Chapter 25
Interacting Tasks

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 tackle the $N$-body zombie program from Chapter 12 and develop a cluster parallel version. This program will be more challenging than the other cluster parallel programs we’ve seen, for several reasons. First, in the previous programs the outer loop had no sequential dependencies, so I was able to parallelize it using the master-worker cluster parallel loop pattern, and I was able to implement that pattern simply by calling the `masterFor()` and `workerFor()` methods. In the zombie program, the outer loop does have sequential dependencies, so I have to parallelize the middle loop rather than the outer loop; but as we will see, this is not quite as simple. Second, in the previous programs there were no data structures that had to be shared among the tasks of the job. In the zombie program there are shared data structures, namely the arrays holding the zombies’ $(x, y)$ coordinates; but it is not possible to directly share data among tasks running in separate processes. Third, in most of the previous cluster parallel programs, the only inter-task communication occurred at the end of the computation, when each worker sent its semifinal result to the reduce task. In the cluster parallel Mandelbrot Set program, the workers sent pixel rows to the output task, but the workers did not interact with each other. As we will see, the zombie program requires frequent and copious communication among the workers.

Following the hybrid parallel programming pattern on a cluster parallel computer (Figure 3.4), the computation is partitioned across multiple nodes, with one task (process) on each node and multiple threads in each process. Following the distributed memory pattern, the data—the zombies’ current and next positions—is likewise partitioned across the nodes. Partitioning the computation and the data, though, requires a bit of thought.

At each time step, the zombie program calculates the next position of each zombie. In the single-node multicore version, I partitioned the next-position computation among several threads (cores), each thread calculating just one “slice” of the next positions. I’ll do the same in the cluster version; each worker task will calculate just one slice of the next positions. But this means that each worker task needs to allocate storage only for its own slice of the next positions, not for all of them. On the other hand, calculating any zombie’s next position requires reading all the zombies’ current positions. Therefore, each worker task needs to allocate storage for all of the current positions. The storage allocation looks like Figure 25.1.

In this example, there are 24 zombies and four worker tasks. Each worker allocates all 24 elements for the current position arrays $x$ and $y$. (Each array element is labeled with its index.) Because calculating any zombie’s next position takes the same amount of time, partitioning the next positions equally among the workers will result in a balanced load. After partitioning, each worker ends up with a six-element slice. Thus, each worker allocates only six elements for the next position arrays $x_{\text{next}}$ and $y_{\text{next}}$. Note that in worker tasks 1 and higher, the $x_{\text{next}}$ and $y_{\text{next}}$ array indexes are not the
same as the corresponding zombie’s x and y array indexes; this subtle detail will turn up later when we write the code.

During each outer loop iteration (each time step), the middle loop iterates over just the worker’s own slice of the zombies, calculates those zombies’ next positions, and stores them in the x\textit{next} and y\textit{next} arrays. For each zombie in the worker’s slice, the inner loop iterates over all the zombies’ current positions in the x and y arrays. These calculations proceed independently in parallel in all the workers.

At the end of each outer loop iteration, the zombies’ just-calculated next positions need to become the current positions for the next time step. Each worker can update its own slice by copying the x\textit{next} and y\textit{next} arrays back to the proper indexes in the x and y arrays. Each worker must obtain the remaining slices of x and y from the other workers. Because the workers are in different processes, this has to be done through tuple space (message passing). Each worker puts multiple copies of a tuple into tuple space, one copy

---

**Figure 25.1.** Zombie program storage allocation
for each of the other workers, containing the worker’s own next positions. Each of the other workers takes one of these copies and stores its contents in the proper indexes in the \(x\) and \(y\) arrays. Figure 25.2 shows the tuples put by worker task 0 and taken by worker tasks 1 through 3. In a similar manner, worker tasks 1 through 3 put tuples containing their slices, and the other tasks take those tuples.

Each worker also puts one additional copy of its next positions tuple into tuple space. This extra copy is taken by a snapshot task at each time step. Before the first time step, at designated time steps during the calculation (if so specified on the command line), and after the last time step, the snapshot task prints the zombies’ positions.

Consequently, the zombie program’s inter-task communication is frequent (it happens on every time step) and copious (each tuple contains numerous zombie \((x, y)\) coordinates). In other words, the cluster parallel zombie program is tightly coupled. It will be interesting to see what effect the frequent and copious communication has on the program’s running time.

Turning to the code (Listing 25.1), the ZombieClu job main program sets up a start rule with a task group consisting of \(K\) worker tasks, where the number of workers \(K\) is specified on the \(pj2\) command line (lines 33–35). Each task’s command line arguments are the same as those of the job. The job main program also sets up a separate start rule with a snapshot task that will print snapshots of the zombie positions at the beginning, the end, and any intermediate time steps (lines 38–39). The snapshot task runs in the job process so it doesn’t tie up a backend node.

To convey the results of each time step from each worker task to the other tasks, the program uses the ZombieTuple subclass (line 64). The tuple has fields for the rank of the worker task that sent the tuple, the time step, the lower bound index of the worker’s slice, the zombies’ \((x, y)\) coordinates, and the “delta.” In addition to calculating the zombies’ next positions, the worker tasks also add up the total distance the zombies moved (the delta), in order to detect when the zombies reach the equilibrium state. However, with the data partitioned among the workers, each worker can calculate only its own part of the delta. To calculate the total delta, each worker must add in all the other worker’s partial deltas. Each worker can then determine whether the total delta is below the threshold, and terminate itself if so. Thus, along with the zombies’ next positions, each worker includes its partial delta in the zombie tuple sent to the other workers.

Because tuples are streamable objects, class ZombieTuple defines a no-argument constructor; the \texttt{writeOut()} method that writes all the fields; and the \texttt{readIn()} method that reads all the fields. Class ZombieTuple also overrides the \texttt{matchContent()} method. When deciding whether a certain target tuple matches a certain template, the job first checks whether the target is an instance of the template’s class or a subclass thereof. If so, the job then calls
Figure 25.2. Zombie program inter-task communication
the `matchContent()` method on the template, passing the target as the argument. If the method returns true, the target matches. The default implementation of the `matchContent()` method always returns true, so any target will match the template no matter what is stored in the target’s fields. This is not what I need for the zombie program. Instead, I overrode the `matchContent()` method so that a target will match a template only if the worker rank and time step fields are the same in the template and the target. This lets each worker match and take the correct zombie tuple from every other worker at each time step.

Next comes the worker task class (line 127). The first difference between this cluster version and the multicore version in Chapter 12 is the storage allocation for the zombies’ positions. The worker allocates storage for all \( n \) current positions in the \( x \) and \( y \) arrays (lines 176–177). To determine this worker’s slice of the next positions, the worker calls the `Chunk.partition()` method, passing in the lower and upper bounds of the entire index range (0 to \( n – 1 \)), the number of equal-sized slices into which to partition that range (i.e., the number of workers), and the index of the desired slice (i.e., this worker’s rank). From the Chunk object that is returned, the worker extracts the lower and upper bound indexes of the slice, \( lb \) and \( ub \), and the length of the slice, \( len \). The worker allocates storage for \( len \) next positions in the \( xnext \) and \( ynext \) arrays (lines 181–186).

Each worker task proceeds to set all the zombies’ current positions to their random initial positions (lines 189–194). Because every worker seeds its own pseudorandom number generator with the same seed, every worker’s PRNG generates exactly the same sequence of random numbers, and every worker ends up with exactly the same initial positions.

Each worker begins the triply-nested loop to calculate the zombies’ positions as they move—the sequential outer loop over the time steps (line 200); the parallel middle loop over the zombies in the worker’s slice, from \( lb \) to \( ub \) rather than 0 to \( n – 1 \) (line 205); and the sequential inner loop over all the zombies (line 221). Note the array index used to store the new \((x, y)\) coordinates into the \( xnext \) and \( ynext \) arrays (lines 235–236). The zombie index \( i \) has to be offset by the lower bound of the worker’s slice \( lb \) to get the proper index for the \( xnext \) and \( ynext \) arrays.

After the middle loop finishes, the just-calculated next positions must become the current positions. The multicore version did this by swapping the array references, but that won’t work in the cluster version. Instead, here is where each worker puts copies of a zombie tuple into tuple space, containing the worker’s slice of the next positions plus the worker’s partial delta (lines 248–249). Each worker puts \( size \) copies of the tuple: \( size – 1 \) copies for the other workers, plus one copy for the snapshot task. Each worker then takes zombie tuples from the other workers out of tuple space. Note how, in the `takeTuple()` method call, the template’s `step` and `rank` fields are set to the
Chapter 25. Interacting Tasks

305

package edu.rit.pj2.example;
import edu.rit.io.InStream;
import edu.rit.io.OutStream;
import edu.rit.pj2.Chunk;
import edu.rit.pj2.Job;
import edu.rit.pj2.Loop;
import edu.rit.pj2.Task;
import edu.rit.pj2.Tuple;
import edu.rit.pj2.vbl.DoubleVbl;
import edu.rit.util.Random;
import java.io.IOException;
import static java.lang.Math.*
public class ZombieClu
extends Job
{
    // Job main program.
    public void main
    (String[] args)
    {
        // Parse command line arguments.
        if (args.length != 9) usage();
        long seed = Long.parseLong (args[0]);
        int N = Integer.parseInt (args[1]);
        double W = Double.parseDouble (args[2]);
        double G = Double.parseDouble (args[3]);
        double L = Double.parseDouble (args[4]);
        double dt = Double.parseDouble (args[5]);
        double eps = Double.parseDouble (args[6]);
        int steps = Integer.parseInt (args[7]);
        int snap = Integer.parseInt (args[8]);

        // Set up a task group of K worker tasks.
        int K = workers();
        if (K == DEFAULT_WORKERS) K = 1;
        rule() .task (K, WorkerTask.class) .args (args);

        // Set up snapshot task.
        rule() .task (SnapshotTask.class) .args (args) .args (""+K)
            .runInJobProcess();
    }

    // Print a usage message and exit.
    private static void usage()
    {
        System.err.println ("Usage: java pj2 [workers=<K>] " +
            "edu.rit.pj2.example.ZombieClu <seed> <N> <W> <G> <L> " +
            "<dt> <eps> <steps> <snap>");
        System.err.println ("<K> = Number of worker tasks " +
            "(default: 1)"");
        System.err.println ("<seed> = Random seed");
        System.err.println ("<N> = Number of bodies");
        System.err.println ("<W> = Region size");
        System.err.println ("<G> = Attraction factor");
        System.err.println ("<L> = Attraction length scale");
        System.err.println ("<dt> = Time step size");
        System.err.println ("<eps> = Convergence length scale");
        System.err.println ("<steps> = Number of time steps " +
            "(0 = until convergence)");
    }

Listing 25.1. ZombieClu.java (part 1)
proper time step and worker rank; this ensures that the template will match the proper zombie tuple. Each worker stores this tuple’s contents into the proper indexes of the current position arrays, and sums up the partial deltas (lines 264–268). For the worker’s own rank, the worker just copies the next position arrays into the proper indexes of the current position arrays, without going through tuple space (lines 258–260).

One subtle detail: It is imperative that every worker sum up the partial deltas in the same order. Why? Because floating point arithmetic is not exact. If different workers summed up the partial deltas in different orders, different workers might end up with slightly different total deltas; one worker’s total delta might end up below the threshold while the other workers’ total deltas did not; one worker might therefore terminate while the other workers did not; and the program would come to a standstill. Summing up the partial deltas in the same order in every worker prevents this from happening.

At this point every worker’s current positions have been updated with their new values for the next time step. The worker checks for termination (lines 275–277) and, if it’s not time to terminate yet, goes on to the next iteration of the outer loop. Once the outer time step loop finishes, each worker’s run() method returns and the worker terminates.

Last comes the snapshot task (line 283). This task follows along with what the worker tasks are doing, time step by time step. The snapshot task begins by generating and printing all the zombies’ initial positions, using a PRNG initialized with the same seed as the worker tasks. The snapshot task then executes a loop over all the time steps. At each time step, the snapshot task takes a series of snapshot tuples out of tuple space, one from each worker task. By setting the step and rank fields properly in each takeTuple() method call’s template, the snapshot task ensures that it takes snapshot tuples in ascending order of time step, and within each time step in ascending order of worker task rank. At the specified snapshot intervals, the snapshot task prints the zombies’ positions as recorded in the workers’ zombie tuples. The snapshot task also adds up the workers’ partial deltas, and exits its outer loop under the exact same conditions as the workers. At this point the snapshot task terminates, the worker tasks have also terminated, and so the job terminates.

Whew! What a program. This is the sort of thing you have to write if you need to partition data across tasks (nodes) of a cluster parallel program, and if the tasks have to communicate data back and forth among themselves.

I ran the ZombieClu program on the tardis cluster to study its strong scaling performance. I ran the program on five problem sizes, $n = 200$, 300, 400, 600, and 800 zombies. I told the program to do $s = 10,000$ time steps (rather than stopping at equilibrium). I scaled the program up from one to ten workers (12 to 120 cores). Figure 25.3 plots the running times, speedups, and efficiencies I observed.
System.err.println("<snap> = Snapshot interval (0 = none)");
            terminate (1);
        }

        // Tuple with results of one time step.
        private static class ZombieTuple
            extends Tuple {
            public int rank; // Worker task rank
            public int step; // Time step number
            public int lb;   // Lower bound zombie index
            public double[] x; // Zombie X coordinates
            public double[] y; // Zombie Y coordinates
            public double delta; // Zombie position delta

            public ZombieTuple () {
            }

            public ZombieTuple (int rank,
                int step,
                int lb,
                double[] x,
                double[] y,
                double delta) {
                this.rank = rank;
                this.step = step;
                this.lb = lb;
                this.x = x;
                this.y = y;
                this.delta = delta;
            }

            public boolean matchContent (Tuple target) {
                ZombieTuple t = (ZombieTuple) target;
                return this.rank == t.rank && this.step == t.step;
            }

            public void writeOut (OutStream out)
                throws IOException {
                out.writeInt (rank);
                out.writeInt (step);
                out.writeInt (lb);
                out.writeDoubleArray (x);
                out.writeDoubleArray (y);
                out.writeDouble (delta);
            }

            public void readIn (InStream in)
                throws IOException {
        }

        }

Listing 25.1. ZombieClu.java (part 2)
Figure 25.3. ZombieClu strong scaling performance metrics
Chapter 25. Interacting Tasks

```java
rank = in.readInt();
step = in.readInt();
lb = in.readInt();
x = in.readDoubleArray();
y = in.readDoubleArray();
delta = in.readDouble();
}
}

// Worker task class.
private static class WorkerTask
    extends Task
{
    // Command line arguments.
    long seed;
    int N;
    double W;
    double G;
    double L;
    double dt;
    double eps;
    int steps;
    // Task group size and worker rank.
    int size;
    int rank;
    // Current zombie positions.
    double[] x;
    double[] y;
    // Next zombie positions.
    int lb, ub, len;
    double[] xnext;
    double[] ynext;
    // For detecting convergence.
    DoubleVbl delta = new DoubleVbl.Sum();

    // Task main program.
    public void main
        (String[] args)
        throws Exception
    {
        // Parse command line arguments.
        seed = Long.parseLong (args[0]);
        N = Integer.parseInt (args[1]);
        W = Double.parseDouble (args[2]);
        G = Double.parseDouble (args[3]);
        L = Double.parseDouble (args[4]);
        dt = Double.parseDouble (args[5]);
        eps = Double.parseDouble (args[6]);
        steps = Integer.parseInt (args[7]);
        // Get task group size and worker rank.
        size = groupSize();
        rank = taskRank();
```

Listing 25.1. ZombieClu.java (part 3)
The plots show that something is seriously askew with the ZombieClu program’s strong scaling behavior. As the number of cores increases, the running times initially decrease as expected, but then the running times turn around and start increasing! Correspondingly, the speedups hit a peak and then decrease as the number of cores increases. (I stretched the speedup plot’s vertical scale to emphasize the decreasing speedups.) The efficiencies are just bad. What is going on?

Fitting the running time model to the data yields this formula:

\[ T = (3.10 + 6.42 \times 10^{-10} N) + (0.161 + 2.61 \times 10^{-11} N) \cdot K + (2.15 \times 10^{-7} N) \div K. \]  

Plugging a certain problem size \( N \) into Equation (25.1) yields a running time formula with three terms: a constant term, a term directly proportional to the number of cores \( K \), and a term inversely proportional to \( K \). For the case with \( s = 10000 \) time steps and \( n = 200 \) zombies, the problem size is \( N = sn^2 = 4.00 \times 10^8 \), and the running time formula becomes

\[ T = 3.36 + 0.171 \cdot K + 86.0 \div K. \]  

Figure 25.4 plots these three terms separately in black, along with their sum \( T \) in red. Because the third term’s coefficient is so much larger than the other terms’ coefficients, the third term dominates for small \( K \) values, and \( T \) de-
// Allocate storage for current positions for all zombies.
x = new double [N];
y = new double [N];

// Allocate storage for next positions for just this
// worker's slice of zombies.
Chunk slice = Chunk.partition (0, N - 1, size, rank);
lb = slice.lb();
ub = slice.ub();
len = (int) slice.length();
xnext = new double [len];
ynext = new double [len];

// Initialize zombies' (x,y) coordinates.
Random prng = new Random (seed);
for (int i = 0; i < N; ++ i)
{
    x[i] = prng.nextDouble()*W;
y[i] = prng.nextDouble()*W;
}

// Do time steps.
t = 0;
ZombieTuple template = new ZombieTuple();
ZombieTuple zt = null;
for (;;)
{
    delta.item = 0.0;

    // Do each zombie i for this worker's slice of zombies.
    parallelFor (lb, ub) .exec (new Loop()
    {
        DoubleVbl thrDelta;
        public void start()
        {
            thrDelta = threadLocal (delta);
        }
        public void run (int i)
        {
            double vx = 0.0;
double vy = 0.0;
double dx, dy, d, v;

            // Accumulate velocity due to every other body j.
            for (int j = 0; j < N; ++ j)
            {
                if (j == i) continue;
dx = x[j] - x[i];
dy = y[j] - y[i];
d = sqrt(dx*dx + dy*dy);
v = G*exp(-d/L) - exp(-d);
vx += v*dx/d;
vy += v*dy/d;
            }

            // Move body i in the direction of its velocity.
    })
}
creases as $K$ increases. But as $K$ gets larger, the third term gets smaller, while the second term gets larger. Eventually the third term becomes smaller than the second term. After that, the running time $T$ increases along with the second term as $K$ increases.

How many cores does it take for $T$ to turn around and start increasing? Elementary calculus tells us that the minimum of a function occurs when the function’s derivative goes to zero. The derivative of Equation 25.2 is

$$dT/dK = 0.171 - 86.0/K^2.$$  \hspace{1cm} (25.3)

Setting $dT/dK = 0$ and solving yields $K = 22.4$ to minimize $T$. This agrees, roughly, with the ZombieClu program’s $T$ versus $K$ plot for 200 zombies.

We’ve seen that the term proportional to $K$ in the running time model eventually causes the running time to increase. But where is that term coming from? It’s coming from the inter-task communication. Specifically, it’s coming from the communication that results when the worker tasks put and take the zombie tuples at the end of every time step. The more worker tasks (cores) there are, the more tuples there are. Each tuple requires a certain amount of time to traverse the cluster’s backend network. Therefore, the more worker tasks there are, the more time the program spends on network communication transferring tuples. This accounts for the $0.171 \cdot K$ term in the running time model.

The lesson here is even more urgent than the lesson from the Mandelbrot Set program in Chapter 24: When doing strong scaling on a cluster parallel computer, you don’t necessarily want to run the program on all the cores in the cluster. Rather, you want to run the program on the number of cores that minimizes the running time. This might be fewer than the total number of cores. Measuring the program’s performance and deriving a running time model, as I did above, lets you determine the optimum number of cores to use.

But there’s another lesson. Under strong scaling, the ZombieClu program was able to scale up to only two or three nodes of the tardis cluster before the running times started increasing again. In general, the heavy communication in a tightly coupled cluster parallel program limits the parallelism the program can achieve. This is bad news. I really want to utilize all ten tardis nodes, not just two or three nodes.

If strong scaling doesn’t let the ZombieClu program utilize the entire tardis cluster effectively, what about doing weak scaling instead?

To study the ZombieClu program’s weak scaling performance, I ran the program on the tardis cluster using the same approach as in Chapter 14. I ran the program on five problem sizes on one core, $n = 200, 300, 400, 600,$ and 800 zombies. I told the program to do $s = 10,000$ time steps. I scaled the program up from one to ten workers (12 to 120 cores). As I ran with more
dx = vx*dt;
dy = vy*dt;
xnext[i-lb] = x[i] + dx;
ynext[i-lb] = y[i] + dy;

// Accumulate position delta.
thrDelta.item += abs(dx) + abs(dy);
}
});

// Advance to next time step.
++ t;

// Send new zombie positions to the other workers and to
// the snapshot task.
putTuple (size, new ZombieTuple
  (rank, t, lb, xnext, ynext, delta.item));

// Receive new zombie positions from the other workers.
double totalDelta = 0.0;
template.step = t;
for (int i = 0; i < size; ++ i)
{
  if (i == rank)
  {
    System.arraycopy (xnext, 0, x, lb, len);
    System.arraycopy (ynext, 0, y, lb, len);
totalDelta += delta.item;
  }
  else
  {
    template.rank = i;
    zt = takeTuple (template);
    System.arraycopy (zt.x, 0, x, zt.lb, zt.x.length);
    System.arraycopy (zt.y, 0, y, zt.lb, zt.y.length);
totalDelta += zt.delta;
  }
}

// Stop when position delta is less than convergence
// threshold or when the specified number of time steps
// have occurred.
if ((steps == 0 && totalDelta < eps) ||
  (steps != 0 && t == steps))
  break;

// Snapshot task class.
private static class SnapshotTask
  extends Task
  {
    // Task main program.
    public void main
      (String[] args)
      throws Exception
      {
  

Listing 25.1.  ZombieClu.java (part 5)
Figure 25.5. ZombieClu weak scaling performance metrics
// Parse command line arguments.
long seed = Long.parseLong (args[0]);
int N = Integer.parseInt (args[1]);
double W = Double.parseDouble (args[2]);
double eps = Double.parseDouble (args[6]);
int steps = Integer.parseInt (args[7]);
int snap = Integer.parseInt (args[8]);
int K = Integer.parseInt (args[9]);

// Print zombies' initial (x,y) coordinates.
Random prng = new Random (seed);
for (int i = 0; i < N; ++ i)
{
    double x_i = prng.nextDouble()*W;
    double y_i = prng.nextDouble()*W;
    System.out.printf ("0\t%d\t%g\t%g%n", i, x_i, y_i);
    System.out.flush();
}

// Do time steps.
int t = 0;
ZombieTuple template = new ZombieTuple();
ZombieTuple[] zt = new ZombieTuple [K];
for (;;)
{
    // Advance to next time step.
    ++ t;
    // Receive and print new zombie positions from the
    // workers.
    double totalDelta = 0.0;
    template.step = t;
    for (int i = 0; i < K; ++ i)
    {
        template.rank = i;
        zt[i] = takeTuple (template);
        totalDelta += zt[i].delta;
    }
    if (snap > 0 && t % snap == 0)
        snapshot (t, zt);
    // Stop when position delta is less than convergence
    // threshold or when the specified number of time steps
    // have occurred.
    if ((steps == 0 && totalDelta < eps) ||
        (steps != 0 && t == steps))
        break;
}

// Print zombies' final positions.
if (snap == 0 || t % snap != 0)
    snapshot (t, zt);

// Print a snapshot of the zombies' positions.
private static void snapshot
(int t,
workers, I increased \( n \) in proportion to the square root of the number of cores. Figure 25.5 plots the running times, sizeups, and efficiencies I observed.

An overall trend is apparent in the running time and efficiency data. As the number of cores increases, at first the running times stay approximately the same, as I would expect under weak scaling. But as the number of cores continues to increase, the running times go up. Correspondingly, the efficiencies go down, ending up around 55 to 80 percent at 120 cores depending on the problem size.

Why do the running times go up? Here is the fitted running time model:

\[
T = (4.81) + (0.446 + 3.31 \times 10^{-12} N) \cdot K + (1.39 + 2.15 \times 10^{-7} N) \div K
\]  

(25.4)

But with weak scaling, the problem size \( N \) increases as the number of cores \( K \) increases. For the zombie program running on one core, the problem size is \( N_1 = sn^2 \). On \( K \) cores, the problem size is \( N = KN_1 \). Substituting \( KN_1 \) for \( N \) in Equation 25.4 gives this alternative formula for the running time in terms of the problem size on one core and the number of cores:

\[
T = (4.81 + 2.15 \times 10^{-7} N_1) + (0.446) \cdot K + (3.31 \times 10^{-12} N_1) \cdot K^2
+ (1.39) \div K
\]  

(25.5)

Rather than \( T \) being constant, as would be the case for ideal weak scaling, \( T \) is actually a quadratic function of the number of cores, although with rather small coefficients on the terms proportional to \( K \) and \( K^2 \). Still, as \( K \) increases, those terms also increase, causing the running time to go up as is apparent in the plots.

Substituting a particular problem size \( N_1 \) into Equation 25.5 yields a formula for the running time that depends only on \( K \). For example, for \( n = 200 \) zombies, \( N_1 = 10,000 \cdot 200^2 = 4.00 \times 10^8 \), and \( T \) is about 90.8 + 0.446 \( \cdot \) \( K \). (The third and fourth terms are negligible in comparison to the first two terms for the number of cores we are considering.) Figure 25.6 plots these two terms separately in black, along with their sum \( T \) in red. Because the first term is so much larger than the second term’s coefficient, the first term dominates for small \( K \) values, and \( T \) stays constant as \( K \) increases. But as \( K \) gets larger, the second term gets larger. Eventually the second term becomes larger than the first term. After that, the running time \( T \) increases along with the second term as \( K \) increases.

What in the program is causing the running time to increase with \( K \)? As we saw with strong scaling, it is because, as the number of cores increases, the amount of inter-task communication increases. The number of zombie tuples sent and received through the cluster backend network is proportional to the number of worker tasks. Furthermore, the amount of data transmitted in all the zombie tuple increases too, because the number of zombies increases due to weak scaling as the number of cores increases. So as the number of
Chapter 25. Interacting Tasks

ZombieTuple[] zt)
{
    for (int i = 0; i < zt.length; ++ i)
    {
        ZombieTuple zt_i = zt[i];
        for (int j = 0; j < zt_i.x.length; ++ j)
            System.out.printf (%d	%d	%f\n%g%n",
                t, zt_i.lb + j, zt_i.x[j], zt_i.y[j]);
        System.out.flush();
    }
}

Listing 25.1. ZombieClu.java (part 7)

Figure 25.6. ZombieClu weak scaling running time model, 200 zombies
cores increases, there are more messages to put and take tuples containing more data, and it requires more time to send all the messages. This is time that cannot be parallelized. Thus, while the running time for the parallelizable computation stays the same due to weak scaling, the running time for the communication continually increases as the number of cores increases.

However, notice that the running time for the larger problem sizes does not go up as much as for the smaller problem sizes as $K$ increases. This is because the coefficient of the term proportional to $K$ in Equation 25.5 does not increase as the problem size $N_1$ increases. The first term in Equation 25.5, on the other hand, goes up as $N_1$ increases. For $n = 200$ zombies, $N_1 = 4.00 \times 10^8$, and $T$ is about $90.8 + 0.446\cdot K$. But for $n = 800$ zombies, $N_1 = 10,000 \cdot 800^2 = 6.40 \times 10^9$, and $T$ is about $1,380 + 0.446\cdot K$. For the larger problem size, the second term is much smaller relative to the first term, so the running time increase with $K$ is relatively not as large.

Consequently, the efficiencies for the larger problem sizes are better than for the smaller problem sizes. If I had calculated 5,000 or 10,000 zombies on one core and had done weak scaling onto more cores, I would have seen very good efficiencies, even with all the inter-task communication.

Putting it another way, the amount of computation in the zombie program is proportional to the square of the number of zombies: at each time step, the program has to do $n^2$ iterations of the innermost loop. On the other hand, the amount of communication in the zombie program is proportional just to the number of zombies: at each time step, the $n$ zombies’ positions are communicated among the tasks. Therefore, as $n$ increases, the ratio of communication to computation goes down, the relative overhead due to communication goes down, and the efficiencies droop to a lesser extent. As I’ve said before, to get good efficiencies in a cluster parallel program with message passing, the amount of computation needs to be large relative to the amount of communication.

Back at the end of Chapter 20 I said that tightly coupled cluster parallel programs, like the ZombieClu program, are outside Parallel Java 2’s design center. Now we can see why. The ZombieClu program works perfectly; it calculates the correct series of zombie positions. But the ZombieClu program does not scale well due to its large amount of communication. It’s possible to get good scaling with this program, but to do so I have to calculate very large problem sizes. To get the 80 percent efficiency I observed on 120 cores with weak scaling, for example, I had to compute $800 \cdot 120^{1/2} = 8,764$ zombies. The program would scale better if implemented using a high-performance message passing library like MPI. Even with MPI, though, the program would still eventually run into scaling issues.

Tightly coupled parallel programs, like the ZombieClu program in particular and $N$-body programs in general, are really best suited to run on a single multicore node, like the ZombieSmp program in Chapter 12. There, each
thread can directly access all the current positions in shared memory, without
needing time-consuming message passing. Multicore machines nowadays
have much larger main memories and many more cores (including GPU ac-
celerator cores) than they did decades ago when MPI was invented, so a sin-
gle multicore node now can solve much larger problems (because of larger
main memories) at a much larger computation rate (because of more cores)
than back then. Still, a problem too large for a single multicore node has to
be run on a cluster, with all the programming difficulty and message passing
overhead that entails.

Under the Hood

The worker tasks in the ZombieClu program were specified as a task
group in a single rule in the job main program. Consequently, the Tracker
schedules all the worker tasks as a unit. The Tracker will not start any of the
worker tasks until there are enough idle nodes to start all the worker tasks,
one worker on each node. Because every worker task communicates with ev-
ey other worker task, it is critical that they all be started as a unit, otherwise
the program would come to a standstill.

As we saw in previous chapters, class edu.rit.pj2.Chunk provides the
chunk tuple used in a master-worker cluster parallel for loop to specify a
chunk of loop iterations, where the loop index is type int. Class edu.rit.pj2-
.LongChunk does the same for a loop index of type long. These classes also
provide the static partition() method, which partitions a given index range
in the same manner as a fixed schedule: one chunk for each of a given num-
ber of workers, each chunk (except possibly the last) having the same num-
ber of indexes. The ZombieClu program uses the chunk returned by the par-
tition() method to partition the zombie next position arrays among the
worker tasks (the length of the next position arrays is that of the chunk). The
ZombieClu program also uses this chunk to partition the middle loop indexes
among the worker tasks (the lower and upper loop index bounds are those of
the chunk). Thus, the ZombieClu program partitions the middle loop using, in effect, a fixed schedule at the job level. Each worker’s parallel loop
in turn partitions its index range among the threads of the task using the de-
fault fixed schedule. This fixed partitioning is appropriate because the com-
putation is inherently balanced.

At each time step, each worker task puts $K$ copies of a zombie tuple into
tuple space, where $K$ is the number of workers. This is done by specifying
the number of copies as the first argument of the putTuple() method call
(line 248). However, to reduce the number of bytes sent over the network,
and hence reduce the amount of time spent, this method call sends only one
message, not $K$ messages. The message consists of the tuple’s contents and
the number of copies ($K$). The job process receives the put-tuple message. To
reduce the number of bytes of storage allocated in the job process, the job stores $K$ references to the one tuple in the job’s internal tuple list. When a worker sends a take-tuple message to the job process, the job process sends the tuple back to the worker and removes one of the references. After all $K$ references have gone away, the tuple goes away. In this manner, the tuple contents are transmitted over the network only once and occupy storage in the job process only once, no matter how many “copies” are put into tuple space.

**Points to Remember**

- Partition the program’s data among the worker tasks where necessary. Use the `partition()` method of class Chunk or LongChunk to do the partitioning.
- When tasks need to communicate data among each other during a computation, the sending task wraps the data in a tuple and puts the tuple into tuple space; the receiving task takes the tuple out of tuple space and extracts the data.
- Such tuple classes must include fields that identify which part of the computation the data is coming from, in addition to fields holding the data itself.
- Such tuple classes must override the `matchContent()` method to check whether the identifying fields in the target equal the identifying fields in the template.
- To get good performance in a cluster parallel program with tasks that communicate during the computation, the problem size must be large enough so that the time spent in computation vastly exceeds the time spent in communication.
- Weak scaling might be more effective than strong scaling at achieving a large ratio of computation to communication.