Chapter 21
Cluster Parallel Loops

► 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 19 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, 12 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 7 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 12 threads, thread 0 computes nonces 0, 12, 24, 36, . . . ; thread 1 computes nonces 1, 13, 25, 37, . . . ; and so on. When a thread finds a 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 21.1) 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 21.1). 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 21.2). 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.

Class Job and class Task provide methods to support the master-worker parallel loop pattern in a cluster parallel program, as well as the multithreaded 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:
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:

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 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.
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.pj2.TupleListener;
import edu.rit.pj2.tuple.EmptyTuple;
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)\n        System.err.println("<coinId> = Coin ID (hexadecimal)\n        System.err.println("<N> = Number of leading zero bits " +
            "(1 .. 63)\n        terminate (1);
    }

    // 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.
        volatile boolean stop;
    }

Listing 21.1. MineCoinClu3.java (part 1)
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. For further information about how a master-worker parallel loop partitions the loop iterations among the tasks in the job and the threads in the tasks, see the “Under the Hood” section below.

In the cluster parallel bitcoin mining program, 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 first thread that finds a golden nonce will inform the other threads by putting a special “stop tuple” into tuple space, where the other threads can see it.

Listing 21.1 gives the third version of the cluster parallel bitcoin mining program, class `MineCoinClu3`. The job main program first parses the command line arguments (lines 18–21): 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(leapfrog)` method on line 24 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 7. Calling the `masterFor()` method on line 25 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 26).

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
// 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);

    // Early loop exit when any task finds the golden nonce.
    addTupleListener(new TupleListener<EmptyTuple>(new EmptyTuple())
    {
        public void run(EmptyTuple tuple)
        {
            stop = true;
        }
    });

    // Try all nonces until the digest has N leading zero bits.
    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));
            }
        }
    });
}

Listing 21.1. MineCoinClu3.java (part 2)
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 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 a 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 45). Let’s think about how all the tasks discover that one of the tasks found a golden nonce. This involves inter-task communication, which has to go through tuple space. As mentioned previously, when a task finds a golden nonce, the task can put a special “stop tuple” into tuple space, and the other tasks can detect this by reading the stop tuple. Each task reads the stop tuple, rather than taking the stop tuple, so that the stop tuple remains in tuple space for the other tasks to read.

However, while waiting for the stop tuple to show up, I want the task to be computing a series of nonces. This means the task cannot block trying to read the stop tuple. This is exactly the situation for which tuple listeners are designed. As its first act, the task can set up a tuple listener, specifying that it wants to read a stop tuple. The task can then proceed to search for a golden nonce. Later, if and when a stop tuple shows up, the tuple listener can cause the golden nonce search to stop.

What shall the stop tuple subclass look like? The stop tuple carries no information; its mere presence in tuple space is enough to signal that a golden nonce was found. So the stop tuple subclass will have no fields. The Parallel Java 2 Library provides just such a tuple subclass, namely class edu.rit.pj2.tuple.EmptyTuple. I will use that for my stop tuple.

Returning to the code, the task adds a tuple listener for the stop tuple (lines 71–78). The tuple listener constructor’s argument specifies the template to match, namely an EmptyTuple (line 72). Because a tuple operation was not specified, the default operation is “read.” The tuple listener’s run() method (line 74), specified in an anonymous inner class, is called when an EmptyTuple is read from tuple space. The run() method sets the shared global stop flag to true. Further down in the code, the golden nonce search polls this flag and exits its loop when the flag becomes true, that is, when an EmptyTuple appears.

Note that the shared global stop flag is declared with the volatile keyword (line 56). Refer back to Chapter 17 to refresh your memory about why this flag needs to be declared volatile.
Next comes 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 87–94) once before commencing the loop iterations, and it calls the `run()` method (lines 96–117) 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 tuple listener (line 98).

If the task finds a golden nonce (lines 105–116), the task puts a stop tuple into tuple space to inform the other tasks (line 108). The stop tuple is an instance of class `EmptyTuple`. The task also does an early loop exit after printing the results (line 115). Whenever one task finds a 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 \( n = 29 \) most significant zero bits. Here are examples of the commands I used:

```
$ java pj2 debug=makespan \n   edu.rit.pj2.example.MineCoinSeq fedcba9876543210 28
Coin ID = fedcba9876543210
```
Nonce   = 0000000006ee7a3e
Digest  = 000000084be04f3b20d2aa095debebf6a84241e4048b5cc90ee97
08b0fc74086
Job 1406 makespan 228866 msec
$ java pj2 debug=makespan workers=2 \n   edu.rit.pj2.example.MineCoinClu3 fedcba9876543210 28
Coin ID = fedcba9876543210
Nonce   = 0000000006ee7a3e
Digest  = 000000084be04f3b20d2aa095debebf6a84241e4048b5cc90ee97
08b0fc74086
Job 1415 makespan 9876 msec

For the cluster parallel program, the workers parameter specifies the number
of worker tasks $K$. Omitting the 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 12, 24, 36, \ldots 120 cores on tardis. Figure
21.3 plots the running times, speedups, and efficiencies I observed.

To fit a running time model to the data, I need to know the problem size
$N$ for the two cases. $N$ is equal to the number of candidate nonces the pro-
gram examined. This is the golden nonce that was reported, plus 1. From the
program outputs, $N = 116,292,159$ and $605,491,489$ for $n = 28$ and 29, re-
spectively. The running time model is

$$T = (0.138 + 6.29 \times 10^{-10} N) + (2.00 \times 10^{-6} N) / K. \quad (21.1)$$

The sequential fractions are $F = 0.000907$ for 28 most significant zero bits,
$F = 0.000428$ for 29 most significant bits. Despite these very low sequential
fractions, the large numbers of cores result in noticeable drops in efficiency.
Still, the program exhibits strong scaling efficiencies of about 0.9 or higher
all the way out to 120 cores.

**Under the Hood**

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

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

The Chunk and LongChunk classes’ matchContent() methods are defined
so that a template chunk will match a target chunk if the template’s rank is
ANY, if the target’s rank is 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.
Figure 21.3. MineCoinClu3 strong scaling performance metrics
Figure 21.4 shows a master-worker parallel for loop, with indexes going from 0 to 9999, with the default fixed schedule, running on a cluster with four nodes, each node having four cores. The `masterFor()` statement in the job’s `main()` method sets up a start rule with four worker tasks, one on each node. The `masterFor()` statement then puts four chunk tuples into tuple space, partitioning the index range into four equal chunks in accordance with the fixed schedule: indexes 0–2499 for worker task rank 0, indexes 2500–4999 for worker 1, indexes 5000–7499 for worker 2, and indexes 7500–9999 for worker 3. When the job’s `main()` method returns, the job commences and the worker tasks start running. In each worker task, the `workerFor()` statement sets up a thread team with four threads. Next, the `workerFor()` statement takes a chunk tuple out of tuple space, using a template whose `rank` field equals the task’s own rank. The `workerFor()` statement then subdivides the loop index range from that chunk tuple into four equal subranges in accordance with the fixed schedule. In worker task 0, whose index range is 0–2499, thread rank 0 does indexes 0–624, thread 1 does indexes 625–1249,
thread 2 does indexes 1250–1874, and thread 3 does indexes 1875–2499. The other worker tasks do the same with their own chunk tuples. In this way, every loop index from 0 through 9999 is performed by one of the team threads in one of the worker tasks.

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

Worker task 0’s worker parallel for loop takes a LongChunk tuple out of tuple space, using a template with rank set to 0. This template matches the first of the two chunk tuples. This chunk tells worker task 0 to do loop indexes 0, 2, 4, . . . 7FFFFFFFFFFFFFF. (This time the index goes up by a stride of 2 at each iteration.) 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, . . . 7FFFFFFFFFFFFFF. 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.

It’s important to realize that the masterFor() statement in the job’s main() method sets up the worker tasks and partitions the loop index range once and only once, at the start of the job. Thus, in a cluster parallel program, the master-worker pattern can be used only if the parallel loop is the outermost loop in the program. If the outermost loop needs to be a regular sequential loop—if it has sequential dependencies, for example—then you can’t use the master-worker pattern, and you have to organize the loops differently. (We will see an example of such a program in Chapter 25.)

Let’s consider how a task, running in a backend process as part of a job, communicates with tuple space. When a task puts a tuple, the task sends a put message to the job’s frontend process over the cluster’s backend network.
The message consists of the tuple’s contents. The code for sending the message is executed by the task thread that calls the putTuple() method.

Receiving a tuple works as follows. When a task takes or reads a tuple, the task sends a take or read request message to the frontend process. The message contains the template for the take or read. The code for sending the message is executed, again, by the task thread that calls the takeTuple() or readTuple() method. After that, however, the task thread blocks on an internal data structure of pending tuple requests, waiting for the frontend to send a response.

The backend process has a separate internal thread that reads incoming messages from the frontend process. The incoming messages are read and processed one at a time. When the frontend process finds a tuple that matches a previously requested template, the frontend process sends a response message to the backend process; the response message contains the matching tuple. The backend process’s internal thread receives this message, puts the matching tuple into the internal data structure of pending tuple requests, and unblocks the task thread. The task thread retrieves the matching tuple and returns it from the takeTuple() or readTuple() method call.

A tuple listener works differently. When a task thread in the backend process calls the addTupleListener() method, the task stores the tuple listener object in an internal list, and the task sends a take or read request message to the frontend process as before. The addTupleListener() method then returns, allowing the task thread to proceed. Later, when the frontend process sends a message containing a tuple that matches the tuple listener’s template, the internal message receiving thread reads this incoming message and then calls the tuple listener’s run() method, passing in the matching tuple.

Consequently, the tuple listener’s run() method must not do any lengthy processing. Rather, the tuple listener should just record the matching tuple or set a flag, as the bitcoin mining worker task does, and return right away; then the task thread can process the tuple matched event. Alternatively, the tuple listener could spawn a separate thread to process the event. Because the internal message receiving thread will not receive and process the next message until the run() method returns, doing lengthy processing in the run() method will interfere with the cluster middleware’s operation.

**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.
• Use the master-worker pattern only if the parallel loop is the outermost loop in the program.
• 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.
• You can specify the parallel loop schedule and chunk size separately for the master and for the workers. The default is a fixed schedule.
• 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, possibly using a tuple listener to read the tuple.
• Do not do any lengthy processing in a tuple listener’s `run()` method.