Chapter 8
Sequential Dependencies

► Part I. Preliminaries
▼ Part II. Tightly Coupled Multicore
    Chapter 2. Parallel Loops
    Chapter 3. Parallel Loop Schedules
    Chapter 4. Parallel Reduction
    Chapter 5. Reduction Variables
    Chapter 6. Load Balancing
    Chapter 7. Overlapping
  Chapter 8. Sequential Dependencies
    Chapter 9. Strong Scaling
    Chapter 10. Weak Scaling
    Chapter 11. Exhaustive Search
    Chapter 12. Heuristic Search
► Part III. Loosely Coupled Cluster
► Part IV. GPU Acceleration
► Part V. Map-Reduce
So-called *N-body problems* pop up now and then when studying natural phenomena. The behavior of flocks of birds, schools of fish, insect swarms, and stars in a star cluster or galaxy can be modeled as *N*-body problems and computed with *N*-body programs. Our next parallel program is an *N*-body program where the bodies literally are corpses, or more precisely, zombies. I could have based this example on birds or stars, but zombies are cooler. (The zombie model in this chapter is actually based on a model of locust swarms.)*

Initially, *N* zombies are scattered at random locations within a certain area; this is the *initial state*. The zombies want to get together, so they all start moving towards each other. However, the zombies don’t want to be too close to one another. Each zombie continues moving until its attraction to the other zombies is exactly counterbalanced by its repulsion from nearby zombies. Thereafter, the zombies stay put; this is the *equilibrium state*. I want to write a program that, given the zombies’ initial state and certain other parameters, computes the zombies’ positions as a function of time (Figure 8.1).

The zombies are indexed from 0 to *N* – 1. The zombies move in a two-dimensional plane. At any given instant in time, zombie *i*’s position is (*x*<sub>i</sub>, *y*<sub>i</sub>). Consider just two zombies at indexes *i* and *j*. The distance between zombie *i* and zombie *j* is the Euclidean distance:

\[ d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \]  

(8.1)

Zombie *i* moves toward or away from zombie *j* at a velocity given by

\[ v_{ij} = G \cdot \exp\left(-\frac{d_{ij}}{L}\right) - \exp\left(-\frac{d_{ij}}{L}\right) \]  

(8.2)

A positive velocity means zombie *i* moves toward zombie *j* (attraction), a negative velocity means zombie *i* moves away from zombie *j* (repulsion). The first term in equation (8.2) models attraction, the second term models repulsion. *G* and *L* are constants that determine the strength of the attraction. Here is a plot of velocity versus distance for *G* = 0.5 and *L* = 10:

As the two zombies move closer together, the attractive velocity increases. But as they move still closer, the attractive velocity diminishes. If they get

---

Figure 8.1. Snapshots of zombie positions, $N = 100$
too close, the velocity reverses and becomes repulsive.

The velocity from zombie \( i \) towards zombie \( j \) is actually a vector quantity \((v_{xij}, v_{yij})\) whose magnitude is \( v_{ij} \). The velocity vector’s X and Y components are

\[
\begin{align*}
  v_{xij} &= v_{ij} \cdot \frac{x_j - x_i}{d_{ij}}, \\
  v_{yij} &= v_{ij} \cdot \frac{y_j - y_i}{d_{ij}}.
\end{align*}
\] (8.3)

Now consider all \( N \) zombies. The net velocity of zombie \( i \) \((vx_i, vy_i)\) is the vector sum of the velocities with respect to all the other zombies:

\[
\begin{align*}
  vx_i &= \sum_{j=0, j \neq i}^{N-1} vx_{ij}, \\
  vy_i &= \sum_{j=0, j \neq i}^{N-1} vy_{ij}.
\end{align*}
\] (8.4)

Given zombie \( i \)’s position \((x_i, y_i)\) at a certain instant, zombie \( i \)’s position a small amount of time \( dt \) later \((nextx_i, nexty_i)\) is obtained by multiplying zombie \( i \)’s velocity times \( dt \) and adding that to zombie \( i \)’s current position:

\[
\begin{align*}
  nextx_i &= x_i + vx_i \cdot dt, \\
  nexty_i &= y_i + vy_i \cdot dt.
\end{align*}
\] (8.5)

Calculating the next positions of all the zombies using the preceding formulas, and then replacing the current positions with the next positions, is called “taking a time step.” By taking a series of time steps, I can plot out the zombies’ positions as a function of time. I’ll stop taking time steps when the zombies reach the equilibrium state; that is, when the zombies are no longer changing their positions. I’ll detect this by adding up the absolute values of the distances the zombies move:

\[
\Delta = \sum_{i=0}^{N-1} |vx_i \cdot dt| + |vy_i \cdot dt|.
\] (8.6)

When \( \Delta \) becomes less than a certain tolerance \( \epsilon \), I’ll stop. (\( \Delta \) will never become exactly 0, so I’ll stop when \( \Delta \) becomes too small to notice.)

Putting it all together, here is the pseudocode for the zombie program.

Initialize zombie positions
Repeat: (time step loop)
  For \( i = 0 \) to \( N - 1 \):
    \((vx_i, vy_i) \leftarrow (0, 0)\)
  For \( j = 0 \) to \( N - 1, j \neq i \):
    Compute \((vx_{ij}, vy_{ij})\) using equations (8.1), (8.2), and (8.3)
    \((vx_i, vy_i) \leftarrow (vx_i + vx_{ij}, vy_i + vy_{ij})\)
    \((nextx_i, nexty_i) \leftarrow (x_i + vx_i \cdot dt, y_i + vy_i \cdot dt)\)
Replace current zombie positions with next zombie positions
package edu.rit.pj2.example;
import edu.rit.pj2.Loop;
import edu.rit.pj2.Task;
import edu.rit.pj2.vbl.DoubleVbl;
import edu.rit.util.Random;
import static java.lang.Math.*;
public class ZombieSmp
    extends Task
{
    // Command line arguments.
    long seed;
    int N;
    double W;
    double G;
    double L;
    double dt;
    double eps;
    int steps;
    int snap;

    // Current body positions.
    double[] x;
    double[] y;

    // Next body positions.
    double[] xnext;
    double[] ynext;

    // For detecting convergence.
    DoubleVbl delta = new DoubleVbl.Sum();

    // Task main program.
    public void main
        (String[] args)
    {
        // Parse command line arguments.
        if (args.length != 9) usage();
        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]);
        snap = Integer.parseInt (args[8]);

        // Set up N bodies' initial (x,y) coordinates at random in a
        // WxW square region.
        x = new double [N];
        y = new double [N];
        xnext = new double [N];
        ynext = new double [N];
        Random prng = new Random (seed);
        for (int i = 0; i < N; ++ i)
        {
            x[i] = prng.nextDouble()*W;
            y[i] = prng.nextDouble()*W;
        }

Listing 8.1. ZombieSmp.java (part 1)
If $\Delta < \varepsilon$:
   Exit time step loop

Note the doubly-nested loop over the zombies within each time step. The outer loop has $N$ iterations, the inner loop also has $N$ iterations (one of which does no calculations), so each time step requires $O(N^2)$ computations. To get accurate results, the time step size $dt$ must be very, very small. Consequently, the program will have to take many, many time steps to reach equilibrium. The $N$-body program can easily require enormous amounts of computation. It is a prime candidate for parallelization.

The zombie $N$-body algorithm consists of three nested loops: the outer loop over the time steps, the middle loop to calculate each zombie’s next position, and the inner loop to calculate each zombie’s velocity with respect to every other zombie. The Mandelbrot Set algorithm in Chapter 7 also had nested loops, the outer loop over the pixel rows and the inner loop over the pixel columns. We were able to make the Mandelbrot Set’s outer loop a parallel loop because the outer loop had no sequential dependencies—each pixel could be calculated independently of all the other pixels, so the pixel rows could all be calculated in parallel. That is not the case with the $N$-body algorithm. This time, the outer loop does have sequential dependencies. It is not possible to calculate each time step independently of all the other time steps. Rather, we must wait for the preceding time step to finish before commencing the next time step, because we need the zombie positions from the previous time step to calculate the zombie positions for the next time step. So the outer loop must remain a regular, non-parallel loop.

The middle loop over the zombies, however, does not have sequential dependencies; the next position of each zombie can be calculated independently of all the other zombies. So to parallelize the algorithm, I can make the middle loop a parallel loop:

```
Initialize zombie positions
Repeat: (time step loop)
   Parallel for $i = 0$ to $N - 1$:
      $(vx_i, vy_i) \leftarrow (0, 0)$
      For $j = 0$ to $N - 1$, $j \neq i$:
         Compute $(vx_{ij}, vy_{ij})$ using equations (8.1), (8.2), and (8.3)
         $(vx_i, vy_i) \leftarrow (vx_i + vx_{ij}, vy_i + vy_{ij})$
         $(nextx_i, nexty_i) \leftarrow (x_i + vx_i \cdot dt, y_i + vy_i \cdot dt)$
      Replace current zombie positions with next zombie positions
      If $\Delta < \varepsilon$:
         Exit time step loop
```

Once I’ve parallelized the middle loop, there’s no point in parallelizing the inner loop. So the inner loop remains a regular loop. Each parallel team thread performs a subset of the iterations over $i$ and all the iterations over $j$. 
// Snapshot all bodies' initial positions.
int t = 0;
snapshot (t);

// Do time steps.
for (;;)
{
    delta.item = 0.0;

    // Do each body i.
    parallelFor (0, N - 1).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.
            dx = vx*dt;
            dy = vy*dt;
            xnext[i] = x[i] + dx;
            ynext[i] = y[i] + dy;

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

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

    // Update positions.
    double[] tmp;
    tmp = x; x = xnext; xnext = tmp;
    tmp = y; y = ynext; ynext = tmp;

Listing 8.1. ZombieSmp.java (part 2)
Listing 8.1 gives the parallel zombie program. It is a straightforward translation of the above pseudocode into Java. The program’s command line arguments are

- **seed**—Seed for a pseudorandom number generator for initializing the zombies’ positions
- **N**—Number of zombies
- **W**—Size of the initial region in which to place the zombies
- **G**—Attraction factor used in equation (8.2)
- **L**—Attraction length scale used in equation (8.2)
- **dt**—Time step size
- **eps**—Convergence threshold \( \varepsilon \)
- **steps**—Number of time steps (0 means iterate until convergence)
- **snap**—Snapshot interval (0 means no snapshots)

If a nonzero `steps` parameter is specified on the command line, the program computes the specified number of time steps, whether or not the zombie positions have converged. Otherwise, the program iterates for as many time steps as necessary to achieve convergence (lines 120–122).

The program prints a snapshot of the zombies’ initial positions (line 63) and final positions (line 129). If a nonzero `snap` parameter is specified on the command line, the program also prints a snapshot of the zombies’ positions every `snap` time steps (line 125). The zombie positions plotted in Figure 8.1 were computed by this program run:

```
$ java pj2 edu.rit.pj2.example.ZombieSeq 142857 100 5 0.5 10 0.00001 0.001 0 3000
```

Each iteration of the middle loop, which calculates the next position of one zombie, does exactly the same number of computations as every other iteration. Thus, each middle loop iteration takes the same amount of time, so the load is inherently balanced, so the parallel for loop can use the default fixed schedule.

The `delta` variable (line 30), which holds the quantity \( \Delta \) used to detect convergence, is a global reduction variable of type `DoubleVbl`. It is initialized to an instance of class `DoubleVbl.Sum`, which does a sum-reduce. Following the parallel reduction pattern, each parallel team thread has a thread-local `thrDelta` variable that accumulates \( \Delta \) for that thread’s subset of the iterations. The thread-local \( \Delta \) values are automatically summed together into the global reduction variable as the middle parallel for loop finishes.

The outer loop body (lines 66–126), which computes one time step, consists of an initial section executed by a single thread (line 68), a parallel for loop executed by multiple threads (lines 71–107), and a final section executed by a single thread again (lines 110–125). The final section is not executed until all the parallel team threads have arrived at the implicit barrier at
// Stop when position delta is less than convergence
// threshold or when the specified number of time steps
// have occurred.
if ((steps == 0 && delta.item < eps) ||
    (steps != 0 && t == steps))
    break;

// Snapshot all bodies' positions every <snap> time steps.
if (snap > 0 && (t % snap) == 0) snapshot(t);
}

// Snapshot all bodies' final positions.
snapshot(t);

// Snapshot all bodies' positions.
private void snapshot
    (int t)
{
    for (int i = 0; i < N; ++ i)
        System.out.printf("
%d\t%d\t%g\t%g
", t, i, x[i], y[i]);
    System.out.flush();
}

// Print a usage message and exit.
private static void usage()
{
    System.err.println("Usage: java pj2 " +
        "edu.rit.pj2.example.ZombieSmp <seed> <N> <W> <G> <L> " +
        "<dt> <eps> <steps> <snap>");
    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 threshold");
    System.err.println("<steps> = Number of time steps (0 = " +
        "until convergence")");
    System.err.println("<snap> = Snapshot interval (0 = none)");
    throw new IllegalArgumentException();
}

Listing 8.1. ZombieSmp.java (part 3)
the end of the parallel for loop, signaling that the zombies’ next position calculations are all complete. This barrier synchronization is essential if the program is to compute the correct results. It is also essential that the final section—checking for convergence, updating the zombies’ positions, and printing a snapshot—be done outside the parallel for loop in a single thread.

The sections of the outer loop body executed by a single thread don’t get sped up when running on multiple cores. Thus, it’s important to minimize the time spent in these single-threaded sections. For this reason, the program updates the zombie positions (lines 113–115) simply by swapping the array references, so that \( x \) and \( y \) refer to the arrays that used to be \( \text{nextx} \) and \( \text{nexty} \), and vice versa. This is faster than copying the array elements, especially if \( N \) is large.

One detail about the snapshots: The \texttt{snapshot()} method, after printing all the zombies’ positions, calls \texttt{System.out.flush()} (line 138). Why? Because if it didn’t call \texttt{flush()}, the snapshot printouts would get stored in an internal buffer and would be printed all at once at the end of the program. This is not what I want. I want each snapshot to appear as soon as its time step has finished, so I can tell that the program is making progress. Calling \texttt{flush()} makes the printout appear on the standard output stream immediately.

Here are the sequential and parallel zombie programs’ running times on a four-core \texttt{tardis} node for \( N = 200 \) zombies:

\begin{verbatim}
$ java pj2 debug=makespan edu.rit.pj2.example.ZombieSeq  \\
  142857 200 5 0.5 10 0.00001 0.001 0 0  \\
  > /var/tmp/ark/zom200_0.txt
Job 14 makespan 84670 msec

$ java pj2 debug=makespan edu.rit.pj2.example.ZombieSmp  \\
  142857 200 5 0.5 10 0.00001 0.001 0 0  \\
  > /var/tmp/ark/zom200_4.txt
Job 15 makespan 23566 msec
\end{verbatim}

The program required 10653 time steps to reach the equilibrium state. Thus, the program had to do \( 10653 \times 200^2 = \text{over 426 million calculations} \) (executions of the inner loop body).

The speedup factor was \( 84640 \div 23566 = 3.592 \), not that good. Both the Mandelbrot Set program and the zombie program experienced less-than-ideal speedups when run on four cores. We’ll explore the reasons for this in the next two chapters.

\section*{Under the Hood}

The outer non-parallel for loop at line 66 has a nested parallel for loop at line 71. This means that \textit{on every outer loop iteration}, the program needs a multiple-thread team to execute the parallel for loop. If these threads were created and destroyed each time through the outer loop, the thread creation
and destruction overhead would severely reduce the program’s performance.

Instead, the Parallel Java 2 Library uses a thread pool under the hood. Initially, the pool is empty. When a thread team is needed, the program first tries to find a team in the pool that has the required number of threads. If such a team is in the pool, the program reuses that team. Only if there is no suitable team in the pool does the program create a new team with new threads. When a team finishes executing a group of parallel sections or a parallel for loop, the program puts the team back in the pool so the team can be reused later. This minimizes the overhead when repeatedly executing parallel sections or parallel for loops.

**Points to Remember**

- Carefully consider whether each loop in the program has sequential dependencies.
- A sequential dependency exists when a later loop iteration needs to use results computed in an earlier loop iteration.
- If a loop has sequential dependencies, the loop must remain a regular non-parallel loop.
- If a loop does not have sequential dependencies, the loop can be converted into a parallel loop.
- Parallel sections, regular loops, and parallel loops can be nested together, following the algorithm’s natural structure.
- The implicit barrier at the end of each group of parallel sections and at the end of each parallel for loop ensures synchronization between the team threads executing the parallel code and the thread executing the enclosing sequential code.
BIG CPU, BIG DATA