Chapter 23
Cluster Load Balancing

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
Consider how the last few cluster parallel programs partitioned their computations among the worker tasks and the threads within each task. The bitcoin mining program in Chapter 21 partitioned the computations (nonces) among the tasks and threads using a leapfrog schedule. The π estimating program in Chapter 22 partitioned the computations (darts) among the tasks and threads using a fixed schedule. Fixed or leapfrog partitioning is appropriate when each loop iteration takes the same amount of time, so the load is inherently balanced.

Now let’s consider a problem in which each loop iteration does not take the same amount of time, so the load is unbalanced. In a single-node multicore parallel program, we saw that we could balance the load, and reduce the parallel program’s running time, by specifying a parallel for loop with a dynamic, proportional, or guided schedule. We can do the same in a cluster parallel program; the master portion of a master-worker cluster parallel for loop lets you specify a load balancing schedule when partitioning the loop iterations into chunks.

A hybrid parallel program, which has two levels of parallelism, has to do two levels of chunking: coarser-grained chunking at the task level, finer-grained chunking at the thread level within each task. An approach that usually works well is to partition the overall computation using a proportional schedule. This yields a number of chunks proportional to \( K \), the number of worker tasks, with each chunk having the same number of loop indexes:

\[
\text{Number of chunks} = K \times \text{chunkFactor}
\]

There’s a tradeoff, though. If the chunk factor is too large, there will be too many chunks, which will increase the message passing overhead—taking each chunk out of tuple space involves sending messages back and forth between the job process and the worker processes. But if the chunk factor is too small, there will be too few chunks to balance the load effectively. I’ve found that a chunk factor of 10 to 100 usually works well.

The master-worker pattern achieves coarse-grained load balancing at the task level. But any particular chunk of computation might itself exhibit an unbalanced load. So the hybrid parallel program also has to do fine-grained load balancing at the thread level within each task. This is achieved automatically by specifying the worker parallel for loop’s schedule—for example, by specifying the schedule and chunk arguments on the \( \text{pj2} \) command line, or by hard-coding these into the worker parallel for loop.

Let’s put these considerations into practice. Listing 23.1 gives the source code for TotientClu, a hybrid parallel version of the Euler totient program from Chapter 10. Recall that the Euler totient of \( n \), \( \Phi(n) \), is the number of numbers between 1 and \( n - 1 \) that are relatively prime to \( n \). The program partitions the numbers between 1 and \( n - 1 \) among the worker tasks, using the master-worker pattern for load balancing. Each task further partitions each
Chapter 23. Cluster Load Balancing

```java
class TotientClu
{
    // Job main program.
    public void main(String[] args)
    {
        // Parse command line arguments.
        if (args.length != 1) usage();
        long n = Long.parseLong(args[0]);

        // Set up a task group of K worker tasks.
        masterFor(2, n - 1, WorkerTask.class).args(""+n);

        // Set up reduction task.
        rule().atFinish().task(ReduceTask.class)
            .runInJobProcess();
    }

    // Print a usage message and exit.
    private static void usage()
    {
        System.err.println("Usage: java pj2 [workers=<K>] " +
            "edu.rit.pj2.example.TotientClu <n>"");
        System.err.println("<K> = Number of worker tasks " +
            "(default: 1)");
        System.err.println("<n> = Number whose totient to compute");
        terminate(1);
    }

    // Class TotientClu.WorkerTask provides a task that computes
    // chunks of iterations in the totient computation.
    private static class WorkerTask
    {
        // Worker task main program.
        public void main(String[] args) throws Exception
        {
            n = Long.parseLong(args[0]);

            phi = new LongVbl.Sum();
            factorize(n, nFactors);

            workerFor().exec(new LongLoop()
            {
                LongList iFactors;
            });
        }
    }
}
```

Listing 23.1. TotientClu.java (part 1)
chunk of work among its parallel team threads, using a parallel loop with a load balancing schedule. The threads’ results are combined within each task using multicore parallel reduction, and the workers’ results are combined using cluster parallel reduction.

The job’s command line argument is \( n \), the number whose totient is to be computed. Several \( pj2 \) options must also be specified. The \texttt{workers} option gives the number of worker tasks. The \texttt{masterSchedule} and \texttt{masterChunk} options specify how to partition the computation at the master level. The schedule and chunk options specify how to partition the computation at the worker level. These options must be specified appropriately to achieve a balanced load.

I ran the totient program on the \texttt{tardis} cluster to study the program’s strong scaling performance. I computed the totients of two numbers, \( n = 20,000,003 \) and \( n = 40,000,003 \) (both prime). For partitioning at the master level, I used a proportional schedule with a chunk factor of 100. For partitioning at the worker level, I used a guided schedule. To measure the sequential version, I ran the \texttt{TotientSeq} program from Chapter 10 on one node using commands like this:

\[
$ java \ pj2 \ debug=makespan \ edu.rit.pj2.example.TotientSeq \ \ 20000003
\]

To measure the parallel version on one core, I ran the \texttt{TotientClu} program with one worker task and one thread using commands like this:

\[
$ java \ pj2 \ debug=makespan \ workers=1 \ \ \ \text{masterSchedule=proportional} \ \ \text{masterChunk=100} \ \text{threads=1} \ \ \text{schedule=guided} \ \ \text{edu.rit.pj2.example.TotientClu} \ \ 20000003
\]

To measure the parallel version on multiple cores, I ran the \texttt{TotientClu} program with one to ten worker tasks and with all cores on each node (12 to 120 cores) using commands like this:

\[
$ java \ pj2 \ debug=makespan \ workers=1 \ \ \ \text{masterSchedule=proportional} \ \ \text{masterChunk=100} \ \ \text{schedule=guided} \ \ \text{edu.rit.pj2.example.TotientClu} \ \ 20000003
\]

Figure 23.1 plots the running times, speedups, and efficiencies I observed.

To derive a running time model, I need to know the problem size. There’s no simple relationship between \( n \), the number whose totient is being computed, and \( N \), the amount of computation needed to calculate \( \Phi(n) \). So I just used the sequential program’s running time in milliseconds as the problem size. This yielded \( N = 1.99 \times 10^5 \) and \( 5.35 \times 10^5 \) for \( n = 20,000,003 \) and \( 40,000,003 \), respectively. The running time model is

\[
T = (0.193) + (0.00161 + 1.21 \times 10^{-9} N)K + (2.52 + 9.29 \times 10^{-4} N)K. \quad (23.1)
\]

Notice that the running time model includes a term directly proportional to \( K \), the number of cores. As \( K \) increases, so does this term. Where does this
LongVbl thrPhi;
public void start()
{
    iFactors = new LongList();
    thrPhi = threadLocal (phi);
}
public void run (long i)
{
    if (relativelyPrime (factorize (i, iFactors),
                        nFactors))
        ++ thrPhi.item;
}
);
// Report result.
putTuple (phi);
}

// Store a list of the prime factors of x in ascending order
// in the given list.
private static LongList factorize
(long x,
 LongList list)
{
    list.clear();
    long p = 2;
    long psqr = p*p;
    while (psqr <= x)
    {
        if (x % p == 0)
        {
            list.addLast (p);
            x /= p;
        }
        else
        {
            p = p == 2 ? 3 : p + 2;
            psqr = p*p;
        }
    }
    if (x != 1)
        list.addLast (x);
    return list;
}

// Determine whether two numbers are relatively prime, given
// their lists of factors.
private static boolean relativelyPrime
(LongList xFactors,
 LongList yFactors)
{
    int xSize = xFactors.size();
    int ySize = yFactors.size();
    int ix = 0;
    int iy = 0;
    long x, y;
    while (ix < xSize && iy < ySize)
    {
Listing 23.1. TotientClu.java (part 2)
Figure 23.1. TotientClu strong scaling performance metrics
term come from? It comes from the parallel loop’s proportional schedule in the master. With a proportional schedule, the number of chunk tuples the master generates is proportional to $K$; therefore, the time needed to transfer these tuples from the frontend master (job) process to the backend worker (task) processes goes up as $K$ goes up. Still, this term only increases the running time by a fraction of a second.

Figure 23.1 shows that the efficiencies degrade as the number of cores increases. This is due to the $(a + bN)$ and $(c + dN)K$ terms in the running time model. The sequential fraction is determined to be $F = 0.00158$ for $n = 20,000,003$ and $F = 0.000598$ for $n = 40,000,003$. The sequential fraction is smaller for the larger problem size. Consequently, the efficiencies for the larger problem size do not degrade as quickly as the efficiencies for the smaller problem size, as is apparent in the plot.

Why is the sequential fraction smaller for the larger problem size? The sequential portion of the cluster totient program consists mainly of sending messages to transfer chunk tuples from the master to the workers, sending messages to transfer the result tuples from the workers to the reduce task,
performing the reduction, and printing the results. The parallelizable portion consists of examining all the numbers from 1 to \(n - 1\) to calculate the totient. For a given number of workers, the sequential portion’s running time is the same no matter what \(n\) is—because with a proportional schedule, the number of chunk tuples depends only on how many workers there are, and likewise for the number of result tuples. However, the parallelizable portion’s running time increases as \(n\) increases. Therefore, the program’s sequential fraction decreases as the problem size increases, and the program’s efficiency goes up.

These observations illustrate a characteristic of cluster parallel programs that involve message passing: The performance improves as the ratio of computation to communication goes up. Communication includes messages to carry out the master-worker load balancing as well as messages to report the worker tasks’ final results. Problems well-suited for a cluster parallel computer are those that require a lot of computation but only a little communication.

As a reminder, the approach the TotientClu program uses to calculate the Euler totient is rather inefficient. I’m using this as an example of why and how to do load balancing in a cluster parallel program, not as an example of how to calculate the totient efficiently. I chose to use an inefficient, computation-intensive, unbalanced-load algorithm to make my points about load balancing and computation-to-communication ratios.

**Points to Remember**

- In a cluster parallel program that needs load balancing, use the hybrid model with multiple multicore worker tasks.
- Do coarse-grained load balancing at the task level using the master-worker pattern.
- Do fine-grained load balancing at the thread level within each task by specifying the appropriate schedule for the parallel for loop.
- The job’s `main()` method partitions the computation and puts chunk tuples containing chunks of work into tuple space.
- The worker tasks repeatedly take chunk tuples out of tuple space and perform the indicated chunks of work.
- To get good performance, make sure the time spent on computation vastly outweighs the time spent on message passing—taking chunk tuples, putting result tuples.