "edu.rit.pj2.example.MineCoinClu3 <coinId> <N>"");
        System.err.println("<K> = Number of worker tasks " +
            "(default 1)");
        System.err.println("<coinId> = Coin ID (hexadecimal)"");
        System.err.println("<N> = Number of leading zero bits " +
            "(1 .. 63)"");
        throw new IllegalArgumentException();
    }

    // Class MineCoinClu3.WorkerTask provides the worker Task in the
    // MineCoinClu3 program. The worker tasks perform the golden
    // nonce search.
    private static class WorkerTask
        extends Task
    {
        // Command line arguments.
        byte[] coinId;
        int N;

        // Mask for leading zeroes.
        long mask;

        // For early loop exit.
        boolean stop;
    
    
Listing 16.1. MineCoinClu3.java (part 1)
properties; you can specify these on the pj2 command line or by calling the job’s masterSchedule() and masterChunk() methods; if not specified, the default is a fixed schedule.

In the worker task class main program, the workerFor() method creates a parallel loop object that executes the given loop body (a subclass of class Loop). Under the hood, the parallel loop object takes chunk tuples for the worker from tuple space and, for each index in each chunk, calls the given Loop object’s run() method.

Like all parallel loops, the workerFor() parallel loop creates a team of threads, one thread per core by default, and executes the loop body in parallel within the worker task. A program with a master-worker parallel loop is thus automatically a hybrid parallel program; it consists of multiple worker tasks running on multiple nodes; within each node, it consists of multiple threads running on multiple cores.

Listing 16.1 gives the third version of the cluster parallel bitcoin mining program, class MineCoinClu3. The job main program first parses the command line arguments (lines 20–23): the coin ID and N, the number of most significant zero bits in the digest.

The job main program sets up the master portion of the master-worker cluster parallel for loop. Calling the masterSchedule(1eapfrog) method on line 26 tells the master to partition the loop iterations for the workers using a leapfrog schedule, as I did in the multicore parallel program in Chapter 3. Calling the masterFor() method on line 27 does several things: it specifies the inclusive lower and upper bounds of the loop index range; it partitions the loop index range into chunks according to the specified leapfrog schedule; it puts one chunk tuple into tuple space for each chunk; and it adds a start rule to the job to fire up a certain number of worker tasks. Each task will be an instance of class WorkerTask (defined later). Each task’s command line arguments are the same, namely the coin ID and N (line 28).

In the previous MineCoinClu and MineCoinClu2 programs, the job main program created multiple rules with a single task each. Here, the master in the job main program creates a single rule with multiple tasks. A single rule with multiple tasks defines a task group. Each task in a task group has a unique rank that serves to distinguish the tasks in the task group from one another. In a task group with K tasks, the task ranks are 0 through K – 1.

What’s the difference between a task group and a bunch of single tasks? The Tracker treats the two cases differently when it schedules tasks to run on nodes. When there is a bunch of single tasks, the Tracker schedules the tasks independently. Depending on when computational resources become available, some of the tasks might start right away, and others might sit in the Tracker’s queue for a while. That’s fine if the tasks don’t interact with each other. On the other hand, when there is a task group, the Tracker schedules all the tasks in the group as a unit. The Tracker will not start any of the tasks in
// Task main program.
public void main
(String[] args)
throws Exception
{
    // 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;

    // Do computation and termination check in parallel.
    parallelDo (new Section()
    {
        public void run() throws Exception
        {
            // Computation section.
            // Try all nonces in the chunk until the digest has N
            // leading zero bits or until another worker has
            // found the solution.
            workerFor().schedule (leapfrog)
            .exec (new LongLoop()
            {
                byte[] coinIdPlusNonce;
                SHA256 sha256;
                byte[] digest;

                public void start()
                {
                    coinIdPlusNonce = new byte [coinId.length + 8];
                    System.arraycopy (coinId, 0, coinIdPlusNonce, 0, coinId.length);
                    sha256 = new SHA256();
                    digest = new byte [sha256.digestSize()];
                }

                public void run (long nonce) throws Exception
                {
                    if (stop) stop();
                    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)
                    {
                        putTuple (new EmptyTuple());
                        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));
                        stop();
                    }
                }
            });
        }
    });
}
the group until there are enough resources to run all the tasks in the group. This is critical if the tasks have to interact with each other. In the MineCoin-Clu3 program, the tasks do interact with each other—namely, one task informs the other tasks when it finds the golden nonce, so the other tasks can stop—and so the tasks have to be specified as a task group.

Next comes the code for the worker task (line 47). Let’s think about how all the tasks discover that one of the tasks found the golden nonce. This involves inter-task communication, which has to go through tuple space. As mentioned previously, when a task finds the golden nonce, the task can put a special “stop tuple” into tuple space. Each task can be calling the `readTuple()` method with a stop tuple as the template. The `readTuple()` method blocks until the stop tuple shows up, then the `readTuple()` method returns; this signals that the golden nonce was found. We read the stop tuple, rather than take the stop tuple, so that the stop tuple remains in tuple space for all the tasks to read.

There’s a problem, though. The `readTuple()` method blocks the calling thread until the stop tuple shows up. This means the calling thread can’t be doing other work while blocked inside the `readTuple()` method. That’s not what I want. While the task is waiting for the stop tuple, I also want the task to be computing a series of nonces. This means I need another thread to carry out the computations. But we know how to set up two threads doing different things: that’s the parallel sections pattern.

Returning to the code, we see the two sections in a parallel do statement starting at line 73: the computation section (lines 77–121) that is computing the nonces, and the termination check section (lines 122–130) that is waiting for the stop tuple. When the stop tuple appears in tuple space, the termination check section sets the shared global `stop` flag (line 58) to true; the computation section polls this flag and exits its loop when the flag becomes true.

The computation section does the worker portion of the master-worker cluster parallel for loop. Calling the `workerFor()` method on line 81 creates a worker parallel for loop object. This is similar to the parallel for loop objects we studied earlier, except the loop index range is not specified. Instead, the worker parallel for loop gets chunks of loop indexes by taking chunk tuples out of tuple space. The worker parallel for loop uses a leapfrog schedule (line 81) to further subdivide each chunk among the parallel team threads. Each team thread executes its own copy of the loop body, an instance of an anonymous inner subclass of class `LongLoop` (line 82). Each team thread calls its own loop body’s `start()` method (lines 88–95) once before commencing the loop iterations, and it calls the `run()` method (lines 97–118) to perform each loop iteration, passing in the loop index (the nonce to be tested). The task stays in the worker parallel for loop until the nonce hits the largest possible value (which it almost surely never will) or until the shared global `stop` flag is set to true by the termination check section (line 99).
If the task finds the golden nonce (lines 106–117), the task puts a stop tuple into tuple space to inform the other tasks. The stop tuple (line 109) is an instance of class edu.rit.pj2.EmptyTuple. This is just a tuple subclass with no content; its mere presence in tuple space is enough to signal termination. The task also does an early loop exit after printing the results (line 116). Whenever one task finds the golden nonce, all the tasks exit their loops, the tasks terminate, and the job terminates.

To sum up, the MineCoinClu3 program is a hybrid parallel program. The master partitions the loop index range among the worker tasks, using a leapfrog schedule. Each worker task’s worker parallel for loop further subdivides each chunk of the loop index range among the threads in the task, again using a leapfrog schedule. All this happens automatically under the hood in the masterFor() and workerFor() methods.

I ran the MineCoinSeq program on one node of the tardis cluster, and I ran the MineCoinClu3 program with one to ten nodes (worker tasks), using strong scaling, for coin ID = fedcba9876543210, and for $N = 28$ and $29$ most significant zero bits. Here are the commands I used:

```
$ java pj2 debug=makespan \n   edu.rit.pj2.example.MineCoinSeq fedcba9876543210 28
Coin ID = fedcba9876543210
Nonce   = 0000000006ee7a3e
Digest  = 000000084be04f3b2d2aa095debebf6a84241e4048b5cc90ee9708b0fc74086
Job 3 makespan 233904 msec
```

```
$ java pj2 debug=makespan workers=2 \n   edu.rit.pj2.example.MineCoinClu3 fedcba9876543210 28
Coin ID = fedcba9876543210
```

Listing 16.1. MineCoinClu3.java (part 3)
Nonce   = 0000000006ee7a3e
Digest  = 000000084be04f3b20d2aa095debebf6a84241e4048b5cc90ee970b0fc7408
Job 10 makespan 29813 msec

For the cluster parallel program, the \textit{workers} parameter specifies the number of worker tasks \(K\). Omitting the \textit{threads} parameter causes each worker task to use one thread for each core of the node on which it is running, so the cluster parallel program ran with 4, 8, 12, \ldots, 40 cores on \texttt{tardis}. Figure 16.3 plots the running times and efficiencies I observed. The running time model is

\[ T = (5.67 \times 10^{-3} + 5.82 \times 10^{-4} N)K + (0.564 + 1.98 \times 10^{-6} N) \div K \quad (16.1) \]

The program exhibits excellent strong scaling efficiencies all the way out to 40 cores.

\section*{Under the Hood}

In a master-worker cluster parallel for loop, the master puts a number of chunk tuples (class \texttt{edu.rit.pj2.Chunk} or class \texttt{edu.rit.pj2.LongChunk}) into tuple space. Each chunk has the following fields:

- \texttt{rank} — Rank of the worker task that will execute the chunk, or \texttt{ANY} if any worker task can execute the chunk.
- \texttt{lb} — Loop index lower bound.
- \texttt{ub} — Loop index upper bound.
- \texttt{stride} — Loop index stride (the amount to increase the loop index on each iteration).

The \texttt{Chunk} and \texttt{LongChunk} classes’ \texttt{matchContent()} methods are defined so that a template chunk will match a target chunk if the template’s rank is \texttt{ANY}, if the target’s rank is \texttt{ANY}, or if the template’s rank equals the target’s rank. The other fields in the chunk don’t matter for purposes of matching.

The \texttt{MineCoinClu3} program’s loop index goes from 0 to \texttt{7FFFFFFFFFFFFFFFFFF} hexadecimal, partitioned using a leapfrog schedule. Suppose there are two worker tasks; then the master puts these two chunk tuples into tuple space:

\begin{center}
\begin{tabular}{|c|c|}
\hline
\texttt{LongChunk} & \texttt{LongChunk} \\
\hline
\texttt{rank} = 0  & \texttt{rank} = 1 \\
\texttt{lb} = 0    & \texttt{lb} = 1 \\
\texttt{ub} = \texttt{7FFFFFFFFFFFFFF} & \texttt{ub} = \texttt{7FFFFFFFFFFFFFF} \\
\texttt{stride} = 2 & \texttt{stride} = 2 \\
\hline
\end{tabular}
\end{center}

Worker task 0’s worker parallel for loop takes a \texttt{LongChunk} tuple out of tuple space, using a template with \texttt{rank} set to 0. This template matches the
Chapter 16. Cluster Parallel Loops

first of the two chunk tuples. This chunk tells worker task 0 to do loop indexes 0, 2, 4, … 7FFFFFFF. Now suppose the worker parallel for loop has four threads. This chunk is further subdivided among the threads using a leapfrog schedule. Thread 0 does loop indexes 0, 8, 16, …; thread 1 does loop indexes 2, 10, 18, …; thread 2 does loop indexes 4, 12, 20, …; thread 3 does loop indexes 6, 14, 22, ….

Worker task 1’s worker parallel for loop takes a LongChunk tuple out of tuple space, using a template with rank set to 1. This template matches the second of the two chunk tuples. This chunk tells worker task 1 to do loop indexes 1, 3, 5, … 7FFFFFFF. Worker task 1’s worker parallel for loop’s thread 0 does loop indexes 1, 9, 17, …; thread 1 does loop indexes 3, 11, 19, …; thread 2 does loop indexes 5, 13, 21, …; thread 3 does loop indexes 7, 15, 23, … . In this way, all the loop indexes are covered in a leapfrog fashion among all the threads in all the worker tasks.

Points to Remember

- When multiple tasks need to interact with each other, specify them as a task group with a single rule.
- To get a work sharing parallel loop across a cluster parallel program, use the master-worker pattern.
- In the job’s main() method, code the master portion by calling the masterFor() method.
- In the worker task’s main() method, code the worker portion by calling the workerFor() method.
• To get a global “flag” shared among the tasks of a cluster parallel program, use tuple space. Set the flag by putting a tuple, get the flag by reading a tuple.