Chapter 22
Cluster Parallel Reduction

► 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
Let’s look at how to do reduction in a cluster parallel program. Recall that in a single-node multicore parallel program that does reduction, each thread computes a partial result, then the partial results are combined into one final result. Now extend that to a multi-node multicore, or hybrid, parallel program, consisting of multiple worker tasks each with multiple threads (Figure 22.1). Just as there are two levels of parallelism—intra-task parallelism among the threads in each task, and inter-task parallelism among the tasks in the job—there are also two levels of reduction. Each thread in each task computes a partial result ($R_{00}$, $R_{01}$, and so on); the partial results in each task are combined into one semifinal result ($R_0$, $R_1$, $R_2$, $R_3$); the semifinal results are combined into one final result $R$.

In the single-node parallel program with reduction, the threads each put their results into shared memory, and the results are combined using a reduction tree (Figure 8.3) in which various threads read various other threads’ results from shared memory. This still works for the first level of reduction in a hybrid parallel program. But this no longer works for the second level of reduction. Running on separate nodes, the tasks do not share memory, and inter-task communication is needed to transfer one task’s semifinal result into another task in order to do the reduction. In a Parallel Java 2 program, inter-task communication goes through tuple space. Therefore, in Figure 22.1 each task records its semifinal result in a tuple and puts the tuple into tuple space.

Like the Bitcoin mining program in Chapter 21, the worker tasks in Figure 22.1 are specified by a start rule in the job’s main() method, and they start executing when the job commences. (The master portion of the master-worker cluster parallel for loop sets up this start rule.) An additional reduction task is specified by a finish rule; this task runs once all the worker tasks have finished. The reduction task takes the tuples with the semifinal results produced by the worker tasks, combines the semifinal results, and calculates the final result.

Why make this a hybrid parallel program? Why not have just one level of parallelism and one level of reduction? Then each task would have a single thread, and each task would put its semifinal result directly into tuple space, without needing the intra-task reduction. The answer has to do with the fact that any tuple space operation, such as putting a tuple, involves sending a message from one process to another. Each message takes a nonzero amount of time to traverse the cluster network. Because network speeds are relatively much slower than CPU speeds, in a cluster parallel program it’s important to minimize both the number of messages and the size of each message. I minimize the number of messages (putTuple() operations) by doing the first level of reduction inside each task, in parallel with the other tasks. Then I only have to put as many semifinal result tuples as there are tasks, not as many tuples as there are threads—in Figure 22.1, four messages instead of 16.
Figure 22.1. Cluster parallel reduction
Listing 22.1 gives the source code for PiClu, a cluster parallel version of the $\pi$ estimating program from Chapter 8. It follows the pattern of Figure 22.1. The job’s `main()` method specifies the master portion of a master-worker cluster parallel for loop (line 20). By default, the master partitions the loop iterations using a fixed schedule, and the worker tasks further subdivide each chunk of iterations using a fixed schedule. Because the loop body’s running time is the same for each iteration, the load is inherently balanced, and a fixed schedule is appropriate. The `main()` method also specifies a finish rule that sets up the reduction task (lines 23–24). Calling the `runInJobProcess()` method specifies that the reduction task will run in the job process on the frontend node, rather than in a separate process on a backend node. The reduction task does not do any heavy computation, and there’s no need to tie up a backend node to run the reduction task.

Class WorkerTask (lines 39–79) defines the worker tasks’ computation. We’ve seen this code before; it is almost the same as the multicore PiSmp program in Chapter 8. However, this task specifies the worker portion of a master-worker cluster parallel for loop (line 59). Like the PiSmp program, class WorkerTask has a global reduction variable of type LongVbl.Sum, and the threads have thread-local reduction variables that are automatically reduced into the global variable.

The threads also have thread-local pseudorandom number generators (PRNGs). This time, however, the PRNG in each thread in each task must be initialized with a different seed, otherwise the tasks will generate the same sequence of random numbers. I do this on line 65 with the expression

$$\text{seed} + 1000 \times \text{taskRank()} + \text{rank()}$$

`taskRank()` returns the current task’s rank within the task group, and `rank()` returns the current thread’s rank within the task. If the program is run with four worker tasks each with four threads, task 0 initializes its threads’ PRNGs with `seed + 0`, `seed + 1`, `seed + 2`, and `seed + 3`; task 1 initializes its threads’ PRNGs with `seed + 1000`, `seed + 1001`, `seed + 1002`, and `seed + 1003`; task 2 initializes its threads’ PRNGs with `seed + 2000`, `seed + 2001`, `seed + 2002`, and `seed + 2003`; and task 3 initializes its threads’ PRNGs with `seed + 3000`, `seed + 3001`, `seed + 3002`, and `seed + 3003`. Every thread in every task gets a different seed.

Finally, instead of printing the answer, the worker task puts the global reduction variable containing the task’s semifinal result into tuple space (line 77). Class LongVbl, as well as all the reduction variable classes in package edu.rit.pj2.vbl, are in fact tuple subclasses—they all extend class Tuple—so putting them into tuple space works. (Fancy that!)

Class ReduceTask (lines 83–100) defines the reduction task, which runs when all the worker tasks have finished. It has its own reduction variable, `count`; it calls the `tryToTakeTuple()` method repeatedly to obtain the
package edu.rit.pj2.example;
import edu.rit.pj2.Job;
import edu.rit.pj2.LongLoop;
import edu.rit.pj2.Task;
import edu.rit.pj2.vbl.LongVbl;
import edu.rit.util.Random;
public class PiClu
    extends Job
{
    // Job main program.
    public void main
        (String[] args)
    {
        // Parse command line arguments.
        if (args.length != 2) usage();
        long seed = Long.parseLong (args[0]);
        long N = Long.parseLong (args[1]);

        // Set up a task group of K worker tasks.
        masterFor (0, N – 1, WorkerTask.class) .args (""+seed);

        // Set up reduction task.
        Rule() .atFinish() .task (ReduceTask.class) .args (""+N)
            .runInJobProcess();
    }

    // Print a usage message and exit.
    private static void usage()
    {
        System.err.println ("Usage: java pj2 " +
            "edu.rit.pj2.example.PiClu <seed> <N>");
        System.err.println ("<seed> = Random seed");
        System.err.println ("<N> = Number of random points");
        terminate (1);
    }

    // Class PiClu.WorkerTask performs part of the computation for
    // the PiClu program.
    private static class WorkerTask
        extends Task
    {
        // Command line arguments.
        long seed;

        // Number of points within the unit circle.
        LongVbl count;

        // Main program.
        public void main
            (String[] args)
                throws Exception
        {
            // Parse command line arguments.
            seed = Long.parseLong (args[0]);
        }
    }

Listing 22.1.  PiClu.java (part 1)
worker tasks’ semifinal result tuples; and it accumulates those into the count variable using the reduce() method, which does a sum-reduce. When the tryToTakeTuple() method returns null, the reduction task knows that it has accumulated all the semifinal result tuples, so it prints the final answer. Because the reduction task terminates when there are no more semifinal result tuples to take, the reduction task cannot begin until after all the worker tasks have put their result tuples and have terminated. That’s why the reduction task was configured with a finish rule in the job main program (line 23). If the reduction task were to start at the same time as the worker tasks, there would be no result tuples in tuple space yet, the tryToTakeTuple() method would return null right away, and the reduction task would print the wrong value (0) as the estimate for \( \pi \).

Let’s consider how the hybrid parallel PiClu program partitions the parallel loop iterations. Suppose I run the program with four worker tasks on a four-node cluster, each node having four cores, with \( N = 1,000,000 \) iterations. The job main program does not specify the master schedule, so the master partitions the loop index range 0 to 999,999 among the workers using the default fixed schedule, with an equal-sized chunk of indexes for each worker:

- Worker 0 — indexes 0 to 249,999
- Worker 1 — indexes 250,000 to 499,999
- Worker 2 — indexes 500,000 to 749,999
- Worker 3 — indexes 750,000 to 999,999

The worker task main program does not specify the parallel for loop schedule, so each worker partitions its own index chunk among the four parallel team threads using the default fixed schedule, with an equal-sized chunk of indexes for each team thread:

- Worker 0 — indexes 0 to 249,999
  - Thread 0 — indexes 0 to 62,499
  - Thread 1 — indexes 62,500 to 124,999
  - Thread 2 — indexes 125,000 to 187,499
  - Thread 3 — indexes 187,500 to 249,999
- Worker 1 — indexes 250,000 to 499,999
  - Thread 0 — indexes 250,000 to 312,499
  - Thread 1 — indexes 312,500 to 374,999
  - Thread 2 — indexes 375,000 to 437,499
  - Thread 3 — indexes 437,500 to 499,999
- Worker 2 — indexes 500,000 to 749,999
  - Thread 0 — indexes 500,000 to 562,499
  - Thread 1 — indexes 562,500 to 624,999
  - Thread 2 — indexes 625,000 to 687,499
  - Thread 3 — indexes 687,500 to 749,999
```
// Generate n random points in the unit square, count how
// many are in the unit circle.
count = new LongVbl.Sum (0);
workerFor() .exec (new LongLoop()
{
  Random prng;
  LongVbl thrCount;
  public void start()
  {
    prng = new Random (seed + 1000*taskRank() + rank());
    thrCount = threadLocal (count);
  }
  public void run (long i)
  {
    double x = prng.nextDouble();
    double y = prng.nextDouble();
    if (x*x + y*y <= 1.0) ++ thrCount.item;
  }
});
// Report results.
putTuple (count);
}

// Class PiClu.ReduceTask combines the worker tasks' results and
// prints the overall result for the PiClu program.
private static class ReduceTask
extends Task
{
// Reduce task main program.
public void main
(String[] args)
throws Exception
{
  long N = Long.parseLong (args[0]);
  LongVbl count = new LongVbl.Sum (0L);
  LongVbl template = new LongVbl();
  LongVbl taskCount;
  while ((taskCount = tryToTakeTuple (template)) != null)
    count.reduce (taskCount);
  System.out.printf ("pi = 4*%d/%d = %.9f%n", count.item, N, 4.0*count.item/N);
}
}
```

Listing 22.1. PiClu.java (part 2)
• Worker 3 — indexes 750,000 to 999,999
  
  Thread 0 — indexes 750,000 to 812,499
  Thread 1 — indexes 812,500 to 874,999
  Thread 2 — indexes 875,000 to 937,499
  Thread 3 — indexes 937,500 to 999,999

In this way, each thread in each worker executes the same number of loop iterations. Because the loop body’s running time is the same in every iteration, the default fixed schedule results in a balanced load.

I ran the π estimating program on the tardis cluster to study the program’s weak scaling performance. For the sequential version, I ran the PiSeq program from Chapter 8 on one node (one core). For the parallel version, I ran the PiClu program with one to ten worker tasks (12 to 120 cores). On one core, I ran it with 2, 4, 6, and 8 billion darts. On K cores, I ran it with K times as many darts as one core. Here are some of the commands I used:

$ java pj2 edu.rit.pj2.example.PiSeq 1234 20000000000
$ java pj2 workers=1 edu.rit.pj2.example.PiClu 1234 24000000000
$ java pj2 workers=2 edu.rit.pj2.example.PiClu 1234 48000000000
$ java pj2 workers=3 edu.rit.pj2.example.PiClu 1234 72000000000
$ java pj2 workers=4 edu.rit.pj2.example.PiClu 1234 96000000000

For each PiClu program run, I specified the number of worker tasks with the workers= option on the pj2 command line.

Figure 22.2 plots the running times, sizeups, and efficiencies I observed. The running time model is

$$ T = (0.235 + 3.04 \times 10^{-12} N) + (1.96 \times 10^{-8} N) \div K. \quad (22.1) $$

The sequential fractions range from 0.00166 to 0.0246, depending on the problem size and the number of cores.

**Under the Hood**

By calling the runInJobProcess() method, I specified the PiClu program’s reduction task to run in the job’s process on the frontend node. I did this to avoid tying up a backend node to run the reduction task. This is not the only reason one might want to run a task in the job’s process. Here are some other reasons.

Suppose a user runs a cluster parallel program needs to read an input file. The job process runs in the user’s account, and so has access to the user’s files. But the backend processes typically do not run in the user’s account; rather, they typically run in a separate Parallel Java account. This is for security; we don’t want different users running jobs on the cluster at the same time to be able to access or interfere with each other’s tasks. But this means that a task running in a backend process does not necessarily have access to the user’s account. In particular, a backend process might not be able to read
Figure 22.2. PiClu weak scaling performance metrics
an input file in the user’s account. One way to deal with this situation is for the user to make the input file readable by everyone. But again, for security, the user might not want to do this. An alternative is for the job to run the input file reading task in the job process. That way, the task’s process does have access to the user’s files. The input file reading task can then distribute the file’s contents to other tasks via tuple space.

Similarly, if the program needs to write an output file rather than printing its results on the console, the file writing task can be specified to run in the job process. Other tasks send data to the file writing task via tuple space.

Another use case is if the program needs to display a graphical user interface (GUI) while running—to show the progress of the computation, for example. The task that drives the GUI must run in the job’s process to be able to display windows on the user’s screen. Other tasks can send progress updates to the GUI task via tuple space. Such a GUI task could even accept input from the user: to change parameters while the program is running, to stop the program gracefully (as opposed to killing it with an interrupt), and so on.

Because they use reduction variables, the worker tasks each compute the semifinal result in parallel in multiple threads using a reduction tree (Figure 8.3). However, the reduction task’s final result is computed sequentially, not in parallel. Why? Because of how tuple space is implemented in the Parallel Java 2 Library, the semifinal result tuples are sent from the worker processes in the backend nodes, via the cluster’s backend network, to the job process in the frontend node, where the reduction task is running. The tuples have to go through the frontend node’s network interface one at a time. Consequently, there is nothing to be gained by reducing the semifinal results together in parallel on the frontend node, and a simple sequential reduction loop suffices. Because the overhead of doing the final reduction is miniscule compared to the worker tasks’ computation time, the program’s performance is degraded hardly at all.

**Points to Remember**

- In a cluster parallel program with reduction, use the hybrid model with multiple multicore worker tasks.
- Do the first level of reduction in shared memory within each task.
- Do the second level of reduction in a separate finish task.
- Use tuple space to send semifinal results from the worker tasks to the finish task.
- To get good performance, make sure the time spent on computation vastly outweighs the time spent on message passing and reduction.